PROVENANCE IN DATA-ORIENTED WORKFLOWS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Robert Ikeda
November 2012

This dissertation is online at: http://purl.stanford.edu/wp108gc6672

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Anish Das Sarma

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Data-processing tasks are commonly managed using data-oriented workflows, in which input data sets are processed by a graph of transformations to produce output data. In data-oriented workflows, it can be useful to track data provenance (also sometimes called lineage), which describes where data came from and how it has been manipulated and combined.

We begin by giving a new general definition of provenance, introducing the notions of correctness, precision, and minimality. We then:

1) Describe a wrapper-based approach for capturing provenance in workflows in which all transformations are either map or reduce functions
2) Describe a provenance-based approach for selectively refreshing one or more elements in the output data, i.e., computing the latest values of particular output elements based on modified input data
3) Show how logical provenance, i.e., provenance information stored at the transformation level, can often capture precise provenance relationships in a compact fashion
4) Describe our prototype system called Panda (for Provenance And Data) that supports refresh in data-oriented workflows, as well as debugging and drill-down using logical provenance

Overall, our work provides a comprehensive foundation, set of algorithms, and prototype system for provenance in data-oriented workflows.

Acknowledgments

I would first like to thank my advisor, Jennifer Widom, for all of her support during my time at Stanford. She has been warm and welcoming since our first meeting. I remember sitting in research group discussions my first year not knowing anything, but Jennifer never made me feel like I didn't belong. She pushed me to work hard, and she definitely made me a more effective researcher. Thank you for being a great advisor.

I would like to thank Hector Garcia-Molina for being my advisor during my first two years at Stanford and for introducing me to research. He taught me through example how to organize my thoughts, improve my writing, and come up with new research ideas. Even after my research interests evolved, he has been supportive throughout. I'm glad I got to spend those first two years as his student.

I would like to thank Anish Das Sarma for being so generous with his help during my first few years at Stanford. I learned a lot from him and am thankful that he agreed to be on my thesis committee. When I arrived at Stanford, Anish was the senior student in Jennifer's group, and I did my best to learn from him. He was always willing to answer my questions, and he helped me gain confidence in my abilities. It was a lot of fun having him in the InfoLab.

I want to thank Michael Genesereth and David Rehkopf for serving on my oral committee, and Nigam Shah for helping my defense go smoothly. I am very thankful for how that day went.

I worked with some awesome collaborators during my time at Stanford. I want to thank some of them: Parag Agrawal, Junsang Cho, Akash Das Sarma, Charlie Fang, Philip Guo, Austin Haugen, Sean Kandel, Diana MacLean, Abhijeet Mohapatra,

Raghotham Murthy, Aditya Parameswaran, Hyunjung Park, Alkis Polyzotis, Semih Salihoglu, and Satoshi Torikai. I also want to thank everyone in the InfoLab for their help, especially Marianne Siroker.

I want to thank my friends for helping make my Ph.D. experience a lot of fun, and my family for being there for me. Finally, I want to thank my wife, Priscilla, for being such a great partner and friend.

Contents

List of Figures

Chapter 1

Introduction

To perform data-processing tasks such as analyzing scientific data [13, 23, 37] or government statistics [1, 3], users often execute a number of processing steps, including data extraction, integration of multiple sources, aggregation, and ad-hoc transformations and queries. Such analyses are commonly managed using *data-oriented workflows*, in which input data sets are processed by a graph of *transformations* to produce output data.

In data-oriented workflows, it can be useful to track *data provenance* (also sometimes called *lineage*). In its most general form, provenance describes where data came from, how it was derived, manipulated, and combined, and how it has been updated over time. Provenance can serve a number of important functions:

- **Explanation.** Users may be particularly interested in specific portions of a derived data set. Provenance supports "drilling down" to examine the sources and evolution of data elements of interest, enabling a deeper understanding of the data.
- **Verification.** Output data may appear suspect—due to possible bugs in data processing and manipulation, because the data may be stale, or even due to maliciousness. Provenance enables auditing how data was produced, either for verifying its correctness, or for identifying the erroneous or outdated source data or transformations that are responsible for erroneous or outdated output data.

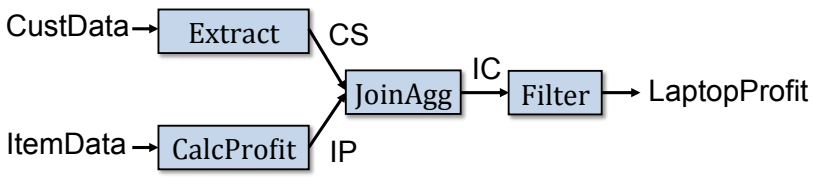

Figure 1.1: Profit calculation workflow example.

- **Recomputation.** Having found outdated or incorrect source data, or transformations that are buggy or have been modified, we may want to propagate corrections or changes to all "downstream" data that are affected. Provenance helps us recompute only those data elements that are affected by corrections.

1.1 Running Example

We present a contrived running example crafted for illustrative purposes. This example appears on and off throughout the thesis. Consider "WebShop," an online reseller. WebShop periodically runs a workflow, shown in Figure 1.1, 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.

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)

Figure 1.2: Profit calculation workflow, sample data.

- 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.
- Finally, transformation **Filter** filters ItemCountryProfit, selecting tuples with type equal to 'laptop', producing table LaptopProfit(item-id, country, brand, profit), abbreviated LP.[1]

Figure 1.2 shows sample input data sets along with all intermediate data, and finally output table LaptopProfit.

We use this workflow example to illustrate the applications of provenance introduced earlier. Provenance itself is discussed next in Section 1.2.

- **Explanation.** Suppose a WebShop analyst runs the workflow and then wants to learn more about the profits on sales of the laptop I1 in Germany. By tracing the provenance of LaptopProfit element (2) back through the workflow to CustSales, the analyst can identify the WebShop customers in Germany who have purchased the item.
- **Verification.** Suppose the analyst finds that French profits on sales of the laptop I3 are surprisingly low. Noticing this anomalous result, the analyst would like to find out why these profits were so low. Tracing the provenance of LaptopProfit element (3) back through the workflow to ItemData, the analyst discovers that the profit per laptop (10, which seems 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. Upon further investigation, the analyst discovers that ItemData element (3) is indeed out of date and updates this element with the latest profit data. Note that both verification and explanation involve "drilling down" to understand how the output data was produced. While explanation simply allows a deeper understanding of the output data, verification can identify the erroneous source data responsible for errors in the output.

[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.

- **Recomputation.** Having updated the data in ItemData element (3), the analyst would like to compute the correct updated value for LaptopProfit element (3) without having to rerun the entire workflow.

To support the above activities, a workflow provenance system needs to support both *provenance capture* and *operations* on the provenance. Before we describe these aspects in more detail, we first discuss how provenance might be expressed.

1.2 Provenance

After discussing the different ways to express provenance (Section 1.2.1), we describe at a high level how to capture provenance for the transformations in a workflow (Section 1.2.2). We then describe the provenance-based operations that enable the functionality presented in the running example (Section 1.2.3).

1.2.1 Expressing Provenance

Provenance Model

One of the most important first steps in expressing provenance is to select a *provenance model*. Possible models range from a bipartite graph structure connecting input and output data elements, to the much higher-level and more "semantic" *Open Provenance Model* [5]. We are interested in data-oriented provenance, so in all of our work, provenance can be thought of as (or mapped to) a bipartite graph, i.e., mappings between input and output data elements. As discussed below, we do not always store the bipartite graph explicitly.

Provenance Storage

To illustrate different possible techniques for storing provenance, consider transformation **JoinAgg** from Figure 1.1 with input data set CustSales and output data set ItemCountryProfit. (To simplify our discussion, for now we ignore input set ItemProfit.) Let output element o = ItemCountryProfit(4), i.e., the fourth element from

data set ItemCountryProfit in Figure 1.2. The input elements from CustSales that contributed to o are CustSales(2) and CustSales(5). One way to store this information is to use *physical provenance*, which requires each output element to be annotated with some type of identifier for the contributing input elements. Assuming that CustSales has an ID column, using physical provenance, we can annotate o with the IDs for CustSales(2) and CustSales(5).

Another way we can store provenance is by using *provenance predicates*. For output element o, we can retrieve the relevant input elements by applying the predicate (item-id='I1' $\wedge$ country='France') to CustSales: $\sigma_{\text{item-id='I1'} \wedge \text{country='France'}}(\text{CustSales}) = \{\text{CustSales}(2), \text{CustSales}(5)\}$. In contrast to physical provenance, provenance predicates can potentially reduce the space overhead of storing provenance, for example in a transformation that is many-one with large fan-in.

For transformation **JoinAgg**, note that the provenance predicates for all output elements would have the same form: Given any output element o, o's provenance can be found by applying the predicate (item-id=o.item-id $\wedge$ country=o.country) to CustSales. Since all predicates have a common form, it is unnecessary to annotate individual elements with predicates; we can instead use *logical provenance*—provenance information stored at the transformation level. An example of a logical-provenance specification that holds for this transformation is the *attribute mapping* (CS.item-id, CS.country) $\leftrightarrow$ (IC.item-id, IC.country), which captures the fact that the output subset of ItemCountryProfit (IC) with particular (item-id, country) values corresponds to the input subset in CustSales (CS) with the same (item-id, country) values. Logical provenance specifications can often capture exactly the same element-level provenance information as physical provenance or provenance predicates, but in a much more compact fashion, and without the overhead of capturing and storing IDs or predicates.

1.2.2 Provenance Capture

Having discussed some different ways to express provenance, we now describe at a high level how to capture provenance for the transformations in a workflow. As can be

seen in the running example, we do not limit ourselves to transformations expressible in relational algebra or SQL.

In our example, the **JoinAgg** and **Filter** transformations perform standard relational operations. For relational transformations, there is a great deal of past work that can be applied for capturing provenance automatically and efficiently (see [19] for a survey). Consider **JoinAgg**, for example, which is a *Select-Project-Join-Aggregate* (*SPJA*) transformation. As defined by past work, the provenance of element ItemCountryProfit(1) in Figure 1.2 output by **JoinAgg** contains the input elements CustSales(1) and ItemProfit(1). Intuitively, we can see that those are the elements that produced ItemCountryProfit(1).

Now consider **Extract**, a transformation that is not a standard relational operation. We cannot rely on the automatic methods for relational queries mentioned above to capture provenance for **Extract**. Either **Extract** must be instrumented to write out provenance information as it executes (presumably in the form of mappings between extracted records and their original input records), or it must provide some sort of procedure for computing provenance. In the worst case, without further information we have to assume that the provenance of each output element is the entire input data set.

1.2.3 Provenance Operations

Once provenance has been captured, our goal is to support the overall functions mentioned earlier: explanation, verification, and recomputation. We can support these functions using the following provenance-based operations:

- **Backward tracing.** Given an output data element e, where did e come from? That is, what data elements contributed to e? In our running example, we can use backward tracing to go from the output element LaptopProfit(3) to the element ItemData(3) in input set ItemData from which e was derived.
- **Forward tracing.** Given an input data element e, to what derived elements did e contribute? In our running example, we can use forward tracing to determine all of the profit calculations that were affected by ItemData(3).

- **Forward propagation.** If an input or derived data element e changes, propagate the change to everything it affects. Clearly this operation is related to forward tracing. In our running example, once we correct an element in ItemData, we can use forward propagation to recalculate only those profit calculations affected by the correction.
- **Refresh.** Given an output data element e, check if e is still valid. If it is not, refresh it to its new valid value. Clearly this operation is related to both backward tracing and forward propagation. In our running example, after updating the out-of-date data in ItemData, the analyst can refresh LaptopProfit(3) to its correct updated value without having to rerun the entire workflow.

1.3 Challenges and Contributions

The goal of this thesis is to support explanation, verification, and recomputation in data-oriented workflows using provenance. Here we outline the challenges encountered and our contributions.

1.3.1 Foundations

Defining provenance formally for general transformations and workflows can be challenging, as evidenced by a variety of definitions in the literature [19, 21, 22]. In Chapter 2 we lay the formal foundations for defining provenance in data-oriented workflows. Our contributions are the following:

- **Defining provenance** (Section 2.2). We give a new general definition of provenance, introducing the notions of *correctness*, *precision*, and *minimality*.
- **Theoretical properties of workflow provenance** (Section 2.3). 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.

1.3.2 Generalized Map and Reduce Workflows

A special case of data-oriented workflows is what we refer to as *generalized map and reduce workflows* (*GMRWs*), in which all transformations are either *map* or *reduce* functions [8, 24]. This setting is more general than conventional MapReduce jobs, which have just one map function followed by one reduce function; rather we consider any acyclic graph of map and reduce functions. In Chapter 3 we will see that GMRWs allow us to capture and exploit provenance more easily and efficiently than for general data-oriented workflows. We describe how provenance can be captured for both map and reduce functions transparently using wrappers in Hadoop [8]. Our contributions are the following:

- **Replay property of GMRW provenance** (Section 3.3). We identify a class of GMRW workflows in which provenance is guaranteed to satisfy the *replay property*, a useful guarantee when provenance is used for GMRW debugging. The replay property states that given a workflow instance and an element in its output data set, we can produce the output element by running its provenance through the workflow.
- **Provenance capture and tracing** (Section 3.4). We describe how provenance can be captured and stored during workflow execution in the GMRW setting, and we specify tracing procedures using provenance.

1.3.3 Provenance-Based Refresh

Consider a workflow in which the input data sets have been modified since the workflow was run, but the workflow has not been rerun on the modified input. In Chapter 4 we consider the problem of selectively refreshing one or more elements in the output data, i.e., compute the latest values of particular output elements based on the modified input data. By exploiting provenance, we can rerun just the relevant computation to refresh the output elements. Our contributions are the following:

- **Foundations for the refresh problem** (Section 4.2). We formalize the refresh problem.

- **Refresh algorithm** (Section 4.2). We specify a refresh procedure for a workflow setting in which each transformation has provenance captured in the form of provenance predicates.
- **Identification of favorable workflow properties** (Section 4.3). We identify specific properties of transformations, provenance, and workflows for which refresh can be very efficient, while it is inherently less efficient when the properties do not hold.

1.3.4 Logical Provenance

As mentioned earlier, logical provenance, i.e., provenance information stored at the transformation level, can often capture exactly the same element-level provenance information as physical provenance or provenance predicates, but in a much more compact fashion, and without the overhead of capturing and storing IDs or predicates. In Chapter 5 we describe how to support backward tracing using logical provenance. Our contributions are the following:

- **Logical provenance specifications for transformations** (Section 5.2). We describe a simple logical-provenance specification language consisting of *attribute mappings* and *filters*.
- **Algorithms for backward tracing** (Section 5.3). We provide algorithms for backward tracing in workflows where logical provenance for each transformation is specified using our language. In our algorithms we perform backward tracing at the schema level to the extent possible, although eventually accessing the data is required obviously.
- **Logical provenance for relational transformations** (Section 5.4). We consider logical provenance in the relational setting, showing that for a class of *Select-Project-Join* (SPJ) *transformations*, logical provenance specifications encode minimal provenance.

1.3.5 Panda System

In Chapter 6 we describe our prototype system called *Panda* (for *Provenance And Data*). Panda supports data-oriented workflows that capture provenance, and it supports all four provenance operations introduced in Section 1.2.3: backward tracing, forward tracing, forward propagation, and refresh. Panda allows arbitrary data-oriented workflows, with each transformation specified in either SQL or in Python. There have been multiple versions of the Panda system as we have developed our work. Our contributions are as follows:

- **Architecture description** (Section 6.1). We describe the high-level architecture of the final version of Panda, which supports debugging and drill-down using logical provenance.
- **Provenance capture and tracing** (Section 6.2). We describe how Panda generates logical provenance specifications and executes our tracing algorithms.
- **Logical provenance experiments** (Section 6.3). We present some performance results for logical provenance.
- **Provenance-based refresh** (Section 6.4). We describe an earlier version of Panda that was used to experiment with provenance-based refresh.
- **Experimental results for refresh** (Section 6.5). We report experimental results based on this earlier version that consider the overhead of provenance capture, and the crossover point between selective refresh and full workflow recomputation.

1.4 Related Work

There has been a large body of work in provenance over the past two decades. Surveys are presented in, e.g., [13, 19, 37]. Provenance models for relational and semistructured transformations are presented in, e.g., [9, 10, 16, 22, 25, 27, 32, 40]. Provenance specifically in the data-oriented workflow setting is considered by [5, 7, 11, 18, 21, 29, 33, 42], among others. Here we give an overview of related work applicable to the thesis in general. More specific and detailed comparisons are given within each chapter.

Provenance granularity. Provenance has two *granularities*:

1) *Schema-level* (*coarse-grained*)

2) *Instance-level* (*fine-grained*)

Schema-level provenance answers questions such as which data sets were used to produce a given output data set. Systems that focus on schema-level provenance (e.g., [11, 29]) are typically targeted for cases where the transformation graph is large and complex. In contrast, we consider instance-level provenance, which treats individual elements within a data set separately.

Transformation type and provenance queries. Another defining characteristic of related work in provenance is the type of transformations considered, ranging from copying operations [15] to *Select-Project-Join-Union* (*SPJU*) queries [22, 30, 36, 41] to arbitrary "black boxes" [21]. The transformations for which provenance is captured are closely related to the types of provenance queries that are typically performed. For example, by considering only SPJ transformations, reference [30] can provide *non-answer provenance*, explanations for why particular output elements are not present in a query result.

Eager vs. lazy. Provenance systems can be classified as either *eager* or *lazy*.

- Eager provenance systems (e.g., [10]) store all provenance information immediately after performing transformations, in preparation for provenance operations that will be asked later.
- In contrast, lazy provenance systems [21, 22] may store high-level provenance information, or perhaps no information at all, doing all the work on-demand when provenance operations are invoked.

In general, eager provenance has a higher overhead at capture time, but can also answer provenance queries more efficiently. One contribution of our work is to blur the distinction between eager and lazy provenance. For example, while we generate logical-provenance specifications eagerly (at workflow execution time), these specifications are used to generate instance-level provenance at tracing time.

Chapter 2

Foundations

This chapter lays foundations for the problem of defining provenance in data-oriented workflows. Section 2.1 defines data-oriented workflows. Section 2.2 gives a new general definition of provenance, introducing the notions of *correctness*, *precision*, and *minimality*. Section 2.3 describes when provenance properties such as correctness and minimality carry over from individual transformations to the workflow as a whole. Section 2.4 discusses related work, and Section 2.5 concludes the chapter.

2.1 Data-Oriented Workflows

Let a *data set* be any set of data *elements*. We are not concerned about the types of individual data elements; we treat them simply as members of a data set. A *transformation* T is any procedure that takes one or more data sets as input and produces one or more data sets as output. As we saw in the running example introduced in Chapter 1, we do not limit ourselves to transformations expressible in relational algebra or SQL. For any input data sets $I_1, \ldots, I_m$, we say that the application of T to $I_1, \ldots, I_m$ resulting in output sets $O_1, \ldots, O_r$, denoted $(O_1, \ldots, O_r) = T(I_1, \ldots, I_m)$, is an *instance* of T.

Data-oriented workflows are graphs where nodes denote data-set transformations, and edges denote the flow of data input to and output from the transformations: Input data sets I_1,...,I_m are fed into a graph of transformations T_1,...,T_n to produce

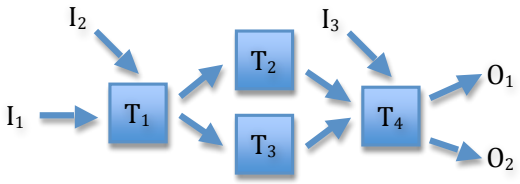

Figure 2.1: Example data-oriented workflow.

output data sets O_1,...,O_r. An example data-oriented workflow is shown in Figure 2.1. In this thesis we only consider workflows that are a directed acyclic graph, i.e., we do not consider cycles.

2.2 Provenance

For now, consider a single transformation T with a single input I and multiple outputs $O_1, \ldots, O_r$. Let $O = O_1 \cup \ldots \cup O_r$. We will generalize to multiple inputs in Section 2.2.4, and we will assemble transformations into workflows in Section 2.3. We assume $T(\emptyset) = \emptyset$ throughout the thesis, i.e., transformations never produce spurious output elements. 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 [19, 21, 22].

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 $I^* \subset I^{**}$. As we will see, it turns out with our definitions that there always exists a most precise provenance, which we refer to as the minimal provenance.

2.2.1 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 2.2.1 (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 $o \in T(I^*)$.

We also note that in our definition of correctness, we consider only subsets I' of the input instance I. That is, if $I' \not\subseteq I$, then we do not need to consider $T(I')$ to determine the correctness of provenance $I^* \subseteq I$. For most of this chapter, we assume T is defined over all subsets of I.

Example 2.2.1 We continue with the example workflow and data shown in Figures 1.1 and 1.2 of Section 1.1. Let $o = \mathsf{CustSales}(1)$, i.e., the first element from data set CustSales. 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 $\mathsf{CustData}(1)$ 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 2.2.1 is satisfied. □

2.2.2 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

$I^* \subseteq I$ and $I^{**} \subseteq I$ that are both correct provenance for an output element o, if $I^* \subset I^{**}$, we prefer the smaller subset I^*, since the smaller subset tells us more about what actually contributed to o.

Definition 2.2.2 (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 2.2.2 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 2.2.1. □

2.2.3 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 2.2.3 (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^*. □

In Section 2.2.4, we will give an example of minimal provenance after we have defined minimality for multi-input transformations. We now show that there always exists a unique minimal provenance.

Theorem 2.2.1 (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.

Proof. Since I is always correct provenance for o, there always exists at least some subset of I that is correct provenance for o. And since I is finite, we can take the

intersection of all subsets of I that are correct provenance for o. Thus I^M is well-defined.

We show that I^M is correct. To do so, we will show that the intersection of any two correct provenances is correct provenance, from which the correctness of I^M follows. Let $I^* \subseteq I$ and $I^{**} \subseteq I$ both be correct provenance for o. We show that $I^* \cap I^{**}$ is correct provenance for o. We need to show that for any $I' \subseteq I$, $o \in T(I')$ if and only if $o \in T(I' \cap (I^* \cap I^{**}))$. Since I^* is correct, we know that $o \in T(I')$ if and only if $o \in T(I' \cap I^*)$. Since I^{**} is correct, we know that $o \in T(I' \cap I^*)$ if and only if $o \in T((I' \cap I^*) \cap I^{**})) = T(I' \cap (I^* \cap I^{**}))$. Thus, $I^* \cap I^{**}$ is correct provenance for o.

We now show that I^M is minimal by contradiction. Suppose there existed correct provenance I^{**} for o such that $I^{**} \subset I^M$. By the definition of I^M, since I^{**} is correct provenance, we know that $I^{**} = I_j^*$ for some j and thus $I^M = I_1^* \cap \ldots \cap I_n^* \subseteq I_j^* = I^{**}$, a contradiction.

We now show that I^M is unique. Let I^{**} be any minimal provenance. Since I^{**} is correct, by the definition of I^M, we know that $I^{**} = I_j^*$ for some j, and thus $I^M = I_1^* \cap \ldots \cap I_n^* \subseteq I_j^* = I^{**}$, implying $I^M \subseteq I^{**}$. Since I^{**} is minimal, we know $I^M \not\subset I^{**}$. Thus, $I^{**} = I^M$. □

2.2.4 Multiple Inputs

The definitions for correctness, precision, and minimality generalize naturally to multiple input sets.

Definition 2.2.4 (Correctness) Let $O = T(I_1, \ldots, I_m)$ and let $o \in O$. Provenance $\langle I_1^*, \ldots, I_m^* \rangle$, $I_1^* \subseteq I_1, \ldots, I_m^* \subseteq I_m$, is *correct* for o with respect to T if:

- For any $\langle I_1', \ldots, I_m' \rangle$ such that $I_1' \subseteq I_1, \ldots, I_m' \subseteq I_m$, $o \in T(I_1', \ldots, I_m')$ if and only if $o \in T(I_1' \cap I_1^*, \ldots, I_m' \cap I_m^*)$. □

Definition 2.2.5 (Precision) Let $O = T(I_1, \ldots, I_m)$ and let $o \in O$. Suppose $\langle I_1^*, \ldots, I_m^* \rangle$ and $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ are both correct provenance for o with respect to T. $\langle I_1^*, \ldots, I_m^* \rangle$ is *at least as precise* as $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ if $I_1^* \subseteq I_1^{**}, \ldots, I_m^* \subseteq I_m^{**}$. $\langle I_1^*, \ldots, I_m^* \rangle$ is *more precise* than $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ if $I_1^* \subseteq I_1^{**}, \ldots, I_m^* \subseteq I_m^{**}$ and there exists some i for which $I_i^* \subset I_i^{**}$. □

Definition 2.2.6 (Minimality) Let $O = T(I_1, \ldots, I_m)$ and let $o \in O$. Suppose $\langle I_1^*, \ldots, I_m^* \rangle$ is correct provenance for o with respect to T. $\langle I_1^*, \ldots, I_m^* \rangle$ is *minimal* for o with respect to T if there does not exist provenance $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ that is correct and more precise than $\langle I_1^*, \ldots, I_m^* \rangle$. □

Theorem 2.2.2 (Unique Minimal Provenance) Let $O = T(I_1, \ldots, I_m)$ and let $o \in O$. Let $\langle I_1^{1*}, \ldots, I_m^{1*} \rangle, \ldots, \langle I_1^{n*}, \ldots, I_m^{n*} \rangle$ be all of the correct provenances for o with respect to T. Let $I^M = \langle (I_1^{1*} \cap \ldots \cap I_1^{n*}), \ldots, (I_m^{1*} \cap \ldots \cap I_m^{n*}) \rangle$. I^M is correct and minimal, and there is no other correct and minimal provenance for o with respect to T.

Proof. Since $\langle I_1, \ldots, I_m \rangle$ is always correct provenance for o, there always exists at least some correct provenance for o. And since $\langle I_1, \ldots, I_m \rangle$ is finite, we can take the intersection of all subsets of $\langle I_1, \ldots, I_m \rangle$ that are correct provenance for o. Thus I^M is well-defined.

We show that I^M is correct. To do so, we will show that the intersection of any two correct provenances is correct provenance, from which the correctness of I^M follows. Let $\langle I_1^*, \ldots, I_m^* \rangle$ and $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ both be correct provenance for o. We show that $\langle (I_1^* \cap I_1^{**}), \ldots, (I_m^* \cap I_m^{**}) \rangle$ is correct provenance for o. We need to show that for any $\langle I_1', \ldots, I_m' \rangle$ such that $I_1' \subseteq I_1, \ldots, I_m' \subseteq I_m$, $o \in T(I_1', \ldots, I_m')$ if and only if $o \in T((I_1' \cap (I_1^* \cap I_1^{**})), \ldots, (I' \cap (I_m^* \cap I_m^{**})))$. Since $\langle I_1^*, \ldots, I_m^* \rangle$ is correct, we know that $o \in T(I_1', \ldots, I_m')$ if and only if $o \in T((I_1' \cap I_1^*), \ldots, (I_m' \cap I_m^*))$. Since $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ is correct, we know that $o \in T((I_1' \cap I_1^*), \ldots, (I_m' \cap I_m^*))$ if and only if $o \in T(((I_1' \cap I_1^*) \cap I_1^{**}), \ldots, ((I_m' \cap I_m^*) \cap I_m^{**})) = T((I_1' \cap (I_1^* \cap I_1^{**})), \ldots, (I_m' \cap (I_m^* \cap I_m^{**})))$. Thus, $\langle (I_1^* \cap I_1^{**}), \ldots, (I_m^* \cap I_m^{**}) \rangle$ is correct provenance for o.

We now show that $I^M = \langle I_1^M, \ldots, I_m^M \rangle$ is minimal by contradiction. Suppose there existed correct provenance $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ for o such that $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ is more precise than I^M. By the definition of I^M, since $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ is correct provenance and $\langle I_1^M, \ldots, I_m^M \rangle = \langle (I_1^{1*} \cap \ldots \cap I_1^{n*}), \ldots, (I_m^{1*} \cap \ldots \cap I_m^{n*}) \rangle$, we know that $I_1^M \subseteq I_1^{**}, \ldots, I_m^M \subseteq I_m^{**}$, contradicting the assumption that $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ is more precise than $\langle I_1^M, \ldots, I_m^M \rangle$.

We now show that $I^M = \langle I_1^M, \ldots, I_m^M \rangle$ is unique. Let $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ be minimal provenance. Since $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ is correct provenance, by definition of I^M we know that $I_1^M \subseteq I_1^{**}, \ldots, I_m^M \subseteq I_m^{**}$. Since $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ is minimal, we know that $\langle I_1^M, \ldots, I_m^M \rangle$ cannot be more precise than $\langle I_1^{**}, \ldots, I_m^{**} \rangle$, implying that $I_1^M \not\subset I_1^{**}, \ldots, I_m^M \not\subset I_m^{**}$. Thus, $\langle I_1^{**}, \ldots, I_m^{**} \rangle = \langle I_1^M, \ldots, I_m^M \rangle$. □

We now give an example of minimal provenance.

Example 2.2.3 Consider $o = \mathsf{ItemCountryProfit}(4)$. Minimal provenance for o with respect to transformation **JoinAgg** is $\langle \mathsf{CS}^*, \mathsf{IP}^* \rangle$, where $\mathsf{CS}^* = \{\mathsf{CustSales}(2), \mathsf{CustSales}(5)\}$ and $\mathsf{IP}^* = \{\mathsf{ItemProfit}(3)\}$. Note that all three elements in $\langle \mathsf{CS}^*, \mathsf{IP}^* \rangle$ are needed for $\langle \mathsf{CS}^*, \mathsf{IP}^* \rangle$ to be correct provenance for o. □

2.3 Workflow Provenance

Next, 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.

Composition is associative, so we denote the linear composition of n transformations as $T_1 \circ T_2 \circ \cdots \circ T_n$. In Section 2.3.5, we extend our formalism to cover workflows where transformations may have multiple input and output data sets, allowing workflows to be arbitrary DAGs.

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 2.3.1 (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 o be any output element in I_{n+1}. For each data set I_i in the workflow instance, define $I_i^* \subseteq I_i$ recursively:

- $I_{n+1}^* = \{o\}$
- For $i \leq n$, $I_i^* = \bigcup_{e \in I_{i+1}^*} P_{T_i}(e)$

The *workflow provenance* of o in W, denoted $P_W(o)$, is input subset $I_1^* \subseteq I_1$. □

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

In our running example of Section 1.1, all of the transformations except **JoinAgg** (technically requiring the extended formalism in Section 2.3.5) are monotonic.

Definition 2.3.3 (One-Many Transformations) A transformation T is *one-many* if for any input set I, each output element $o \in T(I)$ has exactly one input element in its minimal provenance. □

In our running example, transformations **Extract**, **CalcProfit**, and **Filter** are one-many.

Definition 2.3.4 (Many-One Transformations) A transformation T is *many-one* if for any input set I, each input element $e \in I$ is in at most one output element's minimal provenance. □

In our running example, transformations **CalcProfit** and **Filter** are many-one.

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 2.3.5 (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. Also note that for any transformation instance $T(I)$, if provenance $I^* \subseteq I$ is correct, then I^* is weakly correct. Provenance $I^* = I$ is always trivially weakly correct for any $o \in T(I)$.

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.

2.3.1 Remainder of Section

We now outline the major results of this section. 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 each of the transformations T_i includes a correct provenance specification.

- **Preservation of correctness** (Section 2.3.2). If all transformations T_i are monotonic, then W's workflow provenance is correct. However, if there exists any non-monotonic transformation in W, then W's workflow provenance is not necessarily correct.
- **Preservation of minimality** (Section 2.3.3). If all transformations T_i are monotonic, we have minimal provenance at each transformation, $T_1, \ldots, T_j$ are many-one, and $T_{j+1}, \ldots, T_n$ are one-many, then W's workflow provenance is minimal. However, if all transformations T_i are monotonic and we have minimal provenance at each transformation (but we don't have the many-one and one-many properties of the previous statement), then W's provenance is not necessarily minimal.

- **Preservation of weak correctness** (Section 2.3.4). If there exists at most one nonmonotonic transformation in W, then W's workflow provenance is weakly correct. However, if there is more than one nonmonotonic transformation in W, then W's workflow provenance is not necessarily weakly correct.

2.3.2 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 2.3.1) is also correct.

Theorem 2.3.1 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.

Proof. We prove the result by induction on n. The base case of $n = 1$ follows from the assumption. Now suppose the theorem holds for $n = k$ and consider the theorem for $n = k + 1$. Consider $o \in I_{k+2}$. $P_W(o) = \bigcup_{e \in P_{T_{k+1}}(o)} P_W(e)$.

We prove by contradiction. Suppose $I_1^* = P_W(o)$ is not correct for o with respect to $W_{k+1} = T_1 \circ T_2 \circ \ldots \circ T_{k+1}$. Then there exists some $I_1' \subseteq I_1$ such that $W_{k+1}(I_1') \cap \{o\} \neq W_{k+1}(I_1' \cap I_1^*) \cap \{o\}$. Since each transformation in W_{k+1} is monotonic, W_{k+1} is monotonic. It then follows from W_{k+1}'s monotonicity and the above inequality that $o \in W_{k+1}(I_1')$ and $o \notin W_{k+1}(I_1' \cap I_1^*)$.

For any $e \in I_{k+1}^* = P_{T_{k+1}}(o)$, by the inductive hypothesis we know that $P_W(e)$ is correct for e with respect to $W_k = T_1 \circ T_2 \circ \ldots \circ T_k$. Since $P_W(e) \subseteq I_1^*$ for any $e \in I_{k+1}^*$, we know that $e \in W_k(I_1')$ if and only if $e \in W_k(I_1' \cap I_1^*)$, from which it follows that $W_k(I_1') \cap I_{k+1}^* = W_k(I_1' \cap I_1^*) \cap I_{k+1}^*$.

Recall that $o \in W_{k+1}(I_1') = T_{k+1}(W_k(I_1'))$. Since I_{k+1}^* is correct for o with respect to T_{k+1}, we then know that $o \in T_{k+1}(W_k(I_1') \cap I_{k+1}^*) = T_{k+1}(W_k(I_1' \cap I_1^*) \cap I_{k+1}^*)$. Since T_{k+1} is monotonic, it follows that $o \in T_{k+1}(W_k(I_1' \cap I_1^*)) = W_{k+1}(I_1' \cap I_1^*)$, a contradiction. Thus, I_1^* is correct. $\square$

Figure 2.2: Counterexample for correct workflow provenance.

If we drop the assumption that all transformations are monotonic, we find an example demonstrating that workflow provenance is not necessarily correct.

Example 2.3.1 Consider workflow W = **SingleStore** $\circ$ **Projection** shown in Figure 2.2. 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): SSC = **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: **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 2.3.1 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. □

2.3.3 Preservation of Minimality

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

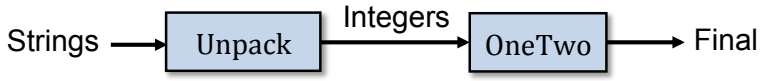

Figure 2.3: Counterexample for minimal workflow provenance.

Example 2.3.2 Consider workflow $W = \textbf{Unpack} \circ \textbf{OneTwo}$ shown in Figure 2.3. 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 Integers $= \{1, 2\}$. We have minimal provenance for **Unpack**: $P_{\textbf{Unpack}}(1) = \{$"1", "1, 2"$\}$ and $P_{\textbf{Unpack}}(2) = \{$"1, 2"$\}$. Transformation **OneTwo** processes intermediate data set Integers as follows:

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

In this workflow instance, Final = **OneTwo**(Integers) $= \{$"success"$\}$. The minimal provenance of "success" with respect to **OneTwo** is $P_{\textbf{OneTwo}}($"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 2.3.1 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 many-one transformations followed by a set of one-many transformations, and we have minimal provenance for each transformation in the workflow, then workflow provenance is minimal.

Theorem 2.3.2 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 many-one transformations, and $T_{j+1}, \ldots, T_n$ are one-many 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. □

The proof of this theorem involves some lemmas. Before the lemmas, we define the notion of *minimal witnesses* from [16].

Definition 2.3.6 (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 difference 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 lemma 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.

Lemma 2.3.1 Let $O = T(I)$ be an instance of a monotonic transformation, and let $o \in O$. Let $I^w_1, \ldots, I^w_m$ 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 $I^* = I^w_1 \cup \ldots \cup I^w_m$.

Proof of Lemma 2.3.1 We first show that $I^* \supseteq I^w_1 \cup \ldots \cup I^w_m$. Given any element $e \in I^w_1 \cup \ldots \cup I^w_m$, we show $e \in I^*$ by contradiction. Suppose $e \notin I^*$. Since $e \in I^w_1 \cup \ldots \cup I^w_m$, e is in some I^w_j. Since I^w_j is a minimal witness of o, I^* is correct for o, and T is monotonic, we know that $\{o\} = (T(I^w_j) \cap \{o\}) = (T(I^w_j \cap I^*) \cap \{o\}) \subseteq (T(I^w_j - \{e\}) \cap \{o\})$. Since I^w_j is a minimal witness, $o \notin T(I^w_j - \{e\})$, thus $(T(I^w_j - \{e\}) \cap \{o\}) = \emptyset$, a contradiction. Thus, $e \in I^*$, implying that $I^* \supseteq I^w_1 \cup \ldots \cup I^w_m$.

Now we show that $I^* \subseteq I^w_1 \cup \ldots \cup I^w_m$. Given any element $e \in I^*$, we show $e \in I^w_1 \cup \ldots \cup I^w_m$. Since e is in the minimal provenance I^* of o, we know that $I^* - \{e\}$ is not correct provenance, implying that there exists some subset $I' \subseteq I$ such that $o \in T(I')$ but $o \notin T(I' \cap (I^* - \{e\})) = T(I' \cap I^* - \{e\})$. Since I^* is correct, we know

that $o \in T(I' \cap I^*)$. Consider all subsets of $I' \cap I^*$. We know at least one subset, namely the entire $I' \cap I^*$, produces o. Take the smallest subset I'' (pick arbitrarily if there is a tie) that produces o. It must contain e because T is monotonic and $o \notin T(I' \cap I^* - \{e\})$. Also, no subset of I'' can produce o, since we picked the minimal subset. Thus, I'' is equal to some minimal witness I_j^w, from which it follows that $e \in I_1^w \cup \ldots \cup I_m^w$. We have shown that $I^* \subseteq I_1^w \cup \ldots \cup I_m^w$. Together with $I^* \supseteq I_1^w \cup \ldots \cup I_m^w$, it follows that $I^* = I_1^w \cup \ldots \cup I_m^w$. □

Lemma 2.3.2 Let $W = T_1 \circ \ldots \circ T_n$ where each T_i is a monotonic many-one transformation. Then W is also a many-one transformation.

Proof of Lemma 2.3.2 We prove by induction on n. For $n = 1$, the lemma follows from the assumption. Now suppose the lemma holds for $n = k$ and consider the lemma for $n = k + 1$. Since the lemma holds for $n = k$, we know $W_k = T_1 \circ \ldots \circ T_k$ is many-one.

For any element $o \in I_{k+2}$, let $P_{T_{k+1}}(o) \subseteq I_{k+1}$ be the minimal provenance of o with respect to T_{k+1}. For any element $e_{k+1} \in I_{k+1}$, let $P_{W_k}(e_{k+1}) \subseteq I_1$ be the minimal provenance of e_{k+1} with respect to W_k. Let $P_{W_{k+1}}(o) = \cup_{e \in P_{T_{k+1}}(o)} P_{W_k}(e)$ be the workflow provenance of o in $W_{k+1} = W_k \circ T_{k+1}$. Then by Theorem 2.3.1, since we have minimal (and thus correct) provenance for W_k and T_{k+1}, $P_{W_{k+1}}(o)$ is correct for o with respect to W_{k+1}.

Given any $e_1 \in I_1$, since e_1 is in the minimal provenance of at most one element $e_{k+1} \in I_{k+1}$, and e_{k+1} is in the minimal provenance of at most one element $o \in I_{k+2}$, we know that e_1 is in $P_{W_{k+1}}(o)$ for at most one element $o \in I_{k+2}$. Let $P(o) \subseteq I_1$ be the minimal provenance of o with respect to W_{k+1}. Since $P_{W_{k+1}}(o)$ is correct, $P_{W_{k+1}}(o) \supseteq P(o)$ for all $o \in I_{k+2}$. Thus, e_1 is in the minimal provenance $P(o)$ for at most one element $o \in I_{k+2}$, i.e., W_{k+1} is many-one. □

Lemma 2.3.3 Let $W = T_1 \circ \ldots \circ T_n$. Consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$ where each T_i is a monotonic many-one transformation. For each transformation T_i and element $e \in I_{i+1}$, suppose $P_{T_i}(e)$ is the minimal provenance of e at transformation T_i. Consider any output element $o \in I_{n+1}$. Workflow provenance $P_W(o)$ is minimal for o with respect to W.

Proof of Lemma 2.3.3 We prove by induction on n. For $n = 1$ the lemma follows from the assumption. Now suppose the lemma holds for $n = k$ and consider the lemma for $n = k + 1$. Since the lemma holds for $n = k$, for all elements $e \in I_{k+1}, P_W(e)$ is minimal for e with respect to $W_k = T_1 \circ \ldots \circ T_k$.

Consider $o \in I_{k+2}$. $P_W(o) = \bigcup_{e \in P_{T_{k+1}}(o)} P_W(e)$. To show workflow provenance $I_1^* = P_W(o)$ is minimal, we need to show that for all elements $e_1 \in I_1^*$, $I_1^* - \{e_1\}$ is not correct provenance for o with respect to W. To show that $I_1^* - \{e_1\}$ is not correct, we will construct an input subset $I_1' \subseteq I_1^* \subseteq I_1$ such that $o \in W(I_1')$ and $o \notin W(I_1' - \{e_1\})$.

Let e_1 be any element in I_1^*. $I_{k+1}^* = P_{T_{k+1}}(o)$ is the minimal provenance of o with respect to T_{k+1}. Since $I_1^* = \bigcup_{e \in I_{k+1}^*} P_W(e)$, we know that $e_1 \in P_W(e_{k+1})$ for some $e_{k+1} \in I_{k+1}^*$. By Lemma 2.3.1, there exists a minimal witness $I_{k+1}^w \subseteq I_{k+1}^*$ of o with respect to T_{k+1} that contains e_{k+1}. Consider $P_W(e_{k+1}) \subseteq I_1$, the workflow provenance of $e_{k+1} \in I_{k+1}^w$ in W. By the inductive hypothesis, $P_W(e_{k+1})$ is minimal for e_{k+1} with respect to $W_k = T_1 \circ \ldots \circ T_k$. By Lemma 2.3.1, since $P_W(e_{k+1})$ is minimal for e_{k+1} and $e_1 \in P_W(e_{k+1})$, there exists a minimal witness $I_1^w \subseteq P_W(e_{k+1}) \subseteq I_1$ for e_{k+1} with respect to W_{k+1} that contains e_1.

Let $P(e)$ be the minimal provenance of $e \in I_{k+1}$ with respect to W_k. We construct subset $I_1' \subseteq I_1$ in the following way: $I_1' = \bigcup_{e \in I_{k+1}^w} D(e)$ where $D(e) = P(e)$ if $e \neq e_{k+1}$, and $D(e_{k+1}) = I_1^w \subseteq P(e_{k+1})$.

Since I_1' contains a minimal witness for every element $e \in I_{k+1}^w$ with respect to W_k, we know that $W_k(I_1') \supseteq I_{k+1}^w$, implying that $W(I_1') = T_{k+1}(W_k(I_1')) \supseteq T_{k+1}(I_{k+1}^w)$ since T_{k+1} is monotonic. $T_{k+1}(I_{k+1}^w) \supseteq \{o\}$, from which it follows that $o \in W(I_1')$.

Consider $W(I_1' - \{e_1\}) = T_{k+1}(W_k(I_1' - \{e_1\}))$. For all $e \in (I_{k+1}^w - \{e_{k+1}\})$, since W_k is many-one (by Lemma 2.3.2), and $D(e)$ is minimal (and thus correct) for e with respect to W_k, we know that $(W_k(I_1' - \{e_1\}) \cap \{e\}) = (W_k((I_1' - \{e_1\}) \cap D(e)) \cap \{e\}) = (W_k(D(e)) \cap \{e\}) = \{e\}$, implying that $e \in (W_k(I_1' - \{e_1\})$. For $e \notin I_{k+1}^w$, $(W_k(I_1' - \{e_1\}) \cap \{e\}) = (W_k((I_1' - \{e_1\}) \cap P(e)) \cap \{e\}) = W_k(\emptyset) \cap \{e\} = \emptyset$, implying that $e \notin W_k(I_1' - \{e_1\})$. For element e_{k+1}, since W_k is many-one, and $P(e_{k+1})$ is minimal (and thus correct) for e_{k+1} with respect to W_k, $(W_k(I_1' - \{e_1\}) \cap \{e_{k+1}\}) = W_k((I_1' - \{e_1\}) \cap P(e_{k+1})) \cap \{e_{k+1}\} = W_k(I_1^w - \{e_1\}) \cap \{e\} = \emptyset$, implying that $e_{k+1} \notin$

$W_k(I'_1 - \{e_1\})$. Thus, $W_k(I'_1 - \{e_1\}) = I^w_{k+1} - \{e_{k+1}\}$. It follows that $W(I'_1 - \{e\}) = T_{k+1}(I^w_{k+1} - \{e_{k+1}\}) = \emptyset$, proving that $I^*_1 - \{e_1\}$ is not correct provenance for o with respect to W, and thus I^*_1 is minimal for o with respect to W_{k+1}. □

Proof of Theorem 2.3.2 For each data set I_i in the workflow instance, define $I^*_i \subseteq I_i$ recursively:

- $I^*_{n+1} = \{o\}$
- For $i \leq n$, $I^*_i = \bigcup_{e \in I^*_{i+1}} P_{T_i}(e)$

It is straightforward to see that $I^*_1 = P_W(o) \subseteq I_1$. For any one-many transformation T, the minimal provenance $P_T(e)$ of any element e in T's output has size 1. Since $T_{j+1}, \ldots, T_n$ are one-many, we know that $|I^*_{j+1}| = 1$. Let $I^*_{j+1} = \{e_{j+1}\}$.

By Lemma 2.3.3, $P_W(e_{j+1}) = I^*_1$ is minimal for e_{j+1} with respect to $W_j = T_1 \circ \ldots \circ T_j$. To show that workflow provenance I^*_1 is minimal for o with respect to W, let e_1 be any element in I^*_1. We need to show that $I^*_1 - \{e_1\}$ is not correct provenance for o with respect to W. We construct an input subset $I'_1 \subseteq I^*_1 \subseteq I_1$ such that $o \in W(I'_1)$ and $o \notin W(I'_1 - \{e_1\})$. Since $I^*_1 = P_W(e_{j+1})$ is minimal provenance for e_{j+1} with respect to W_j, by Lemma 2.3.1 there exists a minimal witness $I^w \subseteq I^*_1$ of e_{j+1} with respect to W_j containing e_1.

Let $I'_1 = I^w$. Then $e_{j+1} \notin (T_1 \circ \ldots \circ T_j)(I'_1 - \{e_1\})$, implying that $o \notin W(I'_1 - \{e_1\})$. However, $e_{j+1} \in (T_1 \circ \ldots \circ T_j)(I'_1)$, implying that $o \in W(I'_1)$. Thus, workflow provenance is minimal. □

2.3.4 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 2.3.3 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.

Proof. For each data set I_i in the workflow instance, define $I_i^* \subseteq I_i$ recursively:

- $I_{n+1}^* = \{o\}$
- For $i \leq n$, $I_i^* = \bigcup_{e \in I_{i+1}^*} P_{T_i}(e)$

It is straightforward to see that $I_1^* = P_W(o) \subseteq I_1$. Given an input subset $I_1' \subseteq I_1$ define sets F_i as:

- $F_1 = I_1'$
- For $2 \leq i \leq n+1$, $F_i = T_{i-1}(F_{i-1})$

Note that $F_{n+1} = W(I_1')$. Recall from Definition 2.3.5 that weak correctness requires $o \in T(I')$ for every $I^* \subseteq I' \subseteq I$. If every transformation T_i is monotonic and $I_1' \supseteq I_1^*$, then $F_i \supseteq I_i^*$ for all i. It follows that $W(I_1') = F_{n+1} \supseteq I_{n+1}^* = \{o\}$, implying $o \in W(I_1')$ for any input subset I_1' such that $I_1^* \subseteq I_1' \subseteq I_1$, and thus workflow provenance I_1^* is weakly correct.

Suppose there is one nonmonotonic transformation T_j. Based on the above argument, since the workflow before T_j is monotonic, $I_j^* \subseteq F_j \subseteq I_j$. For each element $e \in I_{j+1}^*$, since $P_{T_j}(e) \subseteq I_j^*$, we know that $P_{T_j}(e) \subseteq F_j \subseteq I_j$, and since $P_{T_j}(e)$ is weakly correct, we know that $e \in T_j(F_j) = F_{j+1}$. Thus $F_{j+1} \supseteq I_{j+1}^*$. Since the transformations after T_j are all monotonic and $F_{j+1} \supseteq I_{j+1}^*$, we know that $F_i \supseteq I_i^*$ for all $i > j+1$. Thus, $W(I_1') = F_{n+1} \supseteq I_{n+1}^* = \{o\}$ for any input subset I_1' such that $I_1^* \subseteq I_1' \subseteq I_1$. Hence workflow provenance I_1^* is weakly correct. □

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

Example 2.3.3 Consider workflow $W =$ **OneTwo** $\circ$ **ThreeFour** shown in Figure 2.4. 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 **OneTwo**($\mathsf{Initial}$) $= \{3\}$
- Else if $1 \in \mathsf{Initial}$ and $2 \notin \mathsf{Initial}$, then **OneTwo**($\mathsf{Initial}$) $= \{3, 4\}$

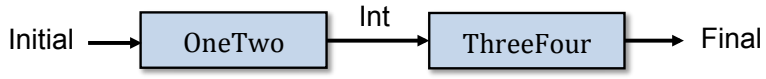

Figure 2.4: Counterexample for weakly correct workflow provenance.

- Else, **OneTwo**(Initial) = $\emptyset$

In this workflow instance, 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 $3 \in$ Int and $4 \notin$ Int, then **ThreeFour**(Int) = $\{5\}$
- Else, **ThreeFour**(Int) = $\emptyset$

In this workflow instance, 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 $o = 5 \in$ 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. □

2.3.5 Workflows with Multi-Input and Multi-Output Transformations

We now give a generalized definition of workflow provenance that supports workflows containing multi-input and multi-output transformations, i.e., the workflow is a directed graph, but without cycles. It is cumbersome to formally define composition for general DAGs as we did for linear workflows in Section 2.3 (i.e., $T_1 \circ T_2 \circ \ldots \circ T_n$ does not generalize comfortably), but the meaning of composition is straightforward.

Definition 2.3.7 (Workflow Provenance) Consider a workflow instance $W(I_1, \ldots, I_m)$ with initial inputs $I_1, \ldots, I_m$. 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 m-tuple $(I_1^*, \ldots, I_m^*)$, where $I_1^* \subseteq I_1, \ldots, I_m^* \subseteq I_m$. If

e is an initial input element, i.e., $e \in I_k$, then $P_W(e) = (\emptyset, \ldots, \{e\}, \ldots, \emptyset)$, a tuple in which every value except the k-th is empty. Otherwise, let e be an element in one of the output sets of a transformation T. Let $P_T(e) = (P_1^T(e), \ldots, P_n^T(e))$ be the one-level provenance of e with respect to T. Then $P_W(e) = (P_1^W(e), \ldots, P_m^W(e))$ where each $P_i^W(e) = \bigcup_{e' \in (P_1^T(e) \cup \ldots \cup P_n^T(e))} P_i^W(e')$. □

In this more general setting, we again want to determine when the properties of correctness, minimality, and weak correctness carry over from the individual transformations' provenance to the workflow provenance. We first generalize the definitions of *monotonicity* and *weak correctness*.

Definition 2.3.8 (Monotonicity) A transformation T is *monotonic* if:

- For any input sets $I_1, \ldots, I_m$ and $I'_1, \ldots, I'_m$ with $I_i \subseteq I'_i$, let $(O_1, \ldots, O_r) = T(I_1, \ldots, I_m)$ and $(O'_1, \ldots, O'_r) = T(I'_1, \ldots, I'_m)$. $O_1 \subseteq O'_1, \ldots, O_r \subseteq O'_r$. □

Definition 2.3.9 (Weak Correctness Definition) Let $(O_1, \ldots, O_r) = T(I_1, \ldots, I_m)$ be a transformation instance and let $o \in O_i$ for some i. We say that provenance $\langle I_1^*, \ldots, I_m^* \rangle$, $I_1^* \subseteq I_1, \ldots, I_m^* \subseteq I_m$, is *weakly correct* for o with respect to T if:

- For any $\langle I'_1, \ldots, I'_m \rangle$ such that $I_1^* \subseteq I'_1 \subseteq I_1, \ldots, I_m^* \subseteq I'_m \subseteq I_m$, $o \in T(I'_1, \ldots, I'_m)$. □

We now generalize the major theorems presented in this section. Proofs are omitted since they are long and notation-heavy, yet directly analogous to the proofs given earlier for linear workflows; no new complexities are encountered.

Theorem 2.3.4 (Preservation of Correctness) Let W be a workflow containing monotonic transformations $T_1, \ldots, T_n$. Consider a workflow instance $W(I_1, \ldots, I_m)$ with initial inputs $I_1, \ldots, I_m$. For each T_i, and each element e in any of T_i's output sets, let $P_{T_i}(e)$ be correct provenance for e with respect to T_i. Consider any output element o. Workflow provenance $P_W(o)$ is correct for o with respect to W. □

Theorem 2.3.5 (Preservation of Minimality) Let W be a workflow containing monotonic transformations $T_1, \ldots, T_n$, in which each transformation is either many-one or one-many, and along no path in the workflow is there a one-many transformation followed by an many-one transformation. Consider a workflow instance $W(I_1, \ldots, I_m)$ with initial inputs $I_1, \ldots, I_m$. For each T_i, and each element e in any of T_i's output sets, let $P_{T_i}(e)$ be minimal provenance for e with respect to T_i. Consider any output element o. Workflow provenance $P_W(o)$ is minimal for o with respect to W. □

Theorem 2.3.6 (Preservation of Weak Correctness) Let W be a workflow containing transformations $T_1, \ldots, T_n$, in which each path contains at most one non-monotonic transformation. Consider a workflow instance $W(I_1, \ldots, I_m)$ with initial inputs $I_1, \ldots, I_m$. For each T_i, and each element e in any of T_i's output sets, let $P_{T_i}(e)$ be weakly correct provenance for e with respect to T_i. Consider any output element o. Workflow provenance $P_W(o)$ is weakly correct for o with respect to W. □

Note that the workflow counterexamples (Examples 2.3.1, 2.3.2, 2.3.3) given earlier are also valid in this more general setting.

2.4 Related Work

Reference [16] 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 2.3.3). In contrast to [22], our work defines notions of correctness and minimality for provenance that apply to general transformations, not just relational transformations. Reference [32] 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 [32] are restricted to conjunctive

queries. Reference [10] considers the provenance of individual attribute values (*where-provenance*), while our work focuses on the provenance of data elements.

Reference [44] describes tracking provenance for arbitrary functions by instrumenting the program binary that implements the function. Since the approach in [44] 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 [21] 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 efficient. Provenance in [21] is defined separately for each transformation type, while the definitions for provenance given in this chapter are unified across all transformations. Also, while [21] 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.

2.5 Conclusions

We considered the problem of defining provenance in data-oriented workflows. We gave a new general definition of provenance, introducing the notions of correctness, precision, and minimality. After we defined workflow provenance in the intuitive recursive way, we then discussed its theoretical properties, identifying when the provenance properties of correctness, minimality, and weak correctness carry over from individual transformations to the workflow as a whole.

Chapter 3

Generalized Map and Reduce Workflows

3.1 Introduction

A special case of data-oriented workflows is what we refer to as *generalized map and reduce workflows* (*GMRWs*), in which all transformations are either *map* or *reduce* functions [8, 24]. Our setting is more general than conventional MapReduce jobs, which have just one map function followed by one reduce function; rather we consider any acyclic graph of map and reduce functions.

In this chapter, we explore data provenance for forward and backward tracing in GMRWs. In particular, we will see that the special case of workflows where all transformations are map or reduce functions allows us to capture and exploit provenance more easily and efficiently than for general data-oriented workflows. We will also see that provenance can be captured for both map and reduce functions transparently using wrappers in Hadoop [8], a popular open-source implementation of the MapReduce framework.

Although map and reduce functions as data transformations have become increasingly popular, we are unaware of any work that focuses specifically on provenance for GMRWs. Here we explore the overhead of provenance capture and the cost of

provenance tracing. Our goal is to enable efficient provenance tracing in GMRWs while keeping the capture overhead low. Overall, our contributions are as follows:

- After establishing foundations specific to this chapter in Section 3.2, in Section 3.3 we show how correct provenance is specified naturally for individual map and reduce functions. We then identify properties that hold for workflow provenance in a GMRW.
- Section 3.4 describes how correct provenance can be captured and stored during workflow execution, and it specifies backward and forward tracing procedures using the captured provenance.
- We have built a system called *RAMP* (*Reduce And Map Provenance*) implementing the concepts in this chapter. In Section 3.5, we report some performance results using RAMP on the time and space overhead of capturing provenance, and on the cost of provenance tracing.
- Sections 3.2-3.5 apply to arbitrary GMRWs. However, optimizations may be applied for transformations and workflows with certain properties, discussed in Section 3.6.

In Section 3.7 we discuss related work, and we conclude in Section 3.8.

3.1.1 Running Example

As a simplified example GMRW that serves primarily to illustrate our definitions and techniques, consider the workflow shown in Figure 3.1, used to gauge public opinion on movies. The inputs to the workflow are data sets Tweets and Diggs, containing user postings collected from Twitter and Digg, respectively. (Note we consider batch processing of data sets, not continuous stream processing.) The workflow involves the following transformations:

- Map functions **TweetScan** and **DiggScan** analyze the postings in data sets Tweets and Diggs, looking for postings that contain a single movie title and one or more positive or negative adjectives. For each such posting, a key-value pair is emitted to TwitterMovies (TM) or DiggMovies (DM), where the key is the title of

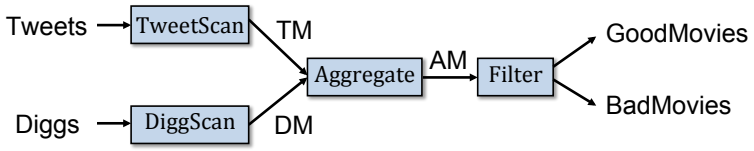

Figure 3.1: Movie sentiment workflow example.

the movie, and the value is a rating between 1 and 10 based on the combination of adjectives appearing.

- Reduce function **Aggregate** computes the number of ratings and the median rating for each movie title, producing data set AggMovies (AM).
- Map function **Filter** copies to GoodMovies those movies with at least 1000 ratings and a median rating of 6 or higher, and copies to BadMovies those movies with at least 1000 ratings and a median rating of 5 or lower.

As a simple example of how provenance might be useful in this workflow, suppose we are surprised to see that *Twilight* is in GoodMovies. Tracing provenance back one level to AggMovies, we see that *Twilight* has a median rating of 9, with over 1000 ratings. Further tracing provenance all the way back to the original postings, we sample usernames of *Twilight* fans. By reading other postings by these fans, we infer that teenage girls in particular have been flooding social media sites with raves for *Twilight*.

3.2 GMRW Transformations

Recall our foundational definitions for data-oriented workflows introduced in Chapter 2. In *generalized map and reduce workflows*, we follow the same formalism, but we specialize to two types of transformations: *map functions* and *reduce functions*. Section 3.2.1 formalizes map and reduce functions with just one input set and one output set. Section 3.2.2 then introduces two more transformation types, *union* and *split*, which are needed to handle map and reduce functions with multiple input and output sets.

3.2.1 Map and Reduce Functions

Map Functions. As in the MapReduce framework, a *map function* M produces zero or more output elements independently for each element in its input set I: $M(I) = \bigcup_{i \in I} M(\{i\})$. In practice, programmers in the MapReduce framework are not prevented from writing map functions that buffer the input or otherwise use "side-effect" temporary storage, resulting in behavior that violates this pure definition of a map function. In our work, we assume pure map functions.

Reduce Functions. A *reduce function* R takes an input data set I in which each element is a key-value pair, and returns zero or more output elements independently for each group of elements in I with the same key: Let $k_1, \ldots, k_n$ be all of the distinct keys in I. Then $R(I) = \bigcup_{1 \leq j \leq n} R(G_j)$, where each G_j consists of all key-value pairs in I with key k_j. Similar to map functions, we consider only pure reduce functions, i.e., those satisfying this definition. In the remainder of the chapter, we use $G_1, .., G_n$ to denote the key-based *groups* of a reduce function's input set I.

Transformation Properties

We now list some properties for GMRW transformations that are relevant for provenance.

Deterministic Functions. We assume that all functions are *deterministic*: Each map and reduce function returns the same output set when given the same input set. Again, programmers in the MapReduce framework are not prevented from creating nondeterministic functions, but we assume determinism in our work.

Multiplicity for Map Functions. We say that a map function M is *one-one* if for any input set I, each element in I produces at most one output element: For all $i \in I$, $|M(\{i\})| \leq 1$. Otherwise, the map function is *one-many*. (Note all map functions satisfy the definition of one-many transformations given earlier in Definition 2.3.3, assuming each output element is assigned a unique ID. By assigning unique IDs, we can assume that the output set contains no duplicates.) In our running example, **TweetScan**, **DiggScan**, and **Filter** are all one-one.

Multiplicity for Reduce Functions. We say that a reduce function R is *many-one* if for any input set I, each key-based group G_j of I returns at most one output element: $|R(G_j)| \leq 1$. Otherwise, the reduce function is *many-many*. (Note that in the context of GMRWs, we define *many-one* reduce functions based on the key-based groups. Thus our usage of the term *many-one* in this chapter is somewhat specialized over the term as introduced for general transformations in Definition 2.3.4.) In our running example, **Aggregate** is many-one.

Monotonicity. Recall that a transformation T is *monotonic* if for any input sets I and I', if $I \subseteq I'$, then $T(I) \subseteq T(I')$ (Definition 2.3.2). Note that map functions are always monotonic, but some reduce functions are nonmonotonic. In our running example, **Aggregate** is nonmonotonic. An example of a monotonic reduce function is one that simply returns the key for all groups above a certain size.

3.2.2 Union and Split Transformations

So far we have assumed map and reduce functions have a single input data set and single output data set. In practice, functions in the MapReduce framework can have multiple input data sets, but logically they union their input sets and then perform the function. Similarly, a map or reduce function with multiple output sets is logically equivalent to a function that outputs one large set, then splits it into multiple separate output sets. For our formal analysis in the next section, it is preferable to model all map and reduce functions as single-input and single-output. Thus, we logically add union and split transformations to GMRWs, without changing workflow behavior.

A *union* transformation takes input data sets $I_1, \ldots, I_m$ and creates output set $O = I_1 \cup \ldots \cup I_m$. A *split* transformation takes input set I and creates output sets $O_1, .., O_r$, with $O_1 \cup \ldots \cup O_r = I$, and $O_i \cap O_j = \emptyset$ for $i \neq j$. For split, we assume that output sets are both deterministic and context-independent, i.e., each $i \in I$ is in the same O_k regardless of other elements in I.

We assume all of our data sets have unique identifiers for all elements. We further assume identifiers are made globally unique, so $\cup$ in the above definitions is always

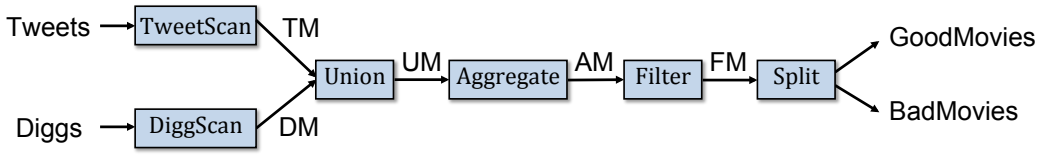

Figure 3.2: Movie workflow example with union and split.

disjoint union. Figure 3.2 adds union and split transformations to our running example.

3.3 Provenance in GMRWs

For each transformation type in GMRWs, there is a natural way to specify correct provenance:

- **Map Provenance.** Given a map function M, the provenance of an output element $o \in M(I)$ is the input element i that produced o, i.e., $o \in M(\{i\})$.
- **Reduce Provenance.** Given a reduce function R, the provenance of an output element $o \in R(I)$ is the group $G_j \subseteq I$ that produced o, i.e., $o \in R(G_j)$.
- **Union Provenance.** Given a union transformation U, the provenance of an output element $o \in U(I_1, \ldots, I_m) = I_1 \cup \ldots \cup I_m$ is the corresponding input element i in some I_k, where $i = o$. (Recall from Section 3.2.2 that $\cup$ is guaranteed to be a disjoint union.)
- **Split Provenance.** Given a split transformation S where $S(I) = (O_1, \ldots, O_r)$ and $I = O_1 \cup \ldots \cup O_r$, the provenance of an output element $o \in O_k$ is the corresponding element $i \in I$, where $i = o$.

Above, we defined the provenance of a single element o for each type of transformation. We can also define the provenance of an output subset $O^* \subseteq O$.

Definition 3.3.1 (Provenance of Output Subset) Consider a transformation instance $O = T(I)$. For any output element $o \in O$, let $P(o)$ be the provenance of o. Consider output subset $O^* \subseteq O$. The *provenance of output subset* O^* is $P(O^*) = \bigcup_{o \in O^*} P(o)$. □

The provenance of an output subset $O^* \subseteq O$ is simply the union of the provenance for all elements $o \in O^*$.

We can verify easily that each of the natural provenance specifications given above for map, reduce, union, and split guarantee correct provenance according to Definition 2.2.4. Furthermore, the provenance specifications for map, union, and split are all minimal according to Definition 2.2.6. However, reduce provenance is not always minimal, as shown by the following example.

Example 3.3.1 Consider a reduce function **MovieAgg** with input data set SalesData containing movie-sales pairs: $\mathsf{SalesData}(\mathsf{movie}, \mathsf{sales}) = \{(Inception, 10), (Twilight, 20), (Twilight, 0)\}$. Transformation **MovieAgg** sums the sales for each movie, producing output set MovieSales $= \{(Inception, 10), (Twilight, 20)\}$. The reduce provenance specified for output element $o = (Twilight, 20)$ is the group $\{(Twilight, 20), (Twilight, 0)\}$. However, the minimal provenance of o is simply $\{(Twilight, 20)\}$, since input element $(Twilight, 0)$ has no impact on the presence or absence of o. □

Now suppose that correct provenance has been specified for each transformation in a GMRW as above. In Section 2.3, we provided a general definition of workflow provenance based on transformation provenance (Definition 2.3.1), and that same definition applies to GMRWs. We then identified when workflow provenance as defined in Definition 2.3.1 is guaranteed to be correct, minimal, or weakly correct. There are many GMRWs in which workflow provenance is not guaranteed to satisfy any of these properties, since even weak correctness is only guaranteed when the workflow contains at most one nonmonotonic transformation (Theorem 2.3.6). But there is an even weaker property that we can guarantee for a large class of GMRWs that can still be useful for debugging. Here we consider when workflow provenance in GMRWs satisfies the following "replay" property.

Property 3.3.1 (Replay Property) Consider an output element o, and let $P_W(o) = (I_1^*, \ldots, I_m^*)$ be the workflow provenance of o in workflow W as defined in Definition 2.3.1, using the definitions of transformation provenance at the start of

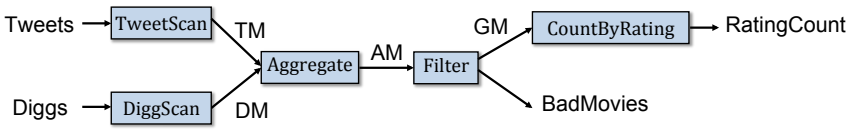

Figure 3.3: Modified movie workflow example with ill-behaved provenance.

this section. If we run $I_1^*, \ldots, I_m^*$ through W, denoted $W(I_1^*, \ldots, I_m^*)$, then o is part of the result: $o \in W(I_1^*, \ldots, I_m^*)$. □

Note that the replay property is a weaker property than weak correctness (Definition 2.3.9), which further requires that for all $\langle I_1', \ldots, I_m' \rangle$ such that $I_1^* \subseteq I_1' \subseteq I_1, \ldots, I_m^* \subseteq I_m' \subseteq I_m$, $o \in W(I_1', \ldots, I_m')$. The replay property holds for our running example and for a very large class of GMRWs, but even the replay property does not hold for all GMRWs. Suppose our running example is changed in the following two ways (shown in Figure 3.3):

- **TweetScan** may output more than one element when a tweet discusses multiple movies, i.e., **TweetScan** is now one-many.
- Output GoodMovies (GM) is input to an additional reduce function **CountByRating**, which emits the number of movies for each good median rating 6–10.

Using the modified workflow, here is a scenario where the replay property does not hold. Suppose Tweets consists of three tweets: tweet t_1 produces ratings (*Inception*,8) and (*Twilight*,8); tweet t_2 produces rating (*Twilight*,2); tweet t_3 produces rating (*Twilight*,5). Let Diggs be empty. Dropping the 1000 ratings requirement, for these input data sets, output RatingCount contains (rating:8,count:1) based on *Inception* with median rating 8, while output BadMovies contains (*Twilight*) with median rating 5.

For the output element o = (rating:8,count:1) in RatingCount, o's workflow provenance is $P_W(o) = \{t_1\}$, which contains all of the elements in Tweets related to those movies with a median rating of 8 (just *Inception*). However, suppose we reran the workflow on o's provenance, i.e., using tweet t_1 only. The result in output RatingCount is the "incorrect" value (rating:8,count:2). Only one of the three ratings for *Twilight* is used, therefore its median is also computed as 8. In terms of our formalism, $o \notin W(P_W(o))$.

Let us try to understand what characteristics of the example workflow caused the replay property to be violated. When **TweetScan** is rerun on $P_W(o) = \{t_1\}$, it produces two elements: (*Inception*,8) and (*Twilight*,8). When reduce function **Aggregate** is run on the two elements produced by tweet t_1, the correct median (*Inception*,8) is produced, but so is incorrect median (*Twilight*,8), since not all data for *Twilight* is being processed by the workflow. The incorrect median wouldn't be harmful on its own, but when it is combined with the correct median in the **CountByRating** transformation, an incorrect output is produced. Note that if either reduce function **Aggregate** or **CountByRating** were monotonic, the problem would not have occurred: If **Aggregate** were monotonic, it could not produce an incorrect output value, since it is operating on a subset of the correct input. If **CountByRating** were monotonic, then extra input could only create additional output, not eliminate the correct output.

It turns out that the specific pattern of three (or more) transformations with certain properties, as exhibited by the above example, is the *only* case in which rerunning a GMRW on the provenance of an output element o is not guaranteed to produce o. In fact, for the replay property $o \in W(I_1^*, \ldots, I_m^*)$ to be violated, the one-many map or many-many reduce function must precede the two nonmonotonic reduce functions in the workflow.

Theorem 3.3.1 Consider a GMRW W composed of transformations $T_1, \ldots, T_n$, with initial inputs $I_1, \ldots, I_m$. Let o be any output element, and consider $P_W(o) = (I_1^*, \ldots, I_m^*)$.

1. If all map and reduce functions in W are one-one or many-one, respectively, then $o = W(I_1^*, \ldots, I_m^*)$. (Note this result is stronger than the general $o \in W(I_1^*, \ldots, I_m^*)$.)
2. If there is at most one nonmonotonic reduce function in W, then $o \in W(I_1^*, \ldots, I_m^*)$. □

The proof for Part 2 follows directly from the preservation of weak correctness (Theorem 2.3.6), since W has at most one nonmonotonic transformation. We now prove Part 1. We actually prove a stronger property: Let O be the output of W and let

$o_1, \ldots, o_n$ be elements of O. Then $W(P_W(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$. The proof is by induction on the structure of W.

Base case $W = M$ where M is a map function. By definition, $P_M(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_M(o_j))$. For $j = 1..n$, $P_M(o_j) = \{i_j\}$ such that $M(\{i_j\}) = \{o_j\}$. By the definition of map functions, and since M is one-one, $M(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(M(i_j)) = \{o_1, ..., o_n\}$.

Base case $W = R$ where R is a reduce function. By definition, $P_R(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_R(o_j))$. For $j = 1..n$, $P_R(o_j) = G_j$ such that $R(G_j) = \{o_j\}$. By the definition of reduce functions, and since R is many-one, $R(G_1 \cup \ldots \cup G_n) = \bigcup_{j=1}^{n}(R(G_j)) = \{o_1, ..., o_n\}$.

Base case $W = U$ where U is a union transformation. U has input data sets $I_1, \ldots, I_m$. By definition, $P_U(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_U(o_j))$. For $j = 1..n$, $P_U(o_j) = \{i_j\}$ where i_j is the element in some I_k that corresponds to o_j. For any set $\mathbb{I}$ that combines subsets of U's input sets $I_1, \ldots, I_m$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_m)$, where each $I'_k = \mathbb{I} \cap I_k$. Then $U(\{i_j\}) = \{o_j\}$. By the definition of union transformations, $U(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(U(\{i_j\})) = \{o_1, ..., o_n\}$.

Base case $W = S$ where S is a split transformation. S has output sets $O_1, \ldots, O_r$. By definition, $P_S(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_S(o_j))$. For $j = 1..n$, o_j is in some O_k, and $P_S(o_j) = \{i_j\}$ such that $S(\{i_j\}) = O'_1, \ldots, O'_r$, where $O'_k = \{o_j\}$ and $O'_h = \emptyset$ for $h \neq k$. Since split transformations are context-independent on each element (Section 3.2.2), $S(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(S(\{i_j\})) = \{o_1, ..., o_n\}$.

Now suppose workflows $W'_1, \ldots, W'_p$ satisfy the inductive hypothesis: $W'(P_{W'}(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$ for any $o_1, ..., o_n$ in the output of W'. Consider an additional transformation T and the workflow W that is constructed by making the outputs of $W'_1, \ldots, W'_p$ the inputs of T. (For all T other than union transformations, $p = 1$.) We use $\circ$ for workflow composition.

Map: Suppose $W = W' \circ M$. Let $P_M(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. Since M is one-one, by the definitions of map provenance and map functions, $M(\{o'_1, ..., o'_n\}) = \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) = \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$.

Reduce: Suppose $W = W' \circ R$. Let $P_R(\{o_1, ..., o_n\}) = (G_1 \cup ... \cup G_n)$, where each

group G_j produces o_j. Since R is many-one, by the definitions of reduce provenance and reduce functions, $R(G_1 \cup ... \cup G_n) = \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(G_1 \cup \ldots \cup G_n)) = G_1 \cup \ldots \cup G_n$. Thus $W(P_W(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$.

Union: Suppose W is composed of $W'_1, \ldots, W'_p$ followed by U. U has input data sets $I^U_1, \ldots, I^U_p$. Let $P_U(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. For any set $\mathbb{I}$ that combines subsets of U's input sets $I^U_1, \ldots, I^U_p$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_p)$, where each $I'_k = \mathbb{I} \cap I^U_k$. By the definitions of union provenance and union transformations, $U(\{o'_1, ..., o'_n\}) = \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) = \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$.

Split: Suppose $W = W' \circ S$. S has output sets $O_1, \ldots, O_r$. Let $P_S(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definitions of split provenance and split transformations, $S(\{o'_1, ..., o'_n\}) = O'_1, \ldots, O'_r$, where $O'_1 \cup \ldots \cup O'_r = \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) = \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$. □

For workflows not satisfying Theorem 3.3.1, workflow provenance as defined in Definition 2.3.1 might still be useful for debugging. However, we believe it is important to be able to rerun a workflow on an output element's provenance—and get the output element in the result—as part of the use of provenance for debugging purposes.[1] In the GMRW context, we can automatically augment any ill-behaved workflow W with extra filters that ensure $o \in W(P_W(o))$ for any output element o. This result is formalized in the following Corollary.

Corollary 3.3.1 Consider a GMRW W composed of transformations $T_1, \ldots, T_n$, with initial inputs $I_1, \ldots, I_m$. Let o be any output element, and consider $P_W(o) = (I^*_1, \ldots, I^*_m)$. Let W^* be constructed from W by replacing all nonmonotonic reduce functions T_j with $T_j \circ \sigma_j$, where σ_j is a filter that removes all elements from the output of T_j that were not in the output of T_j when $W(I_1, \ldots, I_m)$ was run originally.[2] Then $o \in W^*(P_W(o))$.

[1] Provenance-based selective refresh, comprised of backward tracing followed by forward propagation, requires a similar property. In Chapter 4 our approach to selective refresh for arbitrary workflows would deem this example workflow "unsafe" and disallow it.

[2] We assume all intermediate/output data sets have been stored for provenance-tracing purposes; see Section 3.4.

The proof of this Corollary involves two lemmas.

Lemma 3.3.1 Consider a GMRW W with output O. Suppose there are no nonmonotonic reduce functions in W. Let $o_1, \ldots, o_n$ be elements of O. Then $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$.

Proof of Lemma 3.3.1 By induction on the structure of W.

Base case $W = M$ where M is a map function. By definition, $P_M(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_M(o_j))$. For $j = 1..n$, $P_M(o_j) = \{i_j\}$ such that $o_j \in M(\{i_j\})$. By the definition of map functions, $M(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(M(i_j)) \supseteq \{o_1, ..., o_n\}$.

Base case $W = R$ where R is a reduce function. By definition, $P_R(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_R(o_j))$. For $j = 1..n$, $P_R(o_j) = G_j$ such that $o_j \in R(G_j)$. By the definition of reduce functions, $R(G_1 \cup \ldots \cup G_n) = \bigcup_{j=1}^{n}(R(G_j)) \supseteq \{o_1, ..., o_n\}$.

Base case $W = U$ where U is a union transformation. U has input data sets $I_1, \ldots, I_m$. By definition, $P_U(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_U(o_j))$. For $j = 1..n$, $P_U(o_j) = \{i_j\}$ where i_j is the element in some I_k that corresponds to o_j. For any set $\mathbb{I}$ that combines subsets of U's input sets $I_1, \ldots, I_m$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_m)$, where each $I'_k = \mathbb{I} \cap I_k$. Then $U(\{i_j\}) = \{o_j\}$. By the definition of union transformations, $U(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(U(\{i_j\})) = \{o_1, ..., o_n\}$.

Base case $W = S$ where S is a split transformation. S has output sets $O_1, \ldots, O_r$. By definition, $P_S(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_S(o_j))$. For $j = 1..n$, o_j is in some O_k. $P_S(o_j) = \{i_j\}$ such that $S(\{i_j\}) = O'_1, \ldots, O'_r$, where $O'_k = \{o_j\}$ and $O'_h = \emptyset$ for $h \neq k$. Since split transformations are context-independent on each element, $S(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(S(\{i_j\})) = \{o_1, ..., o_n\}$.

Now suppose workflows $W'_1, \ldots, W'_p$ satisfy the inductive hypothesis: $W'(P_{W'}(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$ for any $o_1, ..., o_n$ in the output of W'. Consider an additional transformation T and the workflow W that is constructed by making the outputs of $W'_1, \ldots, W'_p$ the inputs of T.

Map: Suppose $W = W' \circ M$. Let $P_M(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definitions of map provenance and map functions, if $I' \supseteq \{o'_1, ..., o'_n\}$, then $M(I') \supseteq \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) \supseteq \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$.

Reduce: Suppose $W = W' \circ R$. Let $P_R(\{o_1, ..., o_n\}) = (G_1 \cup ... \cup G_n)$, where each group G_j produces o_j. Since R is monotonic, if $I' \supseteq G_1 \cup ... \cup G_n$, then $R(I') \supseteq \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(G_1 \cup \ldots \cup G_n)) \supseteq G_1 \cup \ldots \cup G_n$. Thus $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$.

Union: Suppose W is composed of $W'_1, \ldots, W'_p$ followed by U. U has input data sets $I^U_1, \ldots, I^U_p$. Let $P_U(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. For any set $\mathbb{I}$ that combines subsets of U's input sets $I^U_1, \ldots, I^U_p$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_p)$, where each $I'_k = \mathbb{I} \cap I^U_k$. By the definitions of union provenance and union transformations, if $I' \supseteq \{o'_1, ..., o'_n\}$, then $U(I') \supseteq \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) \supseteq \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$.

Split: Suppose $W = W' \circ S$. S has output sets $O_1, \ldots, O_r$. Let $P_S(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definitions of split provenance and split transformations, if $I' \supseteq \{o'_1, ..., o'_n\}$, then $S(I') = O'_1, \ldots, O'_r$ where $O'_1 \cup \ldots \cup O'_r \supseteq \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) \supseteq \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$. □

Lemma 3.3.2 Consider a GMRW W with output O. Suppose there are no nonmonotonic reduce functions in W. Let $o_1, \ldots, o_n$ be elements of O. Then $W(P_W(\{o_1, ..., o_n\})) \subseteq O$.

Proof of Lemma 3.3.2 By induction on the structure of W.

Base case $W = M$ where M is a map function. Let M have input set I and output set O. Let $P_M(\{o_1, ..., o_n\}) = \{i_1, ..., i_n\}$. By the definition of map functions, $\{i_1, ..., i_n\} \subseteq I$, and $M(\{i_1, ..., i_n\}) \subseteq M(I) = O$.

Base case $W = R$ where R is a reduce function. Let R have input set I and output set O. By definition, $P_R(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_R(o_j))$. For $j = 1..n$, $P_R(o_j) = G_j$ such that $G_j \subseteq I$. Since R is monotonic and $G_1 \cup \ldots \cup G_n \subseteq I$, $R(G_1 \cup \ldots \cup G_n) \subseteq R(I) = O$.

Base case $W = U$ where U is a union transformation. U has input data sets $I_1, \ldots, I_m$. By definition, $P_U(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_U(o_j))$. For $j = 1..n$, $P_U(o_j) = \{i_j\}$ where i_j is the element in some I_k that corresponds to o_j. For any set $\mathbb{I}$ that combines subsets of U's input sets $I_1, \ldots, I_m$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_m)$, where each $I'_k = \mathbb{I} \cap I_k$. Then $U(\{i_j\}) = \{o_j\}$. By the definition of union transformations,

$U(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(U(\{i_j\})) = \{o_1, ..., o_n\} \subseteq O$.

Base case $W = S$ where S is a split transformation. S has output sets $O_1, \ldots, O_r$. By definition, $P_S(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_S(o_j))$. For $j = 1..n$, o_j is in some O_k. $P_S(o_j) = \{i_j\}$ such that $S(\{i_j\}) = O'_1, \ldots, O'_r$, where $O'_k = \{o_j\}$ and $O'_h = \emptyset$ for $h \neq k$. Since split transformations are context-independent on each element, $S(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(S(\{i_j\})) = \{o_1, ..., o_n\} \subseteq O$.

Now suppose workflows $W'_1, \ldots, W'_p$ satisfy the inductive hypothesis: $W'(P_{W'}(\{o_1, ..., o_n\})) \subseteq O'$ for any $o_1, ..., o_n$ in O', where O' is the output of W'. Consider an additional transformation T and the workflow W that is constructed by making the outputs of $W'_1, \ldots, W'_p$ the inputs of T.

Map: Suppose $W = W' \circ M$. Let $P_M(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definition of map provenance, $\{o'_1, ..., o'_n\} \subseteq O'$. Let O^* denote $W'(P_{W'}(\{o'_1, ..., o'_n\}))$. By the inductive hypothesis, $O^* \subseteq O'$. By the definition of map functions, since $O^* \subseteq O'$, $M(O^*) \subseteq M(O') = O$. Thus, $W(P_W(\{o_1, ..., o_n\})) \subseteq O$.

Reduce: Suppose $W = W' \circ R$. Let $P_R(\{o_1, ..., o_n\}) = (G_1 \cup ... \cup G_n)$, where each group G_j produces o_j. By the definition of reduce provenance, $G_1 \cup \ldots \cup G_n \subseteq O'$. Let O^* denote $W'(P_{W'}(G_1 \cup \ldots \cup G_n))$. By the inductive hypothesis, $O^* \subseteq O'$. Since R is monotonic and $O^* \subseteq O'$, $R(O^*) \subseteq R(O') = O$. Thus, $W(P_W(\{o_1, ..., o_n\})) \subseteq O$.

Union: Suppose W is composed of $W'_1, \ldots, W'_p$ followed by U. U has input data sets $I^U_1, \ldots, I^U_p$. Let $P_U(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definition of union provenance, $\{o'_1, ..., o'_n\} \subseteq O'$. For any set $\mathbb{I}$ that combines subsets of U's input sets $I^U_1, \ldots, I^U_p$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_p)$, where each $I'_k = \mathbb{I} \cap I^U_k$. Let O^* denote $W'(P_{W'}(\{o'_1, ..., o'_n\}))$. By the inductive hypothesis, $O^* \subseteq O'$. By the definition of union transformations, since $O^* \subseteq O'$, $U(O^*) \subseteq U(O') = O$. Thus, $W(P_W(\{o_1, ..., o_n\})) \subseteq O$.

Split: Suppose $W = W' \circ S$. S has output sets $O_1, \ldots, O_r$. Let $P_S(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definition of split provenance, $\{o'_1, ..., o'_n\} \subseteq O'$. Let O^* denote $W'(P_{W'}(\{o'_1, ..., o'_n\}))$. By the inductive hypothesis, $O^* \subseteq O'$. By the definition of O', $S(O') = O_1, \ldots, O_r$. Let $S(O^*) = O^*_1, \ldots, O^*_r$. By the definition of split transformations, since $O^* \subseteq O'$, each $O^*_j \subseteq O_j$. Thus, $W(P_W(\{o_1, ..., o_n\})) \subseteq O$. $\square$

Proof of Corollary 3.3.1 We prove a stronger property: Let O be the output of W and let $o_1, ..., o_n$ be elements of O. Then $O \supseteq W^*(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$. This property clearly implies the corollary. The proof is by induction on the structure of W. The base case and inductive step proofs follow directly from the analogous cases in Lemmas 3.3.1 and 3.3.2, with the exceptions of nonmonotonic reduce functions.

Base case $W = R$ where R is a nonmonotonic reduce function. By definition, $P_R(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_R(o_j))$. For $j = 1..n$, $P_R(o_j) = G_j$ such that $o_j \in R(G_j)$. By the definition of reduce functions, $R(G_1 \cup \ldots \cup G_n) = \bigcup_{j=1}^{n}(R(G_j)) \supseteq \{o_1, ..., o_n\}$. Let σ_R be the filter associated with R. Since $\{o_1, \ldots, o_n\} \subseteq O$, no element in $\{o_1, \ldots, o_n\}$ is removed by σ_R. Thus, $\sigma_R(R(G_1 \cup \ldots \cup G_n)) \supseteq \{o_1, ..., o_n\}$. Since σ_R filters only elements not in O, $O \supseteq (R \circ \sigma_R)(G_1 \cup \ldots \cup G_n)$.

Inductive step for Reduce: Suppose $W = W' \circ R$ where R is nonmonotonic. Let $P_R(\{o_1, ..., o_n\}) = (G_1 \cup ... \cup G_n)$, where each group G_j produces o_j. Let O^* denote $W'^*(P_{W'}(G_1 \cup \ldots \cup G_n)$. Let O' be the output of W'. By the inductive hypothesis, $O' \supseteq O^* \supseteq (G_1 \cup \ldots \cup G_n)$.

By the definition of reduce provenance, each group G_j is equal to the set of all elements in O' with G_j's key. Since $G_j \subseteq O^* \subseteq O'$, there cannot be any element in $O^* - G_j$ that has G_j's key. Thus, each group G_j is equal to the set of all elements in O^* with G_j's key. $R(O^*) \supseteq \bigcup_{j=1}^{n} R(G_j) \supseteq \{o_1, \ldots, o_n\}$. Let σ_R be the filter associated with R. Since $\{o_1, \ldots, o_n\} \subseteq O$, no element in $\{o_1, \ldots, o_n\}$ is removed by σ_R. Thus, $W^*(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$. Since σ_R is the final step of W^*, and σ_R filters only elements not in O, $O \supseteq W^*(P_W(\{o_1, ..., o_n\}))$. □

3.4 Provenance Capture & Tracing

For now let us assume that all input, intermediate, and output data sets are persistent. We also assume all data is stored in files, so (*file*, *offset*) can be used as a globally unique identifier (ID) and locator for any data element. To capture the provenance specified in Section 3.3, for map functions we add to each output data element the unique ID for the input element that generated it. For reduce functions, we add to

each output element a list of IDs for the input elements whose grouping key corresponds to the output element. (If a function has multiple input sets and/or multiple output sets, the union and split operations we added in Section 3.2.2 for the purposes of analysis are performed within the function itself.) In a MapReduce framework, wrapping map and reduce functions automatically to emit these extra fields during execution is a relatively straightforward process, as we have demonstrated with our RAMP system (Section 3.5).

An alternative for reduce functions would be to annotate output elements with the grouping key, instead of storing explicit IDs for the relevant input elements. (In fact, the grouping key is frequently preserved in the output of a reduce function anyway.) This approach would save space compared with the ID approach—particularly for reduce functions with large "fan-in"—but it would slow down backward tracing significantly, perhaps mitigated some by building special indexes. In Section 3.5, we measure the space overhead of storing provenance IDs for our running example.

Now consider backward tracing. The following algorithm returns workflow provenance as defined in Definition 2.3.1.

Algorithm 3.4.1 (Backward Tracing) Consider a GMRW W with initial inputs $I_1, \ldots, I_m$. Recursive function *backward_trace* returns the provenance of a set E of data elements from a single input, intermediate, or output data set:

---

backward_trace$(E, W, \{I_1, \ldots, I_m\})$:
if $E \subseteq I_k$ for $1 \leq k \leq m$ **then return** E;
else{ $T \leftarrow$ transformation that output the set containing E;
$E' \leftarrow$ input elements to T with (*file*, *offset*) in E;
$E'_1, \ldots E'_n \leftarrow E'$ partitioned by input sets;
$I^* \leftarrow \emptyset$;
for $i = 1..n$ **do**
$I^* \leftarrow I^* \cup$ *backward_trace*$(E'_i, W, \{I_1, \ldots, I_m\})$;
return I^*; }}

Notice in this algorithm that the only time elements are extracted from a set (construction of E' in the fourth line), they are fetched based on their (*file*, *offset*) identifiers, which also serve as a locators.

Now consider forward tracing. The overall algorithm is simply the converse of backward tracing:

Algorithm 3.4.2 (Forward Tracing) Consider a GMRW W with final outputs $O_1, \ldots, O_r$, and any set E of data elements from a single input, intermediate, or output data set. Algorithm *forward_trace* returns the output elements derived from any element in E:

---

forward_trace$(E, W, \{O_1, \ldots, O_r\})$:
  if $E \subseteq O_k$ for $1 \leq k \leq r$ **then return** E;
  else { $T \leftarrow$ transformation that processes E;
    $E' \leftarrow$ output elements from T with ID corresponding
      to an element in E;
    $E'_1, \ldots E'_n \leftarrow E'$ partitioned by output sets;
    $O^* \leftarrow \emptyset$;
    for $i = 1..n$ **do**
      $O^* \leftarrow O^* \cup$ *forward_trace*$(E'_i, W, \{O_1, \ldots, O_r\})$;
    return O^*;}

Our provenance capture scheme is biased towards backward tracing, which we assume is a more frequent operation. In the forward tracing process, we are given a set of input elements whose IDs are their (*file*, *offset*) pairs, and we need to find all output elements containing those IDs. Recall IDs were added to output elements by a wrapped map or reduce function: a single ID was added to each output element of a map function, and a list of IDs was added to each output element of a reduce function. Without auxiliary structures, each forward-tracing step would require a complete scan of the output data set, so to facilitate forward tracing we recommend building indexes on the special ID fields.

3.5 RAMP System

We have built a system called *RAMP* (*Reduce And Map Provenance*) for capturing and tracing provenance in GMRWs. (A further development of this system led by another author is described in [35].) RAMP is built on top of the open-source MapReduce framework Hadoop [8], using the *Dumbo* module [14]. To capture provenance, RAMP wraps each map and reduce function to store IDs exactly as described in Section 3.4. RAMP's approach to provenance capture is wrapper-based and transparent to Hadoop, retaining Hadoop's parallel execution and fault tolerance.

The high-level structure of the RAMP system is shown in Figure 3.4. The system has two components: the RAMP Wrapper and the RAMP Tracer. In RAMP, users execute a GMRW just as if they were using Hadoop with Dumbo: by submitting a single Dumbo script written in Python containing all of the `map()` and `reduce()` functions in the GMRW. As always, the Dumbo script must also specify the control flow of the GMRW. The RAMP Wrapper wraps the `map()` and `reduce()` functions in the Dumbo script to capture provenance, creating a new Dumbo script with wrapped `map'()` and `reduce'()` functions to be executed by Hadoop. In RAMP, all input, intermediate, and output data sets are stored in files, so RAMP uses (*file*, *offset*) as an ID for each data element. After a GMRW is executed, users can trace the provenance of the output element with ID q by submitting the `trace(`q`)` command to the RAMP Tracer. The RAMP Tracer executes the tracing commands directly on the provenance stored by the wrapped `map'()` and `reduce'()` functions during GMRW execution.

We present performance experiments conducted using RAMP on the original running example of this chapter (Figure 3.1), with minor modifications: We use Twitter data only, and to keep the data sizes large we presume all tweets are discussing movies—we randomly select a movie title for those that do not contain one.

The cluster we used for our experiments consisted of 4 small Amazon EC2 instances (each with 1.7 GB memory, CPU capacity equivalent to a 1.7 GHz Xeon processor, 160 GB instance storage, Ubuntu 10.04). One instance served as the master Hadoop node, and the other three instances served as slave nodes.

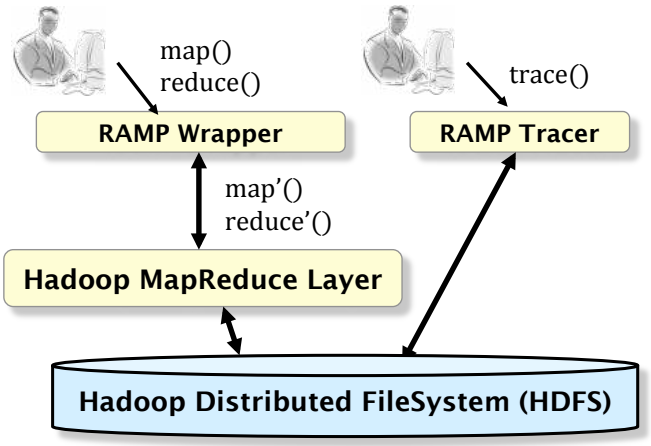

Figure 3.4: Structure of the RAMP system.

Our performance results are summarized as follows:

- We first measure the time and space overhead of provenance capture, for varying input data sizes. Overall, on our running example, provenance capture incurs 111% time overhead and 45% space overhead during workflow execution. Details are reported in Section 3.5.1.
- We then measure the time to backward-trace output elements when provenance has been captured, for varying input data sizes and varying sizes for the output set to be traced. Overall, we see that the running time for backward-tracing increases linearly with respect to both initial input size and tracing set size.

Backward tracing when provenance hasn't been captured requires rerunning map and reduce functions, looking for the output elements being traced, similar to the algorithms in [21]. Although some short-circuiting is possible, running time is proportionate to rerunning the entire workflow. Thus, given the reasonable time and space overhead of provenance capture, tracing just a small number of output elements warrants performing capture during workflow execution.

3.5.1 Performance: Capture

We report time and space overhead associated with capturing provenance. For each input data size, we ran the sample workflow with and without capturing provenance. To give the biggest advantage to *not* capturing provenance, we combined **TweetScan** (map) and **Aggregate** (reduce) into one MapReduce job. When we captured provenance, we separated the two functions so that we could store intermediate data and capture (*file*, *offset*) pairs.

Figure 3.5 reports time overhead. We observe that for 1.2 GB of input data, provenance capture incurs 111% time overhead (2317s vs. 1097s). Figure 3.6 reports space overhead, plotting the total amount of data involved in the workflow (input, intermediate, and output) with and without provenance capture. For 1.2 GB of input data, provenance capture incurs a 45% space overhead. As a comparison point, we note that for fault tolerance, people are willing to pay 100% to 200% space overhead (as well as a time overhead to keep copies up to date). Thus, 45%-111% overhead during provenance capture seems modest to obtain the benefits of provenance tracing.

A significant source of both time and space overhead when capturing provenance was due to processing **TweetScan** and **Aggregate** separately, rather than combining them into a single MapReduce job as we did for the non-provenance measurements. Specifically, the intermediate data set between **TweetScan** and **Aggregate** comprises a large fraction of the space overhead; the remaining overhead is the relatively compact additional IDs in the other intermediate data set (AM in Figure 3.1) and the output data. Certainly one obvious optimization is to combine these two functions even in the provenance-capture case, if provenance involving the intermediate data is not of interest. In Section 3.6 we briefly discuss function merging as an area of future work.

3.5.2 Performance: Tracing

We measured the time to backward-trace output elements when provenance has been captured, for varying input data sizes (Figure 3.7) and varying sizes for the output set to be traced (Figure 3.8). We found that fetching the final textual tweets in the

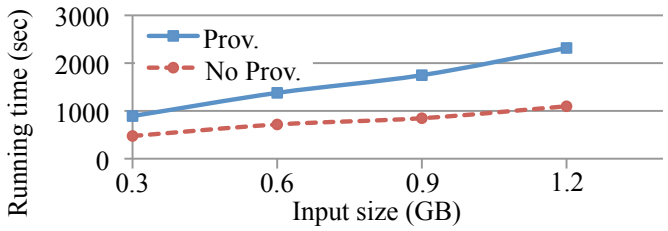

Figure 3.5: Time overhead of provenance capture.

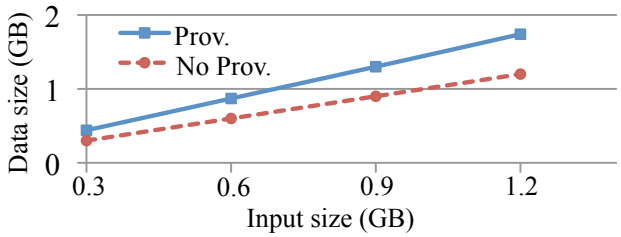

Figure 3.6: Space overhead of provenance capture.

input data dominated tracing costs, so in both experiments we also measured tracing time without the final fetches. For our largest input size, tracing one element without the final fetches took approximately 20 seconds (with fetches took approximately one minute). Note that even when we trace a single output element, the relevant data sizes become quite large: In our 1.2GB input set, the average number of movie ratings for one output element is 200,000. Because we have a fixed number of movies, the provenance of individual output elements grows linearly with input size, explaining the linear growth in Figure 3.7. Also, since no output elements share provenance, the number of lookups during backward tracing depends linearly on tracing set size, explaining the linear growth in Figure 3.8.

3.6 Optimizations

We have proposed provenance capture and tracing techniques that apply generally to any workflow composed of map and reduce functions. Our initial implementation has

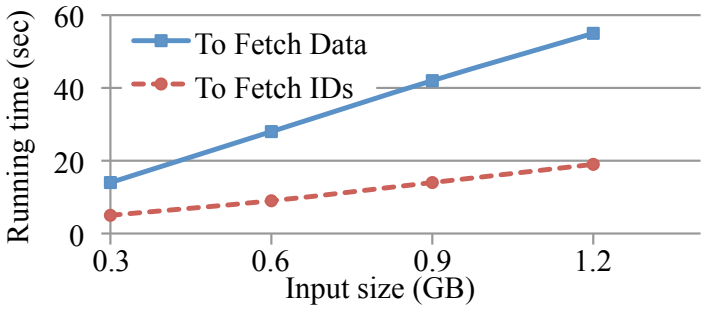

Figure 3.7: Time to backward-trace one output element.

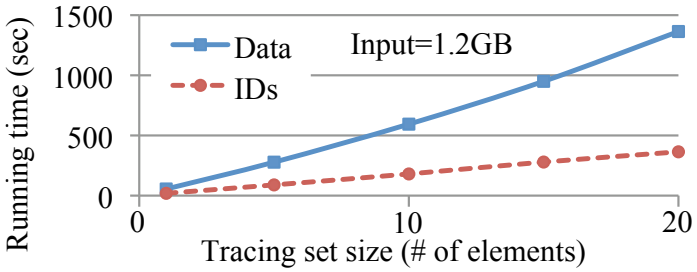

Figure 3.8: Time to backward-trace set of output elements.

chosen a fixed strategy that retains all intermediate data sets, records provenance for each transformation, and traces provenance transformation-by-transformation.

Clearly there are alternatives and optimizations to our approach for specific GMRWs, and for specific provenance capture and tracing needs. The most obvious optimization is to logically "merge" adjacent functions with respect to provenance. When we logically merge two functions f_1 and f_2, we no longer store the intermediate data set between the two functions, and we capture and trace the provenance of f_2's output with respect to f_1's input. In the extreme case, we could logically merge all functions in a workflow, capturing and storing only provenance between the initial inputs and the final outputs. There are a variety of possible strategies for merging transformations. Reference [21] merges transformations based on user-specified transformation *properties* and knowledge of how composing transformation properties impacts tracing performance. In Chapter 5 we will merge transformations when their

logical provenance specifications can be combined without losing any workflow provenance information. As explored in [21], merging some transformation pairs is natural and easy, while in other cases merging can incur significant overhead at capture time, although tracing performance always improves. Also, for some applications, the ability to trace through intermediate data is considered a feature, e.g., for step-by-step debugging.

There are other, less sweeping, optimizations that could be explored. For example, our brute-force storage of (*file*, *offset*) IDs in output data sets certainly always works, but there may be cases where we could compact or streamline provenance storage, or where we could exploit fields (such as reduce-keys) that are already present, perhaps building additional indexes for efficient lookups.

3.7 Related Work

There has been tremendous interest recently in high-performance parallel data processing specified via map and reduce functions, e.g., [8, 24, 43]. In addition, higher-level platforms have been built on top of these systems to make data-parallel programming easier, e.g., [17, 34, 38]. Regardless of which level they operate on, none of these systems or frameworks provides explicit functionality or even formal underpinnings for provenance. At the same time, although there has been a large body of work in lineage and provenance (as discussed in Section 1.4), none of this work considers the specific case of GMRWs, whose special properties and opportunities in the context of provenance are the focus of this chapter.

Reference [21], which perhaps is most related, provides 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. Our map and reduce functions fall into the hierarchy, but they are specific enough that we can capture provenance automatically using a wrapper-based approach. Also, while [21] allows acyclic graphs of transformations, it does not investigate behavioral properties when provenance is traced recursively through them. We show in this chapter that recursive provenance tracing can yield ill-behaved results in certain subtle cases. Finally, in

this chapter we considered the overhead incurred gathering extra information during workflow execution to facilitate provenance tracing, a topic not considered in [21].

3.8 Conclusions

We explored provenance for forward and backward tracing in GMRWs. In particular, we showed that the special case of workflows where all transformations are map or reduce functions allows us to capture and exploit provenance more easily and efficiently than for general data-oriented workflows. We identified properties that hold for workflow provenance in GMRWs. We described how provenance can be captured for both map and reduce functions transparently using wrappers in Hadoop. We have built a prototype system as an extension to Hadoop that supports provenance capture and tracing, and we reported performance numbers on the overhead of provenance capture and the cost of provenance tracing.

Chapter 4

Provenance-Based Refresh

4.1 Introduction

Consider a workflow in which the input data sets have been modified since the workflow was run, but the workflow has not been rerun on the modified input. In this chapter we consider the problem of *selectively refreshing* one or more elements in the output data, i.e., computing the latest values of particular output elements based on the modified input data. By exploiting provenance, we can rerun just the relevant computation to refresh the output elements. Our contributions are the following:

- In Section 4.2, we present a formal foundation for the refresh problem, and we introduce *provenance predicates*. We specify a refresh procedure for single transformations, and we identify the properties of transformations and provenance that are required for correct refresh.
- In Section 4.3, we extend our formalization and refresh procedure to data-oriented workflows. We identify an additional workflow property necessary for the "transitive" workflow refresh procedure to produce correct refreshed data.
- In Sections 4.4 and 4.5, we extend our formalism and algorithms to support transformation types that were excluded, for presentation development, from Sections 4.2 and 4.3.

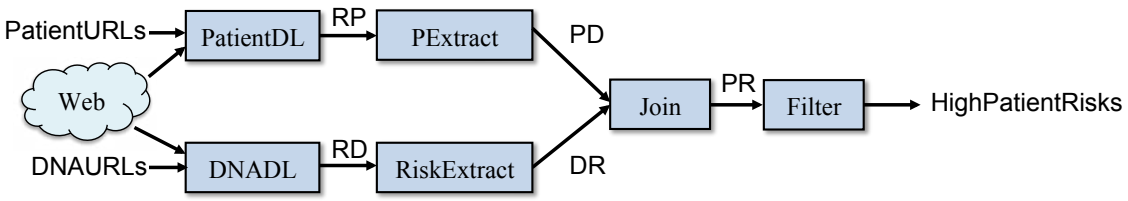

Figure 4.1: Genetic risk workflow example.

- In Section 4.6, we discuss how refresh can still be performed (albeit less efficiently) in workflows for which the workflow property identified in Section 4.3 does not hold.

The remainder of this section introduces a running example. In Section 4.7, we discuss related work, and we conclude in Section 4.8.

4.1.1 Running Example

We present a running example, designed to illustrate challenges and solutions throughout the chapter. Consider *Jen*, a genetic counselor who runs a workflow, shown in Figure 4.1, to calculate her patients' genetic risk profiles. The workflow's input data sets are two lists of URLs: PatientURLs and DNAURLs. PatientURLs point to the patients' genetic test results, recording the patients' DNA sequences at certain DNA locations. DNAURLs identify XML documents describing disease risks associated with specific DNA location-sequence combinations. The workflow involves the following transformations:

- Transformations **PatientDL** and **DNADL** download files located at the URLs in data sets PatientURLs and DNAURLs. They produce directories RawPatientData (abbreviated RP) and RawDNAData (RD), respectively.
- Transformations **PExtract** and **RiskExtract** extract data from the downloaded files into tables: PatientDNA (PD) has attributes for patient name (name), DNA location (loc), and DNA sequence (seq). Table DNARisks (DR) has attributes for DNA location (loc), DNA sequence (seq), disease, and risk.
- Transformation **Join** joins tables PatientDNA and DNARisks on attributes loc and

PatientURLs

	url
1	genetic.com/dl/00501
2	genetic.com/dl/00502
3	genetic.com/dl/00503
4	genetic.com/dl/00504

DNAURLs

	url
1	dnarec.com/1/aaa
2	dnarec.com/2/aga
3	dnarec.com/2/ttt
4	dnarec.com/3/gcc

RawPatientData (RP)

file	
Bob.xml	p: url contains '00501'
Carl.xml	p: url contains '00502'
Denise.xml	p: url contains '00503'
Earl.xml	p: url contains '00504'

RawDNAData (RD)

file	
1-aaa.xml	p: url contains '1/aaa'
2-aga.xml	p: url contains '2/aga'
2-ttt.xml	p: url contains '2/ttt'
3-gcc.xml	p: url contains '3/gcc'

PatientDNA (PD)

	name	loc	seq	
1	Bob	1	aaa	p: file = 'Bob.xml', f: loc = 1
2	Carl	2	ttt	p: file = 'Carl.xml', f: loc = 2
3	Carl	3	ggg	p: file = 'Carl.xml', f: loc = 3
4	Denise	2	aga	p: file = 'Denise.xml', f: loc = 2
5	Earl	2	ata	p: file = 'Earl.xml', f: loc = 2
6	Earl	3	gcc	p: file = 'Earl.xml', f: loc = 3

DNARisks (DR)

	loc	seq	disease	risk	
1	1	aaa	heart	0.6	p: file = '1-aaa.xml'
2	2	aga	heart	0.8	p: file = '2-aga.xml'
3	2	ttt	liver	0.4	p: file = '2-ttt.xml'
4	3	gcc	lung	0.7	p: file = '3-gcc.xml'

PatientRisks (PR)

	name	disease	risk	
1	Bob	heart	0.6	p^{PD}: name='Bob' ∧ loc=1 ∧ seq='aaa', p^{DR}: loc=1 ∧ seq='aaa'
2	Carl	liver	0.4	p^{PD}: name='Carl' ∧ loc=2 ∧ seq='ttt', p^{DR}: loc=2 ∧ seq='ttt'
3	Denise	heart	0.8	p^{PD}: name='Denise' ∧ loc=2 ∧ seq='aga', p^{DR}: loc=2 ∧ seq='aga'
4	Earl	lung	0.7	p^{PD}: name='Earl' ∧ loc=3 ∧ seq='gcc', p^{DR}: loc=3 ∧ seq='gcc'

HighPatientRisks

	name	disease	risk	
1	Bob	heart	0.6	p: name = 'Bob' ∧ disease = 'heart'
2	Denise	heart	0.8	p: name = 'Denise' ∧ disease = 'heart'
3	Earl	lung	0.7	p: name = 'Earl' ∧ disease = 'lung'

Figure 4.2: Genetic risk workflow sample data with provenance predicates. (*'s indicate data elements relevant to HighPatientRisks element #2.)

seq, then projects away the join attributes to produce table PatientRisks (PR) with attributes name, disease, and risk.

- Transformation **Filter** selects from PatientRisks those records with risk > 0.5, producing final output HighPatientRisks.

Figure 4.2 shows sample input data sets along with all intermediate data, and finally output table HighPatientRisks. (The figure also includes *provenance predicates* and *forward filters*, described in Sections 4.2 and 4.4, respectively.) The starred data elements are for reference in the following scenario.

Before seeing a patient, Jen refreshes relevant records in table HighPatientRisks. As an example, we show how our approach efficiently refreshes Denise's heart disease record, i.e., element #2 in table HighPatientRisks. There are two main steps:

(1) Backward tracing: To refresh a given output element, provenance predicates are first used to trace transitively from the output element to its relevant input elements. Starting with HighPatientRisks element #2, its provenance predicate enables tracing backward one step to obtain element #3 in table PatientRisks. From this element, another step is traced backward, resulting in PatientDNA element #4 together with DNARisks element #2. This process continues, to elements *Denise.xml* in RawPatientData and *2-aga.xml* in RawDNAData, and finally to input elements PatientURLs #3 and DNAURLs #2. Details of how provenance predicates support backward tracing in general will be presented in Sections 4.2–4.5.

(2) Forward propagation: Now that the relevant input elements have been found, they are propagated forward. Transformations **PatientDL** and **DNADL** are rerun on input elements PatientURLs #3 and DNAURLs #2 respectively to download the latest data from the web. The resulting elements are then sent through transformations **PExtract** and **RiskExtract**. Suppose that when **RiskExtract** is run on the latest downloaded data, the risk value for DNARisks element #2 has changed from 0.8 to 0.6. Further forward propagation through transformations **Join** and **Filter** sets Denise's new heart-disease risk in HighPatientRisks to 0.6. Note if the new value were ≤ 0.5, then after the **Filter** transformation Denise's record would disappear,

a situation that is also captured by refresh. Details of when forward propagation works correctly, and how, will be covered in Sections 4.2–4.5.

The remainder of the chapter formalizes the basic building blocks and techniques demonstrated in this example, covering a wide class of data types, transformations, and workflows.

4.2 Provenance Predicates and One-Transformation Refresh

For now, we will consider transformations that take a single data set as input and produce a single output set; we will generalize to *multi-input* transformations in Section 4.5. (*Multi-output* transformations do not introduce any interesting challenges for selective refresh, and are thus omitted.) As in Chapter 2, for any transformation T and input data set I, we say that the application of T to I resulting in an output set O, denoted $T(I) = O$, is an *instance* of T. As usual, given a transformation instance $T(I) = O$ and an output element $o \in O$, provenance identifies the input data elements that contributed to o's derivation.

In this chapter, we consider transformations where correct provenance for each output data element is obtained by applying a predicate on the input:

Definition 4.2.1 (Provenance Predicates) Consider a transformation instance $T(I) = O$. We require that each output element $o \in O$ be annotated with a *provenance predicate* p. The elements of I satisfied by predicate p, i.e., the result of *tracing query* $\sigma_p(I)$, constitute o's provenance as specified by predicate p. □

Note that our use of the relational notation $\sigma_p(I)$ for tracing queries is for convenience and familiarity only; data set I need not be a conventional relation. We expect (and it is natural for) predicates to encode provenance that is *correct* according to Definition 2.2.1. Note that predicate p=**true**, selecting all input elements, is always correct provenance according to Definition 2.2.1, although usually not minimal.

Predicates give us a very general notion of provenance. In the GMRW setting from Chapter 3, we can encode provenance using predicates that select on identifiers or grouping keys. Provenance for basic relational operators is naturally expressed as predicates, e.g., for a *group-by aggregation* operator, provenance predicates select relevant input elements based on grouping value. Most of the numerous transformation types covered in [21] are amenable to the provenance-predicate approach. In Chapter 5, we will consider *logical provenance*—provenance specified at the transformation level. Our logical provenance specifications can be translated to provenance predicates. However, since provenance predicates specify provenance at the element level, they are more expressive than logical provenance in general.

Formally, we now assume that the output of each transformation instance produces a set of pairs: $T(I) = O = \{\langle o_1, p_1\rangle, \ldots, \langle o_n, p_n\rangle\}$, combining output elements and their provenance predicates.

Example 4.2.1 Our running example data in Figure 4.2 includes provenance predicates, denoted p. Table PatientDNA includes additional predicates labeled f, which will be explained in Section 4.4. Table PatientRisks includes pairs of provenance predicates, since there are two input data sets to its transformation (Section 4.5). All of the remaining tables have a single provenance predicate for each output element, according to our definition. In all of them it can be seen easily that the predicate p associated with each output element o, when applied to the input table for o's transformation, produces correct provenance for o. □

In our running example, it happens that provenance predicates always select a single input element, but this property is not required in our approach. In general, provenance predicates may select any number of input elements, up to the entire input data set.

Now that we have formalized provenance, we specify a provenance-based refresh procedure for single transformations, and we identify two properties of transformations and their provenance predicates that are required for the procedure to work correctly.

Ignoring provenance for a moment, consider refresh for a single transformation T. Suppose input I has been modified to I^{new}, and we would like to refresh an output element o that was produced by $T(I)$. The refreshed value of o should be the element o' in $T(I^{new})$ that "corresponds" to o, if one exists. (In our work we assume refresh of a single output element produces either a single refreshed element or no element at all, but not several elements.) Note that for refresh to even be well-defined, we need to formalize when an element o' in $T(I^{new})$ corresponds to the element o being refreshed.

One way to make refresh well-defined is to declare one or more output attributes as an immutable key. Then, given an output element o in $T(I)$, the refreshed value of o is the element o' in $T(I^{new})$ with the same key, if one exists.

Example 4.2.2 Consider output directory RawDNAData in Figure 4.2. Intuitively, the file name is an immutable key. To refresh a file in RawDNAData, we could rerun transformation **DNADL** on list DNAURLs, then look for the output file whose file name matches the file we wish to refresh. □

While immutable keys make for a convenient definition, unfortunately performing refresh based on immutable keys (without provenance) typically requires full recomputation of the output data set in order to find the refreshed element. Our goal is to avoid unnecessary computation while performing selective refresh.

Consider as an alternative the following refresh procedure based on provenance predicates. For the remainder of the chapter we assume each transformation T has a (possibly infinite) *input domain* $\mathbb{I}_T$, specifying T's allowable input sets. By giving the option of a known domain for input sets, we allow more transformations to satisfy the requirements for the following procedure. Of course in the general case, $\mathbb{I}_T$ may be the domain of all possible values.

Procedure 4.2.1 (Provenance-Based Refresh) Consider transformation instance $T(I) = O$, and suppose input data set $I \in \mathbb{I}_T$ has been modified to $I^{new} \in \mathbb{I}_T$. To refresh an output element $\langle o, p \rangle \in O$ there are two steps:

1. *Backward tracing:* Run tracing query σ_p on I^{new} to find the subset of I^{new} associated with provenance predicate p.

2. *Forward propagation:* Apply T on $\sigma_p(I^{new})$ to compute the new value $\langle o', p' \rangle$. If the result is empty, o has no refreshed value. □

Example 4.2.3 Suppose we wish to refresh RawDNAData element *2-aga.xml*. Instead of rerunning transformation **DNADL** on the entire input data set as suggested in Example 4.2.2, Procedure 4.2.1 first finds the provenance of the element being refreshed: Provenance predicate p (url contains '2/aga') is applied to the input set DNAURLs to obtain DNAURLs element #2. Next transformation **DNADL** is applied to the selected element only, which yields the refreshed RawDNAData element. □

Example 4.2.3 is more efficient than Example 4.2.2, but does Procedure 4.2.1 always work? We identify two requirements on transformations and provenance predicates under which it does.

Requirement 4.2.1 (Predicate Correctness) Consider any transformation instance $T(I) = O$ for $I \in \mathbb{I}_T$, and any $\langle o, p \rangle \in O$. Then $T(\sigma_p(I)) = \{\langle o, p \rangle\}$. □

Note this requirement is stronger than requiring predicates to specify correct provenance; it further requires that applying T to any predicate-based subset of I cannot yield more than one element. It also implies that no two output elements share the same predicate. For *many-one* (and therefore *one-one*) transformations, this requirement is natural. We will adapt the requirement to also capture *many-many* transformations, but for presentation purposes we defer this topic to Section 4.4.

All of the transformations in our running example satisfy Requirement 4.2.1, with two exceptions: **PExtract** is a many-many transformation that requires the additional machinery introduced in Section 4.4, and **Join** is a multi-input transformation requiring some (minimal) additional machinery covered in Section 4.5.

The second requirement is more subtle, and more central to our approach:

Requirement 4.2.2 (Predicate as Key) Consider any transformation instance $T(I) = O$ for $I \in \mathbb{I}_T$, and any $\langle o, p \rangle \in O$. Then for any $I' \in \mathbb{I}_T$, if $T(\sigma_p(I')) \neq \emptyset$, then $T(\sigma_p(I')) = \{\langle o', p \rangle\}$ for some o', and $\langle o', p \rangle \in T(I')$. □

This requirement intuitively states that when input changes, even if there is a new value for an old output element based on its new provenance, the provenance predicate remains unchanged. Not only does this condition enable efficient refresh, it also identifies what it means for an output element based on new input to be the refreshed version of an old output element:

> *Effectively, we are treating provenance predicates as the immutable keys that coordinate new and old values of output elements.*

Note that, implicitly, we are defining the correct refreshed values of old output elements based on our choice of predicates. In our running example, as well as in all other workflow examples in the thesis, it is natural for transformations to output provenance predicates such that Requirement 4.2.2 is satisfied, and for provenance predicates to act as immutable keys.

The following property, which follows directly from Requirement 4.2.1, further solidifies the key analogy by observing that provenance predicates are unique (under set semantics) within each output data set:

Property 4.2.1 (Unique Predicates) Consider any transformation instance $T(I) = O$ for $I \in \mathbb{I}_T$, and any $\langle o, p\rangle \in O$. There is no $\langle o', p\rangle \in T(I)$ with $o' \neq o$. □

Now let us return to our refresh procedure and see how Requirements 4.2.1 and 4.2.2 guarantee that Procedure 4.2.1 works correctly, under the predicate-as-key approach. Recall, to refresh an output element $\langle o, p\rangle \in T(I)$ after input I has been modified to I^{new}, Procedure 4.2.1 computes $T(\sigma_p(I^{new}))$.

- First suppose $T(\sigma_p(I^{new}))$ produces an empty result. Then there should be no $\langle o', p\rangle$ in $T(I^{new})$, i.e., no new output with provenance predicate p, thus corresponding to o. If there were such an $\langle o', p\rangle$, then by Requirement 4.2.1, $T(\sigma_p(I^{new})) = \{\langle o', p\rangle\}$, contradicting the fact that $T(\sigma_p(I^{new}))$ is empty.
- Now suppose $T(\sigma_p(I^{new}))$ is non-empty. Requirement 4.2.2 guarantees that $T(\sigma_p(I^{new})) = \{\langle o', p\rangle\}$ for some o', and that $\langle o', p\rangle$ is a valid output element in $T(I^{new})$. Thus, o' is the refreshed value for o.

4.3 Refresh for Workflows

Consider any transformations T_1 and T_2. Each transformation takes a data set from its domain as input, and it produces as output a data set annotated with provenance predicates. As usual, the composition $T_1 \circ T_2$ of the two transformations first applies T_1 to an input data set $I_1 \in \mathbb{I}_{T_1}$ to obtain intermediate data set I_2. It then applies T_2 to the data portion (omitting provenance predicates) of I_2 to obtain output data set O. We assume transformations T_1 and T_2 are only composed when the data elements output by T_1 are guaranteed to satisfy the input domain of T_2.

As usual, with associativity, we can denote the linear composition of n transformations as $T_1 \circ T_2 \circ \cdots \circ T_n$. In Section 4.5, we extend our formalism and algorithms to cover workflows where transformations may have multiple input data sets. (As mentioned in Section 4.2, transformations with multiple output sets do not introduce any complexities, and therefore are omitted.) Our running example in Figure 4.1 is a workflow composed of six transformations, with one of them taking multiple inputs.

Consider a workflow $T_1 \circ T_2 \circ \ldots \circ T_n$, and consider an instance of this workflow with input $I_1 \in \mathbb{I}_{T_1}$. Let $I_{i+1} = T_i(I_i)$ for $i = 1..n$. We assume $I_{i+1} \in \mathbb{I}_{T_{i+1}}$ for $i = 1..n-1$. The final output data set is I_{n+1}. We denote this workflow instance as $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$. The following workflow refresh algorithm is a recursive extension of the single-transformation refresh Procedure 4.2.1.

Algorithm 4.3.1 (Workflow Refresh) Consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$. Suppose I_1 has been modified to $I_1^{new} \in \mathbb{I}_{T_1}$. Algorithm *workflow_refresh* recursively refreshes output element $\langle o, p \rangle \in I_{i+1}$:

---

workflow_refresh$(\langle o, p \rangle \in I_{i+1})$:
  if $i = 1$ **then return** $T_1(\sigma_p(I_1^{new}))$
  else { $S = \sigma_p(I_i)$;
    $S' = \bigcup_{\langle o', p' \rangle \in S}$ *workflow_refresh*$(\langle o', p' \rangle \in I_i)$;
    return $T_i(S')$ }

Example 4.3.1 We revisit the original example from Section 4.1.1, now using *workflow_refresh* to refresh (Denise,heart,0.8) (element #2) in HighPatientRisks. The **else** branch of the algorithm traces backward one step by applying provenance predicate p (name='Denise' $\wedge$ disease='heart') to table PatientRisks, yielding element #3. This element is then refreshed recursively: The algorithm traces backward another step by applying provenance predicates to PatientDNA and DNARisks (requiring our extension for multi-input transformations; see Section 4.5), yielding elements #4 and #2 respectively. The recursive refresh continues until the initial transformations **PatientDL** and **DNADL** are reached, for which the **if** branch selects elements #3 and #2 from input sets PatientURLs and DNAURLs respectively.

As the recursion unwinds, the algorithm forward propagates each refreshed element. Transformations **PatientDL** and **DNADL** are run on PatientURLs element #3 and DNAURLs element #2, yielding refreshed values for elements *Denise.xml* in RawPatientData and *2-aga.xml* in RawDNAData. **PExtract** and **RiskExtract** are then run on these refreshed elements. Suppose, as in Section 4.1.1, the refreshed value for DNARisks element #2 now has risk=0.6. After the unwinding recursion runs **Join** and **Filter** on refreshed elements, we finally get the refreshed value of HighPatientRisks element #2, with Denise's heart disease risk set to 0.6. □

In Section 4.2 we identified properties of transformations and provenance predicates required for the single-transformation refresh procedure to work correctly. It turns out that the same properties are not sufficient for the recursive algorithm to work correctly. As seen in Chapters 2 and 3, even when individual transformations have correct provenance, workflow provenance is not always guaranteed to be correct, weakly correct, or even to satisfy the even weaker replay property (Property 3.3.1). Furthermore, supporting selective refresh requires us to backward trace through old intermediate data. Thus, it is not surprising that we need to restrict the composition of transformations and their provenance predicates to ensure correct workflow refresh. We call this requirement *workflow safety*.

We first provide intuition for workflow safety, then formalize it. Consider $T_1 \circ T_2$ applied to input $I_1 \in \mathbb{I}_{T_1}$. Let $I_2 = T_1(I_1)$ and $O = T_2(I_2)$. Suppose I_1 has been modified to $I_1^{new} \in \mathbb{I}_{T_1}$. Let $I_2^{new} = T_1(I_1^{new})$, i.e., I_2^{new} is what would be produced by

running transformation T_1 on the entire new input set. Consider any element $\langle o, p\rangle$ in the original output set O. Safety requires that the set of elements in o's "new provenance," $\sigma_p(I_2^{new})$, is equal to the set obtained by refreshing each element in o's "old provenance," $\sigma_p(I_2)$.

Requirement 4.3.1 (Workflow Safety) Consider any workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$. Every T_i must be *safe with respect to* T_{i-1}, $i = 2..n$, defined as follows. Consider any $I'_{i-1} \in \mathbb{I}_{T_{i-1}}$. Let $I'_i = T_{i-1}(I'_{i-1})$. For any $\langle o, p\rangle \in T_i(I_i)$, we must have $\bigcup_{\langle o', p'\rangle \in \sigma_p(I_i)} T_{i-1}(\sigma_{p'}(I'_{i-1})) = \sigma_p(I'_i)$. □

If a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$ is safe, then it can be shown that its workflow provenance is correct. Since the definition of safety must be satisfied for any $I'_{i-1} \in \mathbb{I}_{T_{i-1}}$, while correctness is with respect to a fixed input, safety is a stronger property than correct workflow provenance.

With extensions to the safety definition to be introduced in Sections 4.4 and 4.5 for many-many and multi-input transformations, the workflow in our running example satisfies Requirement 4.3.1. However, there are some reasonable workflows where safety is not satisfied, as illustrated by the following example.

Example 4.3.2 Consider a workflow $T_1 \circ T_2$ that takes an input set I_1 with attributes salesperson, city, and sales_in_Euros. T_1 converts sales_in_Euros to sales_in_Dollars, with output provenance predicates selecting on salesperson. T_2 then sums sales_in_Dollars grouped by city, with output provenance predicates selecting on city.

Suppose the original input I_1 has two salespeople from Paris, Amelie and Jacques, selling 10 Euros each. The output I_2 of transformation T_1 contains (Amelie,Paris,13) and (Jacques,Paris,13) (at 2010 exchange rates), and the final output is (Paris,26). Now suppose I_1 is modified to I_1^{new}, with an additional salesperson (Marie,Paris,20). Safety requires equality of the following two procedures:

1. Compute the provenance of (Paris,26) in intermediate data set I_2, then refresh the resulting values. Predicate city='Paris' applied to I_2 yields (Amelie,Paris,13) and (Jacques,Paris,13); refreshing does not change their values.

2. Update the intermediate data set to $I_2^{new} = T_1(I^{new})$, then compute the provenance of (Paris,26) in I_2^{new}. I_2^{new} contains Marie as well as Amelie and Jacques, so applying predicate city='Paris' gives us a different result from case 1. □

Intuitively, a workflow is unsafe if, in some intermediate data set I, full forward propagation of modified input would cause "insertions" into a subset of I that comprises the provenance of a data element in the next data set. These insertions will be missed when we perform backward tracing, since we only refresh existing elements in intermediate data sets.

Provenance predicates for relational transformations typically yield safe workflows, but aggregation is an example of a transformation that can cause a workflow to be unsafe, if it is not the first transformation in a workflow, or if groups can grow as a result of input modifications. In Section 4.6 we discuss one way of handling unsafe workflows that allows us to retain some of the advantages of selective refresh without compromising correctness.

We now show that *workflow_refresh* correctly refreshes output elements, when Requirements 4.2.1, 4.2.2, and 4.3.1 all hold. The argument hinges on the following theorem.

Theorem 4.3.1 (Recursive Refresh Theorem) Consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$ satisfying Requirement 4.3.1. Given any element $\langle o, p\rangle \in I_{i+1}$ for $i \geq 1$, *workflow_refresh*$(\langle o, p\rangle) = T_i(\sigma_p(I_i^{new}))$.

Proof. We prove the theorem by induction on i. For the base case of $i = 1$, consider any $\langle o, p\rangle \in I_2 = T_1(I_1)$. By the first line (**if** case) of Algorithm 4.3.1, *workflow_refresh*$(\langle o, p\rangle \in I_2) = T_1(\sigma_p(I_1^{new}))$.

Now suppose the theorem holds for $i = k - 1$, $k > 1$; we show it holds for $i = k$. Consider element $\langle o, p\rangle \in I_{k+1}$. *workflow_refresh* computes the following two sets: $S = \sigma_p(I_k) = \{\langle o_1, p_1\rangle \ldots \langle o_m, p_m\rangle\}$ and $S' = \bigcup_{\langle o_i, p_i\rangle \in S}$ *workflow_refresh*$(\langle o_i, p_i\rangle)$. By the inductive hypothesis, each *workflow_refresh*$(\langle o_i, p_i\rangle) = T_{k-1}(\sigma_{p_i}(I_{k-1}^{new}))$. Thus, $S' = \bigcup_{\langle o_i, p_i\rangle \in \sigma_p(I_k)} T_{k-1}(\sigma_{p_i}(I_{k-1}^{new}))$. By Requirement 4.3.1, the right-hand side of the

last expression is equal to $\sigma_p(I_k^{new})$. Since the last line of *workflow_refresh* returns $T_k(S')$, *workflow_refresh* returns $T_k(\sigma_p(I_k^{new}))$, which completes the proof. □

To understand what the theorem is saying, consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$ and a final output element $\langle o, p\rangle \in I_{n+1}$. Suppose I_1 is updated to I_1^{new}. Theorem 4.3.1 says that running *workflow_refresh* on $\langle o, p\rangle$ is equivalent to computing I_n^{new} by pushing input I_1^{new} through every transformation except the last one, then running T_n on o's "new provenance," $\sigma_p(I_n^{new})$.

With this theorem, we see that the same arguments given in Section 4.2 for the correctness of single-transformation refresh carry over to the general workflow case: Running *workflow_refresh* logically reduces to running single-transformation refresh of T_n on the new input set I_n^{new}. Since T_n satisfies Requirements 4.2.1 and 4.2.2, the arguments in Section 4.2 show that *workflow_refresh* returns the correct refresh of an element: If it returns empty, there is no element in the new output set with provenance predicate p. If it returns an element $\langle o', p'\rangle$, then $p' = p$ and $\langle o', p\rangle$ is the unique element in I_{n+1}^{new} with predicate p.

4.4 Many-Many Transformations

So far we have required $T(\sigma_p(I)) = \{\langle o, p\rangle\}$ for any $\langle o, p\rangle \in O$ in any transformation instance $T(I) = O$ (Requirement 4.2.1). This requirement effectively limits us to transformations that are many-one or one-one. We now weaken this requirement, only insisting $\langle o, p\rangle \in T(\sigma_p(I))$. Let us see how weakening the requirement captures more transformations (specifically allowing one-many and many-many transformations) but complicates the picture.

Example 4.4.1 In our running example, **PExtract** is a one-many transformation. Consider refreshing (Earl,3,gcc) (element #6) in PatientDNA through transformation **PExtract**. Using Refresh Procedure 4.2.1 for single transformations, provenance predicate file='Earl.xml' is applied to RawPatientData, producing input element *Earl.xml*. Suppose when transformation **PExtract** is then run on *Earl.xml*, two elements are produced, (Earl,2,aaa) and (Earl,3,ttt) (indicating corrected DNA

sequences at locations 2 and 3), both with provenance predicate file='Earl.xml'. How do we know which of these elements, if any, corresponds to the one we are trying to refresh? □

To solve the problem illustrated in this example, we require that for many-many (and therefore one-many) transformations, output elements include not only provenance predicates, but also *forward filters*. The forward filter for an output element o is applied after forward propagating o's provenance, to select from multiple output elements the one corresponding to o. In Example 4.4.1, a suitable forward filter for output element #6 is loc=3, capturing the fact that element #6 describes Earl's DNA sequence at location 3. Note that all of the forward filters for table PatientDNA (denoted f in Figure 4.2) select on attribute loc, since locations are unique within each set of elements for a given name.

It is not hard to generalize our entire framework to support many-many transformations using forward filters. We require each transformation instance to produce triples instead of pairs: $T(I) = O = \{\langle o_1, p_1, f_1\rangle, \ldots, \langle o_n, p_n, f_n\rangle\}$. (By implicitly assuming all f_i=True for many-one transformations, our extension is fully "backward compatible" with everything in the chapter thus far.) To refresh an element $\langle o, p, f\rangle \in O$, we add a third step to Procedure 4.2.1 that applies forward filter σ_f to the result from Step 2, i.e., the overall refresh operation is $\sigma_f(T(\sigma_p(I^{new})))$.

All of the formalism and intuitive arguments in Sections 4.2 and 4.3 extend quite easily to incorporate forward filters, generally replacing $\langle o, p\rangle$ with $\langle o, p, f\rangle$ and $T(\sigma_p(I))$ with $\sigma_f(T(\sigma_p(I)))$. Note that in Requirement 4.2.2 (*Predicate as Key*), by extending each output pair to include a forward filter f, we are effectively treating provenance predicates and forward filters together as immutable keys, i.e., only the p-f pairs need be unique, not provenance predicates alone. We now provide details of the extension for many-many transformations. Section 4.4.1 describes the extension for one-transformation refresh, and Section 4.4.2 describes the extension for workflow refresh.

4.4.1 One-Transformation Refresh

Procedure 4.4.1 (Provenance-Based Refresh) Consider transformation instance $T(I) = O$, and suppose input data set $I \in \mathbb{I}_T$ has been modified to $I^{new} \in \mathbb{I}_T$. To refresh an output element $\langle o, p, f \rangle \in O$ there are three steps:

1. *Backward tracing:* Run tracing query σ_p on I^{new} to find the subset of I^{new} associated with provenance predicate p.

2. *Forward propagation:* Apply T on $\sigma_p(I^{new})$ to compute the refreshed elements associated with provenance predicate p.

3. *Forward filtering:* Apply σ_f on $T(\sigma_p(I^{new}))$ to find the new value $\langle o', p', f' \rangle$. If the result is empty, then o has no refreshed value. □

Requirement 4.4.1 (Predicate Correctness) Consider any transformation instance $T(I) = O$ for $I \in \mathbb{I}_T$, and any $\langle o, p, f \rangle \in O$. Then $\sigma_f(T(\sigma_p(I))) = \{\langle o, p, f \rangle\}$. □

Requirement 4.4.2 (Predicate as Key) Consider any transformation instance $T(I) = O$ for $I \in \mathbb{I}_T$, and any $\langle o, p, f \rangle \in O$. Then for any $I' \in \mathbb{I}_T$, if $\sigma_f(T(\sigma_p(I'))) \neq \emptyset$, then $\sigma_f(T(\sigma_p(I'))) = \{\langle o', p, f \rangle\}$ for some o', and $\langle o', p, f \rangle \in T(I')$. □

Property 4.4.1 (Unique Predicates) Consider any transformation instance $T(I) = O$ for $I \in \mathbb{I}_T$, and any $\langle o, p, f \rangle \in O$. There is no $\langle o', p, f \rangle \in T(I)$ with $o' \neq o$. □

We can show in a similar manner as we did at the end of Section 4.2 that Procedure 4.4.1 generates the correct refreshed value of $\langle o, p, f \rangle \in O$. Note that now p-f pairs are treated as our immutable key.

- First suppose $\sigma_f(T(\sigma_p(I^{new})))$ produces an empty result. Then there should be no $\langle o', p, f \rangle$ in $T(I^{new})$. If there were such an $\langle o', p, f \rangle$, then by Requirement 4.4.1, $\sigma_f(T(\sigma_p(I^{new}))) = \{\langle o', p, f \rangle\}$, contradicting that $\sigma_f(T(\sigma_p(I^{new})))$ is empty.

- Now suppose $\sigma_f(T(\sigma_p(I^{new})))$ is non-empty. Requirement 4.4.2 guarantees that $\sigma_f(T(\sigma_p(I^{new}))) = \{\langle o', p, f\rangle\}$ for some o', and that $\langle o', p, f\rangle$ is a valid element in $T(I^{new})$. Thus, o' is the refreshed value for o.

4.4.2 Workflow Refresh

Algorithm 4.4.1 (Workflow Refresh) Consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$. Suppose I_1 has been modified to I_1^{new}. Algorithm *workflow_refresh* refreshes output element $\langle o, p, f\rangle \in I_{i+1}$:

---

workflow_refresh$(\langle o, p, f\rangle \in I_{i+1})$:
  if $i = 1$ **then return** $\sigma_f(T_1(\sigma_p(I_1^{new})))$
  else { $S = \sigma_p(I_i)$;
    $S' = \bigcup_{\langle o', p', f'\rangle \in S}$ *workflow_refresh*$(\langle o', p', f'\rangle \in I_i)$;
    return $\sigma_f(T_i(S'))$ }

Requirement 4.4.3 (Workflow Safety) Consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$. Every T_i must be *safe with respect to* T_{i-1}, $i = 2..n$, defined as follows. Consider any $I'_{i-1} \in \mathbb{I}_{T_{i-1}}$. Let $I'_i = T_{i-1}(I'_{i-1})$. For any $\langle o, p, f\rangle \in T_i(I_i)$, we must have $\bigcup_{\langle o', p', f'\rangle \in \sigma_p(I_i)} \sigma_{f'}(T_{i-1}(\sigma_{p'}(I'_{i-1}))) = \sigma_p(I'_i)$. □

Theorem 4.4.1 (Recursive Refresh Theorem) Consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$ satisfying Requirement 4.4.3. Given any element $\langle o, p, f\rangle \in I_{i+1}$ for $i \geq 1$, *workflow_refresh*$(\langle o, p, f\rangle) = \sigma_f(T_i(\sigma_p(I_i^{new})))$.

Proof. We prove the theorem by induction on i. For the base case of $i = 1$, consider any $\langle o, p, f\rangle \in I_2 = T_1(I_1)$. By the first line (**if** case) of our algorithm, *workflow_refresh*$(\langle o, p, f\rangle \in I_2) = \sigma_f(T_1(\sigma_p(I_1^{new})))$, so the theorem holds for the base case.

Now suppose that the theorem holds for $i = k - 1$ where $k > 1$; we show it holds for $i = k$. Consider element $\langle o, p, f\rangle \in I_{k+1}$. *workflow_refresh*

computes the following two sets: $S = \sigma_p(I_k) = \{\langle o_1, p_1, f_1\rangle \ldots \langle o_m, p_m, f_m\rangle\}$ and $S' = \bigcup_{\langle o_i,p_i,f_i\rangle \in S} \textit{workflow\_refresh}(\langle o_i, p_i, f_i\rangle)$. By the inductive hypothesis, each $\textit{workflow\_refresh}(\langle o_i, p_i, f_i\rangle) = \sigma_{f_i}(T_{k-1}(\sigma_{p_i}(I_{k-1}^{new})))$. Thus, $S' = \bigcup_{\langle o_i,p_i,f_i\rangle \in \sigma_p(I_k)} \sigma_{f_i}(T_{k-1}(\sigma_{p_i}(I_{k-1}^{new})))$. By Requirement 4.4.3, the right-hand side of the last expression is equal to $\sigma_p(I_k^{new})$. Since the last line of *workflow_refresh* returns $\sigma_f(T_k(S'))$, *workflow_refresh* returns $\sigma_f(T_k(\sigma_p(I_k^{new})))$, which completes our proof. □

4.5 Multi-Input Transformations

For presentation purposes, so far we have assumed each transformation has one input data set. The extension for transformations with multiple input sets is straightforward and intuitive: Each output element carries a separate provenance predicate for each of its transformation's input sets. Now a transformation instance is $T(I_1, I_2, \ldots, I_m) = O$, and O consists of extended triples: $\{\langle o_1, (p_1^1, \ldots, p_1^m), f_1\rangle, \ldots, \langle o_n, (p_n^1, \ldots, p_n^m), f_n\rangle\}$. (We obtain full "backward compatibility" with everything in the chapter thus far by setting $m = 1$.) Refresh proceeds in a similar manner as before, except during backward tracing m provenance predicates are evaluated on their corresponding input data sets, and during forward propagation the transformation is run on the m input subsets.

Example 4.5.1 From our running example, to refresh (Carl,liver,0.4) (element #2) in PatientRisks, provenance predicate p^{PD} (name='Carl' ∧ loc=2 ∧ seq='ttt') is applied on PatientDNA to obtain element #2, and provenance predicate p^{DR} (loc=2 ∧ seq='ttt') is applied on DNARisks to obtain element #3. These elements are forward propagated as the two inputs to transformation **Join**, yielding the refreshed value of PatientRisks element #2. □

We can easily adapt all of the formalism from Sections 4.2–4.4 to handle multi-input transformations. In general we replace p with $p_1, \ldots, p_m$, and $T(\sigma_p(I))$ with $T(\sigma_{p_1}(I_1), \ldots, \sigma_{p_m}(I_m))$. In Requirement 4.2.2 (*Predicate as Key*), we now treat the entire combination of $p_1, \ldots, p_m, f$ as the immutable key. In Requirement 4.3.1

(*Workflow Safety*), we require each transformation to be safe with respect to all of its predecessor transformations in concert. We now provide details of the extension for multi-input transformations. Section 4.5.1 describes the extension for one-transformation refresh, and Section 4.5.2 describes the extension for workflow refresh.

4.5.1 One-Transformation Refresh

Procedure 4.5.1 (Provenance-Based Refresh) Consider transformation instance $T(I_1, \ldots, I_m) = O$, and suppose input data sets $(I_1, \ldots, I_m) \in \mathbb{I}_T$ have been modified to $(I_1^{new}, \ldots, I_m^{new}) \in \mathbb{I}_T$. To refresh an output element $\langle o, (p_1, \ldots, p_m), f\rangle \in O$ there are three steps:

1. *Backward tracing:* Run tracing queries σ_{p_i} on I_i^{new} to find the subsets of I_i^{new} associated with provenance predicates p_i, $i = 1..m$.

2. *Forward propagation:* Apply T on $(\sigma_{p_1}(I_1^{new}), \ldots, \sigma_{p_m}(I_m^{new}))$ to compute the refreshed elements associated with provenance predicates $p_1, \ldots, p_m$.

3. *Forward filtering:* Apply σ_f on $T(\sigma_{p_1}(I_1^{new}), \ldots, \sigma_{p_m}(I_m^{new}))$ to find the new value $\langle o', (p'_1, \ldots, p'_m), f'\rangle$. If the result is empty, then o has no refreshed value. □

Requirement 4.5.1 (Predicate Correctness) Consider any transformation instance $T(I_1, \ldots, I_m) = O$ and any $\langle o, (p_1, \ldots, p_m), f\rangle \in O$. Then $\sigma_f(T(\sigma_{p_1}(I_1), \ldots, \sigma_{p_m}(I_m))) = \{\langle o, (p_1, \ldots, p_m), f\rangle\}$. □

Requirement 4.5.2 (Predicate as Key) Consider any transformation instance $T(I_1, \ldots, I_m) = O$ and any $\langle o, (p_1, \ldots, p_m), f\rangle \in O$. Then for any $(I'_1, \ldots, I'_m) \in \mathbb{I}_T$, if $\sigma_f(T(\sigma_{p_1}(I'_1), \ldots, \sigma_{p_m}(I'_m))) \neq \emptyset$, then $\sigma_f(T(\sigma_{p_1}(I'_1), \ldots, \sigma_{p_m}(I'_m))) = \{\langle o', (p_1, \ldots, p_m), f\rangle\}$ for some o', and $\langle o', (p_1, \ldots, p_m), f\rangle \in T(\sigma_{p_1}(I'_1), \ldots, \sigma_{p_m}(I'_m))$. □

Property 4.5.1 (Unique Predicates) Consider any transformation instance $T(I_1, \ldots, I_m) = O$ and any $\langle o, (p_1, \ldots, p_m), f\rangle \in O$. There is no $\langle o', (p_1, \ldots, p_m), f\rangle \in O$ with $o' \neq o$. □

We can show in a similar manner as we did at the end of Section 4.2 that Procedure 4.5.1 generates the correct refreshed value of $\langle o, (p_1, \ldots, p_m), f\rangle \in O$. Note that now $\langle p_1, \ldots, p_m, f\rangle$ vectors are treated as our immutable key.

- First suppose $\sigma_f(T(\sigma_{p_1}(I_1^{new}), \ldots, \sigma_{p_m}(I_m^{new})))$ produces an empty result. Then there should be no $\langle o', (p_1, \ldots, p_m), f\rangle$ in $T(\sigma_{p_1}(I_1^{new}), \ldots, \sigma_{p_m}(I_m^{new}))$. If there were such an $\langle o', (p_1, \ldots, p_m), f\rangle$, then by Requirement 4.5.1, $\sigma_f(T(\sigma_{p_1}(I_1^{new}), \ldots, \sigma_{p_m}(I_m^{new}))) = \{\langle o', (p_1, \ldots, p_m), f\rangle\}$, contradicting the fact that $\sigma_f(T(\sigma_{p_1}(I_1^{new}), \ldots, \sigma_{p_m}(I_m^{new})))$ is empty.
- Now suppose $\sigma_f(T(\sigma_{p_1}(I_1^{new}), \ldots, \sigma_{p_m}(I_m^{new})))$ is non-empty. Requirement 4.5.2 guarantees that $\sigma_f(T(\sigma_{p_1}(I_1^{new}), \ldots, \sigma_{p_m}(I_m^{new}))) = \{\langle o', (p_1, \ldots, p_m), f\rangle\}$ for some o', and that $\langle o', (p_1, \ldots, p_m), f\rangle$ is a valid element in $T(I_1^{new}, \ldots, I_m^{new})$. Thus, o' is the refreshed value for o.

4.5.2 Workflow Refresh

Algorithm 4.5.1 (Workflow Refresh) Consider a workflow composed of an acyclic graph of transformations $T_1, \ldots, T_n$ and input data sets $I_1, \ldots, I_k$. Let $O_i = T_i(I_1^i, \ldots, I_{m_i}^i)$ for $i = 1..n$; the final output data set is O_n. Suppose the input sets $I_1, \ldots, I_k$ have been modified to $I_1^{new}, \ldots, I_k^{new}$. Let $\bar{p}$ be shorthand for $p_1..p_m$; its use is clear in context. Algorithm *workflow_refresh* refreshes output element $\langle o, (p^1, \ldots, p^{m_i}), f\rangle \in O_i$:

---

workflow_refresh$(\langle o, (p^1, \ldots, p^{m_i}), f\rangle \in O_i)$:
  for $j = 1..m_i$:
    if I_j^i is an input data set **then** $S'_j = \sigma_{p^j}(I_j^{i\ new})$
    else { $S_j = \sigma_{p^j}(I_j^i)$;

$$S'_j = \bigcup_{\langle o', \bar{p}', f' \rangle \in S_j} \textit{workflow\_refresh}(\langle o', \bar{p}', f' \rangle)\}$$

return $\sigma_f(T_i(S'_1, \ldots, S'_{m_i}))$ }

Requirement 4.5.3 (Workflow Safety) Consider a workflow composed of an acyclic graph of transformations $T_1, \ldots, T_n$ and input data sets $I_1, \ldots, I_k$. Let $O_i = T_i(I^i_1, \ldots, I^i_{m_i})$ for $i = 1..n$; the final output data set is O_n. Assume the output of each transformation matches the input domain of the next transformation. Every T_i must be *safe with respect to each of its input transformations*, defined as follows. Let T_j be an input transformation of T_i. Consider any $(I'^j_1, \ldots, I'^j_{m_j}) \in \mathbb{I}_{T_j}$. Let $O'_j = T_j(I'^j_1, \ldots, I'^j_{m_j})$. For any $\langle o, (p_1, \ldots, p_{m_i}), f \rangle \in O_i$, let p^j be the predicate corresponding to O_j. We must have $\bigcup_{\langle o', \bar{p}', f' \rangle \in \sigma_{p^j}(O_j)} \sigma_{f'}(T_j(\sigma_{p'_1}(I'^j_1), \ldots, \sigma_{p'_{m_j}}(I'^j_{m_j}))) =$ $\sigma_{p^j}(O'_j)$. □

Theorem 4.5.1 (Recursive Refresh Theorem) Consider a workflow composed of an acyclic graph of transformations $T_1, \ldots, T_n$ and input data sets $I_1, \ldots, I_k$, satisfying Requirement 4.5.3. Given any element $\langle o, (p_1, \ldots, p_{m_i}), f \rangle \in O_i$ for $i \geq 1$, $\textit{workflow\_refresh}(\langle o, (p_1, \ldots, p_{m_i}), f \rangle) = \sigma_f(T_i(\sigma_{p_1}(I^{i\ new}_1), \ldots, \sigma_{p_{m_i}}(I^{i\ new}_{m_i})))$.

Proof. Given i, let $l(i)$ be the length of the longest path from O_i to input data: if T_i has input data sets as all inputs, then $l(i) = 1$. We prove the theorem by induction on $l(i)$. For the base case of $l(i) = 1$, consider any $\langle o, (p_1, \ldots, p_{m_i}), f \rangle \in O_i$. Using *workflow_refresh*, we see for all j that since I^i_j are input data sets, $S'_j = \sigma_{p^j}(I^{i\ new}_j)$. We return $\sigma_f(T_i(S'_1, \ldots, S'_{m_i})) = \sigma_f(T_i(\sigma_{p_1}(I^{i\ new}_1), \ldots, \sigma_{p_{m_i}}(I^{i\ new}_{m_i})))$, so the theorem holds for the base case.

Now suppose that the theorem holds for all i such that $1 \leq l(i) \leq d-1$ where $d > 1$; we show it holds for all i such that $l(i) = d$. Consider i such that $l(i) = d$.

Consider element $\langle o, (p_1, \ldots, p_{m_i}), f \rangle \in O_i$. For each j in $1..m_i$, if I^i_j is a input data set, $S'_j = \sigma_{p_j}(I^{i\ new}_j)$. Else, *workflow_refresh* computes the following two sets: $S_j = \sigma_{p^j}(I^i_j)$ and $S'_j = \bigcup_{\langle o', \bar{p}', f' \rangle \in S_j} \textit{workflow\_refresh}(\langle o', \bar{p}', f' \rangle)$. Since for all

input transformations T_t to T_i we have $1 \leq l(t) \leq d-1$, by the inductive hypothesis, each *workflow_refresh*$(\langle o, (p_1, \ldots, p_{m_t}), f\rangle) = \sigma_f(T_t(\sigma_{p_1}(I_1^{t\ new}), \ldots, \sigma_{p_{m_t}}(I_{m_t}^{t\ new})))$. Thus, since $S'_j = \bigcup_{\langle o', \bar{p}', f'\rangle \in \sigma_{p^j}(I_j^i)}$ *workflow_refresh*$(\langle o', \bar{p}', f'\rangle) = \bigcup_{\langle o', \bar{p}', f'\rangle \in \sigma_{p^j}(I_j^i)} \sigma_{f'}(T_j(\sigma_{p'_1}(I_1^{j\ new}), \ldots, \sigma_{p'_{m_j}}(I_{m_j}^{j\ new})))$, then by Requirement 4.5.3, each $S'_j = \sigma_{p_j}(I_j^{i\ new})$.

Since the last line of *workflow_refresh* returns $\sigma_f(T_i(S'_1, \ldots, S'_{m_i}))$, *workflow_refresh* returns $\sigma_f(T_i(\sigma_{p_1}(I_1^{i\ new}), \ldots, \sigma_{p_{m_i}}(I_{m_i}^{i\ new})))$, which completes our proof. □

4.6 Unsafe Workflows

In Section 4.3 we introduced *safe workflows* (Requirement 4.3.1), and our refresh algorithms thus far require workflow safety. We suggest one simple mechanism that allows refresh in unsafe workflows to still make some use of our algorithms.

Consider a workflow instance $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$. (Our argument generalizes easily to workflows with multi-input transformations.) Suppose the workflow is unsafe at T_k for some k, $1 < k \leq n$, but the rest of the workflow is safe. (More generally, consider the largest k such that the workflow is unsafe at T_k.) Suppose I_1 has been modified to I_1^{new}. To support correct refresh of output elements in I_{n+1}, we can first bring intermediate data set I_k up-to-date by sending the entire modified input I_1^{new} through all transformations up to T_{k-1}, producing I_k^{new}. Then we can treat I_k^{new} as if it were the first input data set (and T_k the first transformation), performing refresh as normal: backward-trace from I_{n+1} to I_k, then forward propagate through to I_{n+1}.

In this setting, we would certainly want to keep track of when I_k^{new} is up-to-date, perhaps even propagating input modifications through the first $k-1$ transformations eagerly. Combining eager propagation with on-demand refresh is an interesting direction of future work.

4.7 Related Work

None of the many papers on lineage and provenance discussed in Section 1.4 exploit provenance for selective refresh in a general workflow environment. There has been a large body of work in *incremental view maintenance*: the efficient propagation of base data modifications to bring materialized views up to date, usually in a relational setting [12, 28]. We consider general workflows rather than relational views. Also, in contrast to the view-maintenance problem, selective refresh considers efficiently computing the up-to-date value of individual output elements, not the entire output. Reference [26] considers the problem of "update exchange" between data peers linked by mappings. A subproblem they address is determining when a derived data element is no longer valid, but they do not provide a means to selectively refresh out-of-date values. Also, transformations in [26] are restricted to those that can be expressed in Datalog.

4.8 Conclusions

We presented a formal foundation and algorithms for efficient selective refresh of output elements in data-oriented workflows. We identified properties of transformations, provenance, and workflows that are required for the algorithms to perform refresh correctly, and we discussed how the algorithms can be adapted to handle unsafe workflows.

Chapter 5

Logical Provenance

5.1 Introduction

Recall from Chapter 1 that most approaches to provenance in workflows either track *coarse-grained* (schema and/or transformation level) provenance 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.

In this chapter, we introduce *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 chapter are as follows:

- **Logical provenance specifications for transformations** (Section 5.2). We

describe a simple logical-provenance specification language consisting of *attribute mappings* and *filters*.

- **Algorithms for provenance tracing** (Section 5.3). We provide algorithms for provenance tracing in workflows where logical provenance for each transformation is specified using our language. In our algorithms we perform provenance tracing at the schema level to the extent possible, although eventually accessing the data obviously is required.

- **Logical provenance for relational transformations** (Section 5.4). We consider logical provenance in the relational setting, showing that for a class of *Select-Project-Join* (*SPJ*) *transformations*, logical provenance specifications encode minimal provenance.

In Section 5.5 we discuss related work, and we conclude in Section 5.6.

5.2 Logical Provenance Specifications

This chapter builds on the foundations of provenance presented in Chapter 2. Recall we discussed the desirable formal properties of provenance in Section 2.2, and we explored how these properties carry over from individual transformations to a workflow in Section 2.3. In this chapter we start by discussing how to logically specify provenance relationships between the input elements and output elements of a given transformation. To design such a specification language, 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 5.2.1 (Attribute Mapping) Let T be a transformation with input sets $I_1, \ldots, I_m$ and output sets $O_1, \ldots, O_r$. Let A be an attribute of input set I_i, and let B_j be an attribute of output set O_j. Given any value $x \in domain(B_j)$ and any input set instances $I'_1, \ldots, I'_m$, let $\sigma_{B_j=x}(T(I'_1, \ldots, I'_m))$ be shorthand for $\sigma_{B_j=x}(O'_j)$ where $(O'_1, \ldots, O'_r) = T(I'_1, \ldots, I'_m)$. T has an *attribute mapping* between input attribute $I_i.A$ and output attribute $O_j.B_j$, denoted $I_i.A \leftrightarrow O_j.B_j$, if for all possible values of $x \in domain(B_j)$ and all possible input set instances $I'_1, \ldots, I'_m$:

$$\sigma_{B_j=x}(T(I'_1, \ldots, I'_m)) = \sigma_{B_j=x}(T(I'_1, \ldots, \sigma_{A=x}(I'_i), \ldots, I'_m)) \quad \square$$

In words, attribute mapping $I_i.A \leftrightarrow O_j.B_j$ states that the output subset of O_j where $B_j = x$ is unaffected by all elements in I_i except those where $A = x$.

Example 5.2.1 We revisit our running example workflow and data shown in Figures 1.1 and 1.2 of Section 1.1. Consider transformation **JoinAgg** from the example, 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. 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. $\square$

Definition 5.2.2 (Filter) Let T be a transformation with input sets $I_1, \ldots, I_m$ and output sets $O_1, \ldots, O_r$. T has a *filter condition* C on input I_i if for all possible input set instances $I'_1, \ldots, I'_m$:

$$T(I'_1, \ldots, I'_m) = T(I'_1, \ldots, \sigma_C(I'_i), \ldots, I'_m) \quad \square$$

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

Example 5.2.2 Consider transformation **Filter** from the running example (Figure 1.1), 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 prove its correctness.

Definition 5.2.3 (Provenance Encoded by Logical Specification) Consider a transformation instance $(O_1, \ldots, O_r) = T(I_1, \ldots, I_m)$ 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). Let $M_{i,j}$ denote the subset of attribute mappings $A \leftrightarrow B$ in M such that attribute A is from input I_i and attribute B is from output O_j. Let F_i denote the subset of filters in F that are on I_i. Consider output element $o \in O_j$ for some j. The provenance of o as encoded by (M, F) is $\langle I_1^*, \ldots, I_m^* \rangle = \langle \sigma_{C_1}(I_1), \ldots, \sigma_{C_m}(I_m) \rangle$ where each $C_i = (\bigwedge_{(A \leftrightarrow B) \in M_{i,j}} A = o.B) \wedge (\bigwedge_{C \in F_i} C)$. □

Example 5.2.3 Consider again transformation **Filter** from the running example (Figure 1.1), 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↔LP.item-id, IC.country ↔ LP.country, IC.brand ↔ LP.brand$\}$ and $F = \{$type='laptop'$\}$. Referring to Figure 1.2, if $o =$ LP(1), the first tuple in LP, then o's provenance as encoded by (M, F) is $\sigma_{\text{item-id='I1'} \wedge \text{country='France'} \wedge \text{brand='HP'} \wedge \text{type='laptop'}}(\text{IC}) = \{\text{IC}(1)\}$. □

We now prove that provenance encoded by logical specifications is indeed correct, according to Definition 2.2.4 of provenance correctness.

Theorem 5.2.1 Consider a transformation instance $(O_1, \ldots, O_r) = T(I_1, \ldots, I_m)$ with logical provenance specification (M, F). Given any output element $o \in O_j$ for some j, o's provenance $\langle I_1^*, \ldots I_m^* \rangle = \langle \sigma_{C_1}(I_1), \ldots, \sigma_{C_m}(I_m) \rangle$ as specified in Definition 5.2.3 is correct for o with respect to T according to Definition 2.2.4.

Proof. Consider any output element $o \in O_j$ for some j. $\langle I_1^*, \ldots, I_m^* \rangle = \langle \sigma_{C_1}(I_1), \ldots, \sigma_{C_m}(I_m) \rangle$ where each C_i is a conjunction of predicates as in Definition 5.2.3. Consider any $\langle I_1', \ldots, I_m' \rangle$ such that $I_1' \subseteq I_1, \ldots, I_m' \subseteq I_m$. We need to show that $o \in T(I_1', \ldots, I_m')$ if and only if $o \in T(I_1' \cap I_1^*, \ldots, I_m' \cap I_m^*)$.

First suppose $o \in T(I'_1, \ldots, I'_m)$. Then $o \in \sigma_{\mathbf{B}=o.\mathbf{B}}(T(I'_1, \ldots, I'_m))$, where $\mathbf{B}$ contains all attributes in o. Since M and F hold for T, $\sigma_{\mathbf{B}=o.\mathbf{B}}(T(I'_1, \ldots, I'_m)) = \sigma_{\mathbf{B}=o.\mathbf{B}}(T(\sigma_{C_1}(I'_1), \ldots, \sigma_{C_m}(I'_m)))$, from which it follows that $o \in \sigma_{\mathbf{B}=o.\mathbf{B}}(T(\sigma_{C_1}(I'_1), \ldots, \sigma_{C_m}(I'_m)))$. Since $T(\sigma_{C_1}(I'_1), \ldots, \sigma_{C_m}(I'_m)) = T(I'_1 \cap I^*_1, \ldots, I'_m \cap I^*_m)$, $o \in \sigma_{\mathbf{B}=o.\mathbf{B}}(T(I'_1 \cap I^*_1, \ldots, I'_m \cap I^*_m))$, implying $o \in T(I'_1 \cap I^*_1, \ldots, I'_m \cap I^*_m)$.

Now suppose $o \in T(I'_1 \cap I^*_1, \ldots, I'_m \cap I^*_m)$. Since $T(I'_1 \cap I^*_1, \ldots, I'_m \cap I^*_m) = T(\sigma_{C_1}(I'_1), \ldots, \sigma_{C_m}(I'_m))$, $o \in T(\sigma_{C_1}(I'_1), \ldots, \sigma_{C_m}(I'_m))$. Then $o \in \sigma_{\mathbf{B}=o.\mathbf{B}}(T(\sigma_{C_1}(I'_1), \ldots, \sigma_{C_m}(I'_m)))$. Since M and F hold for T, $\sigma_{\mathbf{B}=o.\mathbf{B}}(T(\sigma_{C_1}(I'_1), \ldots, \sigma_{C_m}(I'_m))) = \sigma_{\mathbf{B}=o.\mathbf{B}}(T(I'_1, \ldots, I'_m))$, from which it follows that $o \in \sigma_{\mathbf{B}=o.\mathbf{B}}(T(\sigma_{C_1}(I'_1), \ldots, \sigma_{C_m}(I'_m)))$. Thus, $o \in \sigma_{\mathbf{B}=o.\mathbf{B}}(T(I'_1, \ldots, I'_m))$, implying $o \in T(I'_1, \ldots, I'_m)$. □

5.2.1 Attribute Mappings with Functions

We now show how attribute mappings can be extended beyond simple equalities to involve functions.

Definition 5.2.4 (Attribute Mapping with Functions) Let T be a transformation with input sets $I_1, \ldots, I_m$ and output sets $O_1, \ldots, O_r$. Let $\mathbf{A}$ be attributes of input set I_i, and let $\mathbf{B}$ be attributes of output set O_j. Let f be a function whose domain includes all possible values of $\mathbf{A}$, and let g be a function whose domain includes all possible values of $\mathbf{B}$. T has an attribute mapping between $f(I_i.\mathbf{A})$ and $g(O_j.\mathbf{B})$, denoted $f(I_i.\mathbf{A}) \leftrightarrow g(O_j.\mathbf{B})$, if for all possible values of $g(\mathbf{B})$ and all possible input set instances $I'_1, \ldots, I'_m$:

$\sigma_{g(\mathbf{B})=x}(T(I'_1, \ldots, I'_m)) = \sigma_{g(\mathbf{B})=x}(T(I'_1, \ldots, \sigma_{f(\mathbf{A})=x}(I'_i), \ldots, I'_m))$. □

Example 5.2.4 Consider a transformation **NameCombine** that takes input set Emp(first-name, last-name, addr) and creates output set Person(name, addr) by concatenating first-name and last-name for each record in Emp. Let $f(a, b)$ be a function that concatenates a and b. Attribute mapping f(Emp.first-name, Emp.last-name) $\leftrightarrow$

Person.name holds for this transformation, indicating that the subset of output elements in Person with particular name values corresponds to the input subset of Emp for which concatenating first-name and last-name produces that name. □

In the remainder of this chapter, we will only consider mappings that involve equalities. Much of the formalism that follows can be extended to handle functions without introducing any new complexities.

5.2.2 Multi-Attribute Mappings

Given attribute mappings involving single input and output attributes, we can combine them to create attribute mappings across sets of attributes. The definition below considers a single output set. (We explain later why multiple output sets are not considered.)

Definition 5.2.5 (Multi-Attribute Mapping) Let T be a transformation with input sets $I_1, \ldots, I_m$ and output sets $O_1, \ldots, O_r$. 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 $\mathbf{I} = (I_1, \ldots, I_m)$ and a single output set O_j respectively. Let $\sigma_{\mathbf{A}=\mathbf{x}}(I_1, \ldots, I_m)$ denote $(\sigma_{p_1}(I_1), \ldots, \sigma_{p_m}(I_m))$ where p_i is the conjunction of all terms in the set $\{A_1 = x_1, \ldots, A_n = x_n\}$ such that A_k is an attribute in I_i. Given any $\mathbf{x} \in \mathit{domain}(\mathbf{B})$ and any input set instances $I'_1, \ldots, I'_m$, let $\sigma_{\mathbf{B}=\mathbf{x}}(T(I'_1, \ldots, I'_m))$ be shorthand for $\sigma_{B_1=x_1 \wedge \ldots \wedge B_n=x_n}(O'_j)$ where $(O'_1, \ldots, O'_r) = T(I'_1, \ldots, I'_m)$. We say that attribute mapping $\mathbf{I}.\mathbf{A} \leftrightarrow O_j.\mathbf{B}$ *holds* iff $\forall \mathbf{I}, \mathbf{x} \in \mathit{domain}(\mathbf{B})$, we have $\sigma_{\mathbf{B}=\mathbf{x}}(T(I_1, \ldots, I_m)) = \sigma_{\mathbf{B}=\mathbf{x}}(T(\sigma_{\mathbf{A}=\mathbf{x}}(I_1, \ldots, I_m)))$.

Example 5.2.5 Consider again transformation **JoinAgg** from the running example (Figure 1.1). 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 5.2.2 (Combining Attribute Mappings) Let T be a transformation with input sets $I_1, \ldots, I_m$ and output sets $O_1, \ldots, O_r$. 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 $\mathbf{I} = (I_1, \ldots, I_m)$ and a single output set O_j respectively. Attribute mappings $A_1 \leftrightarrow B_1, \ldots, A_n \leftrightarrow B_n$ hold if and only if $\mathbf{A} \leftrightarrow \mathbf{B}$ holds.

Proof. We prove by induction on the number of attributes. For $n = 1$, the theorem follows from the assumption. Now suppose that the theorem holds for $n = k$ and consider $n = k + 1$.

Suppose $A_1 \leftrightarrow B_1, \ldots, A_{k+1} \leftrightarrow B_{k+1}$ all hold. Since the theorem holds for $n = k$, we know that $(A_1, A_2, \ldots, A_k) \leftrightarrow (B_1, B_2, \ldots, B_k)$ holds, from which it follows that $\sigma_{\mathbf{B_k}=\mathbf{x_k}}(T(I_1, \ldots, I_m)) = \sigma_{\mathbf{B_k}=\mathbf{x_k}}(T(\sigma_{\mathbf{A_k}=\mathbf{x_k}}(I_1, \ldots, I_m)))$ for all possible values of $\mathbf{x_k}$. We can then use attribute mapping $A_{k+1} \leftrightarrow B_{k+1}$ to deduce that for all possible values of $\mathbf{x_{k+1}}$, we have that

$\sigma_{\mathbf{B_{k+1}}=\mathbf{x_{k+1}}}(T(I_1, \ldots, I_m))$

$= \sigma_{(\mathbf{B_k}=\mathbf{x_k})\wedge(B_{k+1}=x_{k+1})}(T(I_1, \ldots, I_m))$

$= \sigma_{\mathbf{B_k}=\mathbf{x_k}}(\sigma_{B_{k+1}=x_{k+1}}(T(I_1, \ldots, I_m)))$

$= \sigma_{\mathbf{B_k}=\mathbf{x_k}}(\sigma_{B_{k+1}=x_{k+1}}(T(\sigma_{A_{k+1}=x_{k+1}}(I_1, \ldots, I_m))))$ (via $A_{k+1} \leftrightarrow B_{k+1}$)

$= \sigma_{B_{k+1}=x_{k+1}}(\sigma_{\mathbf{B_k}=\mathbf{x_k}}(T(\sigma_{A_{k+1}=x_{k+1}}(I_1, \ldots, I_m))))$

$= \sigma_{B_{k+1}=x_{k+1}}(\sigma_{\mathbf{B_k}=\mathbf{x_k}}(T(\sigma_{\mathbf{A_k}=\mathbf{x_k}}(\sigma_{A_{k+1}=x_{k+1}}(I_1, \ldots, I_m)))))$ (via assumption that theorem holds for $n = k$)

$= \sigma_{\mathbf{B_{k+1}}=\mathbf{x_{k+1}}}(T(\sigma_{\mathbf{A_{k+1}}=\mathbf{x_{k+1}}}(I_1, \ldots, I_m)))$.

Thus, if $A_1 \leftrightarrow B_1, \ldots, A_{k+1} \leftrightarrow B_{k+1}$, then $(A_1, A_2, \ldots, A_{k+1}) \leftrightarrow (B_1, B_2, \ldots, B_{k+1})$.

Now suppose $(A_1, A_2, \ldots, A_{k+1}) \leftrightarrow (B_1, B_2, \ldots, B_{k+1})$. We need to show that $A_j \leftrightarrow B_j$ holds for all j. Without loss of generality, we will set $j = k + 1$ to simplify notation. Suppose $A_{k+1} \leftrightarrow B_{k+1}$ does not hold. Then there exists x_{k+1}, $I'_1, \ldots, I'_m$ such that:

$\sigma_{B_{k+1}=x_{k+1}}(T(I'_1, \ldots, I'_m)) \neq \sigma_{B_{k+1}=x_{k+1}}(T(\sigma_{A_{k+1}=x_{k+1}}(I'_1, \ldots, I'_m)))$

Given this inequality, there exists at least one element e that is in the symmetric difference of the two sides. Let $\mathbf{x_k}$ be the value of $e.\mathbf{B_k}$. Then we have:

$\sigma_{(\mathbf{B_k}=\mathbf{x_k})\wedge(B_{k+1}=x_{k+1})}(T(I'_1, \ldots, I'_m)) \neq \sigma_{(\mathbf{B_k}=\mathbf{x_k})\wedge(B_{k+1}=x_{k+1})}(T(\sigma_{A_{k+1}=x_{k+1}}(I'_1, \ldots, I'_m)))$

$\sigma_{\mathbf{B_{k+1}}=\mathbf{x_{k+1}}}(T(I'_1, \ldots, I'_m)) \neq \sigma_{\mathbf{B_{k+1}}=\mathbf{x_{k+1}}}(T(\sigma_{A_{k+1}=x_{k+1}}(I'_1, \ldots, I'_m)))$

By the definition of $(A_1, A_2, \ldots, A_{k+1}) \leftrightarrow (B_1, B_2, \ldots, B_{k+1})$ holding for T, we know that for all $I_1, \ldots, I_m$:

$\sigma_{\mathbf{B_{k+1}=x_{k+1}}}(T(I_1, \ldots, I_m)) = \sigma_{\mathbf{B_{k+1}=x_{k+1}}}(T(\sigma_{\mathbf{A_{k+1}=x_{k+1}}}(I_1, \ldots, I_m)))$

Setting $(I_1, \ldots, I_m)$ equal to $(I'_1, \ldots, I'_m)$ we get:

$\sigma_{\mathbf{B_{k+1}=x_{k+1}}}(T(I'_1, \ldots, I'_m)) = \sigma_{\mathbf{B_{k+1}=x_{k+1}}}(T(\sigma_{\mathbf{A_{k+1}=x_{k+1}}}(I'_1, \ldots, I'_m)))$

Setting $(I_1, \ldots, I_m)$ equal to $\sigma_{A_{k+1}=x_{k+1}}(I'_1, \ldots, I'_m)$ we get:

$\sigma_{\mathbf{B_{k+1}=x_{k+1}}}(T(\sigma_{A_{k+1}=x_{k+1}}(I'_1, \ldots, I'_m))) = \sigma_{\mathbf{B_{k+1}=x_{k+1}}}(T(\sigma_{\mathbf{A_{k+1}=x_{k+1}}}(I'_1, \ldots, I'_m)))$

But then we have $\sigma_{\mathbf{B_{k+1}=x_{k+1}}}(T(I'_1, \ldots, I'_m)) = \sigma_{\mathbf{B_{k+1}=x_{k+1}}}(T(\sigma_{A_{k+1}=x_{k+1}}(I'_1, \ldots, I'_m)))$, a contradiction. Thus, $A_{k+1} \leftrightarrow B_{k+1}$ holds. □

In Definition 5.2.5 and Theorem 5.2.2 we have not considered multi-attribute mappings involving multiple output sets. While we could extend Definition 5.2.5 in a natural way to allow multiple output sets, Theorem 5.2.2 would not hold, as demonstrated by the following example.

Example 5.2.6 Consider transformation T with input data sets $I_1(A) = \{(1), (2), (3)\}$ and $I_2(B) = \{(2), (3), (4)\}$, and output data sets $O_1(C) = \{(2), (3)\}$ and $O_2(D) = \{(2), (3)\}$. Both O_1 and O_2 are computed by taking the intersection of the values in I_1 and I_2. Attribute mappings $I_1.A \leftrightarrow O_1.C$ and $I_2.B \leftrightarrow O_2.D$ both hold individually. However, consider multi-attribute mapping $(I_1.A, I_2.B) \leftrightarrow (O_1.C, O_2.D)$ defined as follows: For any $x \in \mathit{domain}(C)$, $y \in \mathit{domain}(D)$ and any input set instances I'_1, I'_2, we have $\sigma_{C=x}(T(I'_1, I'_2)) = \sigma_{C=x}(T(\sigma_{A=x}(I'_1), \sigma_{B=y}(I'_2)))$ and $\sigma_{D=y}(T(I'_1, I'_2)) = \sigma_{D=y}(T(\sigma_{A=x}(I'_1), \sigma_{B=y}(I'_2)))$. Recall $(O_1, O_2) = T(I_1, I_2)$. Let $(O'_1, O'_2) = T(\sigma_{A=2}(I_1), \sigma_{B=3}(I_2))$. Note that $O'_1 = \emptyset$ and $O'_2 = \emptyset$. Mapping $(I_1.A, I_2.B) \leftrightarrow (O_1.C, O_2.D)$ implies that $\sigma_{C=2}(O_1) = \sigma_{C=2}(O'_1)$ and $\sigma_{D=3}(O_2) = \sigma_{D=3}(O'_2)$. However, $\sigma_{C=2}(O_1) = \{(2)\} \neq \emptyset = \sigma_{C=2}(O'_1)$ and $\sigma_{D=3}(O_2) = \{(3)\} \neq \emptyset = \sigma_{D=3}(O'_2)$, and thus $(I_1.A, I_2.B) \leftrightarrow (O_1.C, O_2.D)$ does not hold. □

In the remainder of the chapter we consider single-attribute mappings only, since multi-attribute mappings generally don't add useful expressive power.

5.3 Provenance Tracing in Workflows

Consider a workflow in which each transformation has a logical provenance specification in the language described in Section 5.2. 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 as efficient as possible.

As we saw in Section 2.3, even if we have minimal provenance for each transformation, workflow provenance isn't always minimal, or even correct. Using the theoretical results of Section 2.3, 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 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 5.3.1, we specify when and how logical provenance can be combined across transformations without losing correctness. In Section 5.3.2, 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 5.3.3, we present our overall algorithm for provenance tracing in workflows with logical provenance specifications given for each transformation.

5.3.1 Combining Logical Provenance Across Transformations

Let T_1 and T_2 be transformations such that T_2 takes as input one of T_1's output data sets. Given an element o from an output set I_3 of T_2, suppose we wanted to trace o's

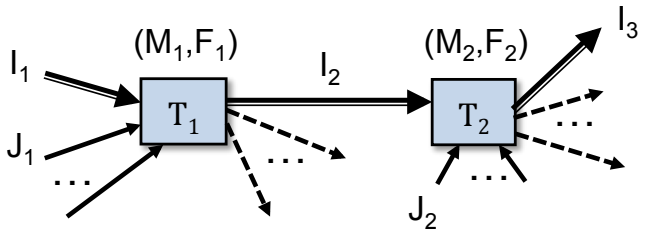

Figure 5.1: Abstract workflow $W = T_1 \circ T_2$.

provenance to all input sets of T_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.

Consider Figure 5.1. Let I_2 be the output set of T_1 that is passed to T_2. Without loss of generality, we assume T_1 has only one input set in addition to I_1, which we call J_1. Similarly T_2 has input data sets I_2 and J_2. Throughout this section, we use $W = T_1 \circ T_2$ to denote the workflow composed of T_1 and T_2, with input sets I_1, J_1, J_2. $T_1(I_1, J_1)$ may output multiple sets; as convention we use $T_1(I_1, J_1)_2$ to denote the output set of $T_1(I_1, J_1)$ that is passed to T_2. Let M_1 denote the set of attribute mappings for T_1 between I_1 and I_2, and let F_1 denote the set of filters for T_1 on I_1. Similarly let M_2 denote the set of attribute mappings for T_2 between I_2 and I_3, and let F_2 denote the set of filters for T_2 on I_2. We first show how attribute mappings can be combined across transformations.

Theorem 5.3.1 (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.

Proof 5.3.1 Since $I_1.A \leftrightarrow I_2.B$ holds for T_1, we know that for all possible values of x and I_1', J_1', $\sigma_{B=x}(T_1(I_1', J_1')) = \sigma_{B=x}(T_1(\sigma_{A=x}(I_1'), J_1')))$. Since $I_2.B \leftrightarrow I_3.C$ holds for T_2, we know that for all x and I_2', J_2', $\sigma_{C=x}(T_2(I_2', J_2')) = \sigma_{C=x}(T_2(\sigma_{B=x}(I_2'), J_2'))$. Thus, we know that for all x and I_1', J_1', J_2':

$\sigma_{C=x}((T_1 \circ T_2)(I_1', J_1', J_2'))$

$= \sigma_{C=x}(T_2(T_1(I_1', J_1')_2, J_2'))$

$= \sigma_{C=x}(T_2(\sigma_{B=x}(T_1(I_1', J_1')), J_2'))$ (since $I_2.B \leftrightarrow I_3.C$ holds for T_2)

$= \sigma_{C=x}(T_2(\sigma_{B=x}(T_1(\sigma_{A=x}(I_1'), J_1')), J_2'))$ (since $I_1.A \leftrightarrow I_2.B$ holds for T_1)
$= \sigma_{C=x}(T_2(T_1(\sigma_{A=x}(I_1'), J_1')_2, J_2'))$ (since $I_2.B \leftrightarrow I_3.C$ holds for T_2)
$= \sigma_{C=x}((T_1 \circ T_2)(\sigma_{A=x}(I_1'), J_1', J_2'))$
proving that $I_1.A \leftrightarrow I_3.C$ holds for $W = T_1 \circ T_2$. □

Example 5.3.1 Consider transformations **JoinAgg** and **Filter** from the running example (Figure 1.1). Since attribute mapping IP.brand ↔ IC.brand holds for **JoinAgg**, and attribute mapping IC.brand ↔ LP.brand holds for **Filter**, then by Theorem 5.3.1 attribute mapping IP.brand ↔ LP.brand holds for **JoinAgg** ∘ **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 5.3.2 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.

Proof 5.3.2 Since T_1 has filter condition $C \in F_1$, we know that for all possible values of I_1', J_1', $T_1(I_1', J_1') = T_1(\sigma_C(I_1'), J_1')$. Thus, we know that for all I_1', J_1', J_2':
$(T_1 \circ T_2)(I_1', J_1', J_2')$
$= T_2(T_1(I_1', J_1')_2, J_2')$
$= T_2(T_1(\sigma_C(I_1'), J_1')_2, J_2')$
$= (T_1 \circ T_2)(\sigma_C(I_1'), J_1', J_2')$
proving that $C \in F_W$ holds for $W = T_1 \circ T_2$. □

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 5.3.3 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.

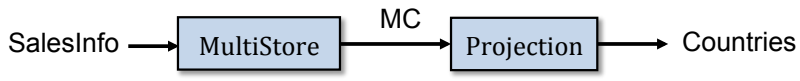

Figure 5.2: Example where combining logical provenance is not equivalent to workflow provenance.

Proof 5.3.3 Since T_2 has a filter condition C on I_2, we know that for all I'_1, J'_1, J'_2, $(T_1 \circ T_2)(I'_1, J'_1, J'_2) = T_2(\sigma_C(T_1(I'_1, J'_1)_2), J'_2)$. We need to show that for all I'_1, J'_1, $\sigma_C(T_1(I'_1, J'_1)_2) = \sigma_C(T_1(\sigma_{C'}(I'_1), J'_1)_2)$. Suppose $\sigma_C(T_1(I'_1, J'_1)_2) \neq \sigma_C(T_1(\sigma_{C'}(I'_1), J'_1)_2)$. Then there exists some o that is in one side but not the other. Since o is only in one side, it follows that $\sigma_{\mathbf{B}=o.\mathbf{B}}(\sigma_C(T_1(I'_1, J'_1)_2)) \neq \sigma_{\mathbf{B}=o.\mathbf{B}}(\sigma_C(T_1(\sigma_{C'}(I'_1), J'_1)_2))$. But since the attribute mappings hold, we know $\sigma_{\mathbf{B}=o.\mathbf{B}}(\sigma_C(T_1(I'_1, J'_1)_2)) = \sigma_{\mathbf{B}=o.\mathbf{B}}(\sigma_C(T_1(\sigma_{\mathbf{A}=o.\mathbf{B}}(I'_1), J'_1)_2)) = \sigma_{\mathbf{B}=o.\mathbf{B}}(\sigma_C(T_1(\sigma_{\mathbf{A}=o.\mathbf{B}\wedge C'}(I'_1), J'_1)_2)) = \sigma_{\mathbf{B}=o.\mathbf{B}}(\sigma_C(T_1(\sigma_{C'}(I'_1), J'_1)_2))$, a contradiction. Thus $\sigma_C(T_1(I'_1, J'_1)_2) = \sigma_C(T_1(\sigma_{C'}(I'_1), J'_1)_2)$. □

5.3.2 Relationship Between Combined Logical Provenance and Workflow Provenance

Although attribute mappings and filters can be combined across transformations using Theorems 5.3.1, 5.3.2, and 5.3.3, sometimes the combined logical provenance is weaker (less precise) than the workflow provenance computed by considering transformations separately, accessing intermediate data between transformations.

Example 5.3.2 Consider workflow W = **MultiStore** ∘ **Projection** shown in Figure 5.2. The initial input data set 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 ↔ MC.country and SalesInfo.city ↔ 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 2.3.1 is $\{(France, Paris, 10), (France, Paris, 20)\} \subseteq$ SalesInfo. By Theorem 5.3.1, 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 5.3.4 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 $I_3 = W(I_1', J_1', J_2')$ 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 $\langle I_1^*, J_1^*, J_2^* \rangle$ be o's provenance as encoded by (M_W, F_1) for W according to Definition 5.2.3. Let $\langle I_1^{**}, J_1^{**}, J_2^{**} \rangle$ be the workflow provenance of o according to Definition 2.3.1. Then $I_1^* = I_1^{**}$.

Proof. $I_1^* = \sigma_{(A_1=o.D_1)\wedge\ldots\wedge(A_r=o.D_r)\wedge F_1}(I_1)$. We now compute I_1^{**}. The provenance of o encoded by (M_2, F_2) for I_2 is $\sigma_{(B_1=o.D_1)\wedge\ldots\wedge(B_s=o.D_s)\wedge F_2}(I_2)$. By assumption, $\sigma_{(B_1=o.D_1)\wedge\ldots\wedge(B_s=o.D_s)\wedge F_2}(I_2)$ is nonempty. Then $I_1^{**} = \bigcup_{e' \in \sigma_{(B_1=o.D_1)\wedge\ldots\wedge(B_s=o.D_s)\wedge F_2}(I_2)} \sigma_{(A_1=e'.D_1)\wedge\ldots\wedge(A_r=e'.D_r)\wedge F_1}(I_1) = \sigma_{(A_1=o.D_1)\wedge\ldots\wedge(A_r=o.D_r)\wedge F_1}(I_1) = I_1^*$. □

5.3.3 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 5.3.4 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 5.3.4 (whether the provenance of output elements at intermediate data sets is nonempty), in rare cases the algorithm can reduce precision.

Algorithm 5.3.1 (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 denote 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 denote the subset of filters in F^T that are on I_i^T
if $I = I'$ **then**:
let $\langle I_1^i, \ldots, I_m^i \rangle = PT(E \subseteq I, M_i, F_i, I_i^T)$ for $1 \leq i \leq m_T$
else:
if every right attribute in M_i is a left attribute of M **then**:
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 $I_j^* = \bigcup_{1 \leq i \leq m_T} I_j^i$

5.4 Logical Provenance for Relational Transformations

We now consider logical provenance in the relational setting. We establish notation in Section 5.4.1 and discuss how to generate logical provenance specifications for *Select-Project-Join* (*SPJ*) *transformations* in Section 5.4.2. In Section 5.4.3 we show that for a wide class of SPJ transformations, provenance encoded by our logical specifications is minimal. For transformations outside of this class, in Section 5.4.4 we introduce *augmentation*, which carries some overhead but enables minimal provenance. In Section 5.4.5 we extend our results to *Select-Project-Join-Aggregate* (*SPJA*) *transformations*.

5.4.1 Preliminaries

In the relational setting, all data sets are *tables*. A table contains a set of tuples $\{t_1, \ldots, t_n\}$ conforming to a given schema. We assume set semantics.

Definition 5.4.1 (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*. Note since we assume set semantics, 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 [39]. Note $\mathbf{A}$ is the final projection list, C_i are "local" (single-table) selection conditions, and C contains "join" (multi-table) conditions. Hereafter we will operate on the canonical form of relational transformations.

5.4.2 Logical Provenance for SPJ Transformations

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 5.4.1 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.

Proof. Let $I'_1, \ldots, I'_m$ be any input set instances and let x be any possible value of $O.A$.

$$
\begin{aligned}
&\sigma_{A=x}(T(I'_1, \ldots, I'_m)) \\
&\quad = \sigma_{A=x}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I'_1) \times \ldots \times \sigma_{C_m}(I'_m)))) \\
&\quad = \sigma_{A=x}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{A=x}(\sigma_{C_1}(I'_1) \times \ldots \times \sigma_{C_m}(I'_m))))) \\
&\quad = \sigma_{A=x}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I'_1) \times \ldots \times \sigma_{C_i}(\sigma_{A=x}(I'_i)) \times \ldots \times \sigma_{C_m}(I'_m)))) \\
&\quad = \sigma_{A=x}(T(I'_1, \ldots, \sigma_{A=x}(I'_i), \ldots, I'_m))
\end{aligned}
$$

□

Theorem 5.4.2 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.

Proof. Let $I'_1, \ldots, I'_m$ be any input set instances. Then we have:
$T(I'_1, \ldots, I'_m)$
$= \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I'_1) \times \ldots \times \sigma_{C_i}(I'_i) \times \ldots \times \sigma_{C_m}(I'_m)))$
$= \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I'_1) \times \ldots \times \sigma_{C_i}(\sigma_{C_i}(I'_i)) \times \ldots \times \sigma_{C_m}(I'_m)))$
$= T(I'_1, \ldots, \sigma_{C_i}(I'_i), \ldots, I'_m)$ □

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

Definition 5.4.2 (Canonical Specification for SPJ Transformation) 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. □

5.4.3 Encoding Minimal Provenance for SPJ Transformations

We show that for a certain class of SPJ transformations, the provenance encoded by the canonical logical specification is minimal.

Theorem 5.4.3 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 $\langle I^*_1, \ldots, I^*_m \rangle$ be the provenance of o as encoded by the canonical logical specification for T. $\langle I^*_1, \ldots, I^*_m \rangle$ is minimal for o with respect to T. □

The Theorem's proof follows directly from the following two Lemmas.

Lemma 5.4.1 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 that $\mathbf{A}$ contains all attributes in C. Given $o \in O$, let $\langle I^*_1, \ldots, I^*_m \rangle$ be the provenance of o as encoded by the canonical logical

specification for T. Let $\mathbf{I_i}$ contain all attributes in the schema of I_i. Then $\langle I_1^*, \ldots, I_m^* \rangle = \langle I_1^{**}, \ldots, I_m^{**} \rangle$ where each $I_i^{**} = \pi_{\mathbf{I}_i}(\sigma_{C \wedge \mathbf{A}=o}(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$.

Proof. Let $\mathbf{A}_i$ contain the attributes of I_i in $\mathbf{A}$. The provenance of o as encoded by T's canonical logical provenance specification is $\langle I_1^*, \ldots I_m^* \rangle = \langle \sigma_{\mathbf{A}_1=o.\mathbf{A}_1 \wedge C_1}(I_1), \ldots, \sigma_{\mathbf{A}_m=o.\mathbf{A}_m \wedge C_m}(I_m) \rangle$. Let us show that $I_i^* = I_i^{**}$.

First suppose $e_i \in I_i^* = \sigma_{\mathbf{A}_i=o.\mathbf{A}_i \wedge C_i}(I_i)$. Then $e_i \in I_i$, and since $o \in T(I_1, \ldots, I_m) = \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$, $o \in \sigma_{\mathbf{A}=o}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))) = \pi_{\mathbf{A}}(\sigma_{C \wedge (\mathbf{A}=o)}(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m))) = \pi_{\mathbf{A}}(\sigma_{C \wedge (\mathbf{A}=o)}(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{\mathbf{A}_i=o.\mathbf{A}_i \wedge C_i}(I_i) \times \ldots \times \sigma_{C_m}(I_m)))$. Since $\mathbf{A}$ contains all attributes in C, $\pi_{\mathbf{A}}(\sigma_{C \wedge (\mathbf{A}=o)}(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{\mathbf{A}_i=o.\mathbf{A}_i \wedge C_i}(I_i) \times \ldots \times \sigma_{C_m}(I_m))) = \pi_{\mathbf{A}}(\sigma_{C \wedge (\mathbf{A}=o)}(\sigma_{C_1}(I_1) \times \ldots \times \{e_i\} \times \ldots \times \sigma_{C_m}(I_m)))$. Since $o \in \pi_{\mathbf{A}}(\sigma_{C \wedge (\mathbf{A}=o)}(\sigma_{C_1}(I_1) \times \ldots \times \{e_i\} \times \ldots \times \sigma_{C_m}(I_m)))$, there exists some $(e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_m) \in (\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_{i-1}}(I_{i-1}) \times \sigma_{C_{i+1}}(I_{i+1}) \times \ldots \times \sigma_{C_m}(I_m))$ such that $\sigma_{C \wedge \mathbf{A}=o}(\{e_1\} \times \ldots \times \{e_m\}) = \{e_1\} \times \ldots \times \{e_m\}$. Thus $I_i^{**} = \pi_{\mathbf{I}_i}(\sigma_{C \wedge \mathbf{A}=o}(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m))) \supseteq \pi_{\mathbf{I}_i}(\{e_1\} \times \ldots \times \{e_m\}) \supseteq \{e_i\}$, implying $e_i \in I^{**}$.

Now suppose $e_i \in I_i^{**} = \pi_{\mathbf{I}_i}(\sigma_{C \wedge \mathbf{A}=o}(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$. Then $e_i \in I_i$, $e_i.\mathbf{A}_i = o.\mathbf{A}_i$, and e_i satisfies C_i. Thus, $e_i \in \sigma_{(\mathbf{A_i}=o.\mathbf{A_i}) \wedge C_i}(I_i) = I_i^*$. □

Lemma 5.4.2 Consider an SPJ transformation instance: $O = T(I_1, \ldots, I_m) = \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$. Given $o \in O$, consider $\langle I_1^*, \ldots, I_m^* \rangle$ where each $I_i^* = \pi_{\mathbf{I}_i}(\sigma_{C \wedge \mathbf{A}=o}(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$. $\langle I_1^*, \ldots, I_m^* \rangle$ is minimal for o with respect to T.

Proof. We first show that $\langle I_1^*, \ldots, I_m^* \rangle$ is correct. Consider any $I_1' \subseteq I_1, \ldots I_m' \subseteq I_m$.

First suppose $o \in T(I_1', \ldots, I_m')$. Let us show that $o \in T(I_1' \cap I_1^*, \ldots, I_m' \cap I_m^*)$. Since $o \in T(I_1', \ldots, I_m')$, there exist $e_1 \in I_1', \ldots, e_m \in I_m'$ such that $\{o\} = \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(\{e_1\}) \times \ldots \times \sigma_{C_m}(\{e_m\})))$. We know $e_i \in I_i^*$, since $\{e_i\} = \pi_{\mathbf{I}_i}(\sigma_{C \wedge A=o}(\sigma_{C_1}(\{e_1\}) \times \ldots \times \sigma_{C_m}(\{e_m\}))) \subseteq \pi_{\mathbf{I}_i}(\sigma_{C \wedge A=o}(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m))) = I_i^*$. Thus, $e_1 \in I_1' \cap I_1^*, \ldots, e_m \in I_m' \cap I_m^*$, and it follows that $o = T(\{e_1\}, \ldots, \{e_m\}) \subseteq T(I_1' \cap I_1^*, \ldots, I_m' \cap I_m^*)$.

Now suppose $o \in T(I_1' \cap I_1^*, \ldots, I_m' \cap I_m^*)$. Since T is monotonic and $I_1' \cap I_1^* \subseteq I_1', \ldots, I_m' \cap I_m^* \subseteq I_m'$, $T(I_1' \cap I_1^*, \ldots, I_m' \cap I_m^*) \subseteq T(I_1', \ldots, I_m')$, and so $o \in T(I_1', \ldots, I_m')$. Thus, $\langle I_1^*, \ldots, I_m^* \rangle$ is correct.

Let us now show $\langle I_1^*, \ldots, I_m^* \rangle$ is minimal. Suppose there existed a more precise $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ that was also correct. Then there exists some j for which $I_j^{**} \subset I_j^*$, i.e., there exists some element e_j such that $e_j \in I_j^*$, $e_j \notin I_j^{**}$. Since $e_j \in I_j^* = \pi_{\mathbf{I}_j}(\sigma_{C \wedge \mathbf{A}=o}(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$, there exists some $(e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_m) \in (\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_{j-1}}(I_{j-1}) \times \sigma_{C_{j+1}}(I_{j+1}) \times \ldots \times \sigma_{C_m}(I_m))$ such that $\{o\} = \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(\{e_1\}) \times \ldots \times \sigma_{C_m}(\{e_m\})))$. Let us choose subsets $I_1' = \{e_1\}, \ldots, I_m' = \{e_m\}$. Since we are assuming that $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ is correct and $o \in T(I_1', \ldots, I_m')$, we would expect o to be in $T(I_1' \cap I_1^{**}, \ldots, I_m' \cap I_m^{**})$. But $T(I_1' \cap I_1^{**}, \ldots, I_m' \cap I_m^{**}) \subseteq T(\{e_1\}, \ldots, \{e_{j-1}\}, \emptyset, \{e_{j+1}\}, \ldots, \{e_m\}) = \emptyset$. Thus, $o \notin T(I_1' \cap I_1^{**}, \ldots, I_m' \cap I_m^{**})$, a contradiction. There exists no correct $\langle I_1^{**}, \ldots, I_m^{**} \rangle$ that is more precise, implying that $\langle I_1^*, \ldots, I_m^* \rangle$ is minimal. □

Theorem 5.4.3 follows directly from Lemmas 5.4.1 and 5.4.2. We also note that $\langle I_1^*, \ldots, I_m^* \rangle$ where each $I_i^* = \pi_{\mathbf{I}_i}(\sigma_{C \wedge \mathbf{A}=o}(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$ is proven in [22] to be the *lineage* of an SPJ transformation. Thus, we have shown via Lemma 5.4.2 that the lineage of an SPJ transformation given in [22] corresponds to our definition of minimal provenance (Definition 2.2.6). Of course our notion of minimal provenance also supports general transformations.

5.4.4 Augmentation

Given an SPJ transformation $T(I_1, \ldots, I_m) = \pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m)))$, recall that Theorem 5.4.3 only guarantees that the canonical logical specification given in Definition 5.4.2 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 5.4.3 (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.

Definition 5.4.4 (Provenance Encoded by Augmentation) Let $O = T(I_1, \ldots, I_m)$ be a transformation instance with output schema $\mathbf{A}$. Let T' be an augmentation of T with output schema $\mathbf{A}' \supset \mathbf{A}$ and logical specification (M, F). Let $O' = T'(I_1, \ldots, I_m)$. Given $o' \in O'$, let $I^*(o') = \langle I_1^*(o'), \ldots, I_m^*(o') \rangle$ be the provenance of o' as encoded by the logical specification for T'. Given $o \in O$, the provenance of o as encoded by (M, F) is $\langle I_1^*, \ldots, I_m^* \rangle$, where each $I_i^* = \bigcup_{(o' \in O') \wedge (\pi_{\mathbf{A}}(o') = o)} I_i^*(o')$. □

Theorem 5.4.4 (Correctness of Provenance Encoded by Augmentation) Let $O = T(I_1, \ldots, I_m)$ be a transformation instance with output schema $\mathbf{A}$. Let T' be an augmentation of T with output schema $\mathbf{A}' \supset \mathbf{A}$ and logical specification (M, F). Given $o \in O$, the provenance of o as encoded by (M, F) is correct for o with respect to T.

Proof. Let $O' = T'(I_1, \ldots, I_m)$. Then $o \in O$ if and only if there exists $o' \in O'$ such that $\pi_{\mathbf{A}}(o') = o$. Consider any $I_1' \subseteq I_1, \ldots, I_m' \subseteq I_m$. We have:

$$
\begin{aligned}
&\{o\} \cap T(I_1', \ldots, I_m') \\
&\quad = \{o\} \cap \pi_{\mathbf{A}}(T'(I_1', \ldots, I_m')) \\
&\quad = \{o\} \cap \pi_{\mathbf{A}}(T'(I_1', \ldots, I_m') \cap \sigma_{\mathbf{A}=o.\mathbf{A}}(O')) \\
&\quad = \{o\} \cap \pi_{\mathbf{A}}(T'(I_1' \cap I_1^*, \ldots, I_m' \cap I_m^*) \cap \sigma_{\mathbf{A}=o.\mathbf{A}}(O')) \\
&\quad = \{o\} \cap \pi_{\mathbf{A}}(T'(I_1' \cap I_1^*, \ldots, I_m' \cap I_m^*)) \\
&\quad = \{o\} \cap T(I_1' \cap I_1^*, \ldots, I_m' \cap I_m^*)
\end{aligned}
$$

□

5.4.5 Logical Provenance for SPJA Transformations

We now extend our results to *Select-Project-Join-Aggregate* (*SPJA*) transformations. We first introduce the *aggregation* operator.

Definition 5.4.5 (Aggregation) Let I be an input table I with schema $\mathbf{I}$, and let $\mathbf{G} \subseteq \mathbf{I}$, $\mathbf{B} \subseteq \mathbf{I}$. Let $g_1, \ldots, g_n$ be all of the distinct values of $\mathbf{G}$ in I, and let each group $G_i = \sigma_{\mathbf{G}=g_i}(I)$. The *aggregation* operator $\alpha_{\mathbf{G},aggr(\mathbf{B})\to V}$ takes an input table I and outputs for each group $G_i \subseteq I$ the tuple $\langle g_i, v_i \rangle$, where $v_i = aggr(G_i.\mathbf{B})$ and $aggr$ is an aggregation function on $\mathbf{B}$: $\alpha_{\mathbf{G},aggr(\mathbf{B})\to V}(I) = \bigcup_{1\leq i\leq n}\{\langle g_i, v_i\rangle\}$. We could extend to multiple aggregation functions, but the added complexity wouldn't introduce any interesting challenges. □

Definition 5.4.6 (SPJA Transformation) A *Select-Project-Join-Aggregate* (*SPJA*) *transformation* is a transformation that can be expressed in the canonical form $T(I_1, \ldots, I_m) = \pi_{\mathbf{A}_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m))))))$. □

In the above canonical form, $\mathbf{A}_0$ and C_0 refer to the schema $\mathbf{G} \cup \{V\}$. Note that there exist transformations that can be expressed as a tree of selection (σ), projection (π), cross-product ($\times$), and aggregation (α) operators that cannot be transformed into the above canonical form. However, any such transformation can be expressed as a tree of SPJA transformations, each of which can be transformed into the canonical form; see [39] for details. Our work considers only SPJA transformations with this canonical form, assuming more complex queries will be split into multiple transformations.

Two of the theorems for SPJ transformations have a corresponding theorem for SPJA transformations.

Theorem 5.4.5 Let T be an SPJA transformation: $O = T(I_1, \ldots, I_m) = \pi_{\mathbf{A}_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m))))))$. For each attribute $I_i.A \in (\mathbf{A}_0 \cap \mathbf{G})$, attribute mapping $I_i.A \leftrightarrow O.A$ holds for T.

Proof. Let $I'_1, \ldots, I'_m$ be any input set instances and let x be any possible value of A. Since $A \in \mathbf{A}_0$ and $A \in \mathbf{G}$, we have:

$\sigma_{A=x}(T(I'_1, \ldots, I'_m))$
$= \sigma_{A=x}(\pi_{\mathbf{A}_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I'_1) \times \ldots \times \sigma_{C_m}(I'_m)))))))$
$= \sigma_{A=x}(\pi_{\mathbf{A}_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\sigma_{A=x}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I'_1) \times \ldots \times \sigma_{C_m}(I'_m)))))))$
$= \sigma_{A=x}(\pi_{\mathbf{A}_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{A=x}(\sigma_{C_1}(I'_1) \times \ldots \times \sigma_{C_m}(I'_m)))))))$
$= \sigma_{A=x}(\pi_{\mathbf{A}_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I'_1) \times \ldots \times \sigma_{C_i}(\sigma_{A=x}(I'_i)) \times \ldots \times \sigma_{C_m}(I'_m)))))))$
$= \sigma_{A=x}(T(I'_1, \ldots, \sigma_{A=x}(I'_i), \ldots, I'_m))$ □

Theorem 5.4.6 Let T be an SPJA transformation: $O = T(I_1, \ldots, I_m) = \pi_{\mathbf{A}_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m))))))$. Then T has filter condition C_i on I_i.

Proof. Let $I'_1, \ldots, I'_m$ be any input set instances. Then we have:
$T(I'_1, \ldots, I'_m)$
$= \pi_{\mathbf{A}_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I'_1) \times \ldots \times \sigma_{C_i}(I'_i) \times \ldots \times \sigma_{C_m}(I'_m))))))$
$= \pi_{\mathbf{A}_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I'_1) \times \ldots \times \sigma_{C_i}(\sigma_{C_i}(I'_i)) \times \ldots \times \sigma_{C_m}(I'_m))))))$
$= T(I'_1, \ldots, \sigma_{C_i}(I'_i), \ldots, I'_m)$ □

As we did for SPJ transformations, we can generate canonical logical provenance specifications for SPJA transformations.

Definition 5.4.7 (Canonical Specification for SPJA Transformation) Let T be an SPJA transformation: $O = T(I_1, \ldots, I_m) = \pi_{\mathbf{A}_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m))))))$. The *canonical logical provenance specification* for T is (M, F) where M contains all mappings $I_i.A \leftrightarrow O.A$ such that $I_i.A \in (\mathbf{A}_0 \cap \mathbf{G})$, and F contains all C_i. □

Based on Theorems 5.2.1, 5.4.5, and 5.4.6, we know that the canonical specification for SPJA transformations encodes correct provenance. For SPJA transformations T in which the outermost projection operator $\pi_{\mathbf{A}_0}$ projects out attributes in $\mathbf{G}$, we can augment T such that all attributes in $\mathbf{G}$ are retained.

Definition 5.4.8 (Augmentation of SPJA Transformation) Consider an SPJA transformation $T(I_1, \ldots, I_m) = \pi_{\mathbf{A}_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times$

$\sigma_{C_m}(I_m)))))$ where $(\mathbf{A}_0 \cap \mathbf{G}) \subset (\mathbf{A} \cap \mathbf{G})$. The *canonical augmentation* of T is T' where $T'(I_1, \ldots, I_m) = \pi_{\mathbf{A}'_0}(\sigma_{C_0}(\alpha_{\mathbf{G},aggr(B)\to V}(\pi_{\mathbf{A}}(\sigma_C(\sigma_{C_1}(I_1) \times \ldots \times \sigma_{C_m}(I_m))))))$ and $\mathbf{A}'_0 = \mathbf{A}_0 \cup \mathbf{G}$. □

Given an augmentation T', its logical specification can contain additional attribute mappings for all attributes in $\mathbf{G}$ that are retained by $\pi_{\mathbf{A}'_0}$ but not by $\pi_{\mathbf{A}_0}$. However, even for SPJA transformations that retain all attributes in $\mathbf{G}$, the following example demonstrates that the logical specification does not always encode minimal provenance.

Example 5.4.1 Consider transformation **CountryAgg** with input data set SalesInfo containing country-sales pairs for each of a corporation's worldwide stores: $\mathsf{SalesInfo}(\mathsf{country}, \mathsf{sales}) = \{(France, 10), (Germany, 20), (Germany, 0)\}$. Transformation **CountryAgg** sums up the sales for each country, producing output table $\mathsf{CountrySales} = \alpha_{country,SUM(sales)}(\mathsf{SalesInfo}) = \{(France, 10), (Germany, 20)\}$. The canonical logical provenance specification is (M, F), where $M = \{\mathsf{SalesInfo.country} \leftrightarrow \mathsf{CountrySales.country}\}$ and $F = \emptyset$. Let $o = (Germany, 20) \in \mathsf{CountrySales}$. Then o's provenance as encoded by (M, F) is $\sigma_{\mathsf{country}=\text{`Germany'}}(\mathsf{SalesInfo}) = \{(Germany, 20), (Germany, 0)\}$. However, the minimal provenance of o is simply $\{(Germany, 20)\}$, since input tuple $(Germany, 0)$ has no impact on the presence or absence of o. □

In the above example, we see that input values of 0 are not in the minimal provenance of a nonzero sum in the output. Similarly, for MIN and MAX aggregates, the minimal provenance of an output tuple only contains those input tuples with the corresponding minimum or maximum value. Since the focus of our work is on logical provenance for general transformations, we do not treat individual aggregate functions here as special cases; instead, we only capture the logical provenance that can be found by treating the individual aggregate functions as black boxes.

5.5 Related Work

Provenance in the context of schema mappings is studied in, e.g., [20, 26, 40]. References [20] and [26] both consider schema mapping languages that cannot express general transformations (e.g., aggregation). Reference [40] considers a different type of provenance that consists of schema elements and transformations as opposed to input data elements.

Reference [6] uses Pig Latin [34] 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 [18] presents techniques for reducing the space overhead of provenance storage, while our logical provenance uses transformation properties to avoid storing physical provenance altogether. Reference [42] 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 [25] 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.

5.6 Conclusions

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, showing that for a class of Select-Project-Join (SPJ) transformations, logical provenance specifications encode minimal provenance.

Chapter 6

Panda System

We have built a prototype system called *Panda* (for *Provenance And Data*) that supports data-oriented workflows, with provenance capture and tracing. Panda permits data-oriented workflows of arbitrary (but acyclic) structure, with each transformation specified in either SQL or in Python [2].

There have been multiple versions of the Panda system as we have developed our work. In Section 6.1 we describe the high-level architecture of the final version of Panda, which is based on logical provenance. In Section 6.2 we describe how Panda generates logical provenance specifications and executes provenance tracing. In Section 6.3 we present some performance results for logical provenance. In Section 6.4 we describe an earlier version of Panda that was used to experiment with provenance-based refresh. In Section 6.5 we report experimental results based on this earlier version; we consider the overhead of provenance capture, and the crossover point between selective refresh and full workflow recomputation.

To the best of our knowledge, no other general-purpose data-oriented workflow system (e.g., [31, 33]) supports significant provenance functionality. Therefore, there is no separate related work section in this chapter.

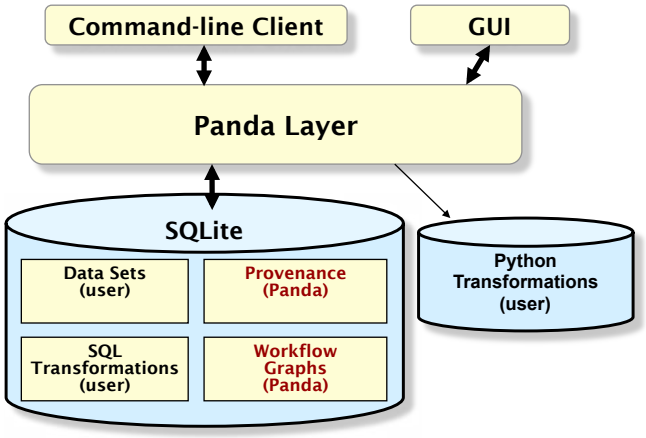

Figure 6.1: Architecture of the prototype Panda system.

6.1 System Architecture

The high-level architecture of the Panda system is shown in Figure 6.1. Each data set handled by Panda is encoded in a separate relational table, 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. Python transformations are stored separately in files.

Users interact with Panda through either the command-line interface or the Panda GUI. There are three main types of user commands: (1) Creating or modifying input data sets; (2) Creating transformations that generate newly-defined data sets from existing ones, to build up workflows; (3) Provenance tracing. The *Panda Layer* processes all user commands: It stores workflow graphs and their transformations, generates logical provenance specifications for each transformation as described in Chapter 5, and executes tracing operations.

6.2 Generating and Tracing Logical Provenance

This section references both the foundations of provenance presented in Chapter 2 and the logical-provenance specification language and formalism presented in Chapter 5.

We describe how Panda generates logical provenance specifications for SQL transformations (Section 6.2.1) and Python transformations (Section 6.2.2). We then describe how Panda performs provenance tracing (Section 6.2.3). We focus on backward tracing in this chapter, although our system supports both backward and forward tracing.

6.2.1 SQL Transformations

Panda supports transformations specified as SQL queries, restricted to single SELECT blocks with optional grouping and aggregation. Note that this form of query corresponds to the SPJA transformations in Definition 5.4.6 of Section 5.4.5, although in our system we restrict WHERE clauses to conjunctive conditions.

As an example, consider transformation **JoinAgg** from our running example (Figure 1.1). 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
```

In relational algebra, $\textbf{JoinAgg}(\mathsf{CS}, \mathsf{IP})$ can be expressed as:

$$\alpha_{\textit{item-id},\textit{country},\textit{brand},\textit{type},\textit{SUM}(\textit{quantity}*\textit{profit\_per\_item})\rightarrow\textit{profit}}(\sigma_{\textit{CS.item-id}=\textit{IP.item-id}}(\mathsf{CS}\times\mathsf{IP}))$$

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): $M = \{\mathsf{CS.item\_id} \leftrightarrow \mathsf{IC.item\_id}, \mathsf{CS.country} \leftrightarrow \mathsf{IC.country}, \mathsf{IP.item\_id} \leftrightarrow \mathsf{IC.item\_id}, \mathsf{IP.brand} \leftrightarrow \mathsf{IC.brand}, \mathsf{IP.type} \leftrightarrow \mathsf{IC.type}\}$, $F = \{\}$.

Now suppose the above query did not contain the item_id attribute in its SELECT clause. In this case Panda augments the query automatically as in Definition 5.4.7:

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
```

To illustrate how Panda augments SPJ transformations (i.e., without aggregation), consider the following query:

```
Create Table CountryBrands As
Select country, brand
From CustSales CS, ItemProfit IP
Where CS.item-id = IP.item-id
```

In relational algebra, the above query can be expressed as:

$\pi_{country,brand}(\sigma_{CS.item\text{-}id=IP.item\text{-}id}(\mathsf{CS} \times \mathsf{IP}))$

Note that this query does not satisfy the requirements of Theorem 5.4.3, because the attributes in the condition $CS.item\text{-}id = IP.item\text{-}id$ are not included in the final result. Thus, Panda augments the query so that the augmented transformation satisfies the requirements of Theorem 5.4.3:

```
Create Table AugCountryBrands As
Select country, brand, CS.item-id
From CustSales CS, ItemProfit IP
Where CS.item-id = IP.item-id
```

Since the requirements of Theorem 5.4.3 are now satisfied, logical provenance for the augmented transformation encodes minimal provenance. To recover the original output, Panda creates the following view:

```
Create View CountryBrands As
Select country, brand
From AugCountryBrands
```

6.2.2 Python Transformations

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 could be lifted with additional work. For Python transformations, Panda 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 (Figure 1.1). The user may provide Panda with the logical specification (M, F) for **Extract**: $M = \{\mathsf{CustData.cust\text{-}id} \leftrightarrow \mathsf{CS.cust\text{-}id}, \mathsf{CustData.country} \leftrightarrow \mathsf{CS.country}\}$, $F = \{\}$. If no user-specified logical provenance is provided, Panda augments each output record with the ID of the corresponding input record (calling it $\mathsf{src\_id}$ in the output), and generates the attribute mapping $\mathsf{I.ID} \leftrightarrow \mathsf{O.src\_id}$ 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.

6.2.3 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 from the source. To perform tracing, Panda generates and executes a SQL query that joins multiple tables, returning the same provenance as in Algorithm 5.3.1.

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.

Algorithm 6.2.1 (Provenance Tracing) Consider a workflow instance in which each transformation T has logical provenance specification (M^T, F^T). Let $o \in O$ be the output element we want to trace through $Path = \langle I_1, I_2, \ldots, I_n, I_{n+1} \rangle$ where $I_{n+1} = O$. Using helper functions *CombinePath* and *GenerateQuery*, PT returns a query that traces the provenance of o.

$PT(o \in O, Path)$:
    for i in $[1, n]$:
        let T_i be the transformation with input set I_i and output set I_{i+1}
        let M_i denote the subset of mappings $A \leftrightarrow B$ in M^{T_i} such that
            A is from I_i and B is from I_{i+1}
        let F_i denote the subset of filters in F^{T_i} that are on I_i
    let $ProvPath = \langle I_1, M_1, F_1, I_2, M_2, F_2, \ldots, I_n, M_n, F_n \rangle$
    let $CPath = CombinePath(ProvPath)$
    return $GenerateQuery(CPath, o \in O)$

// Combine logical provenance in *ProvPath*
$CombinePath(ProvPath)$:
    suppose $ProvPath = \langle I_1, M_1, F_1, I_2, M_2, F_2, \ldots, I_n, M_n, F_n \rangle$
    $CPath := \langle\rangle$
    $M := \emptyset$
    for i in $[n, n-1, \ldots, 2, 1]$
        if $M = \emptyset$ **then**:
            $M := M_i$
        else:
            if every right attribute in M_i is a left attribute of M **then**:
                suppose $M_i = \{A_1 \leftrightarrow B_1, \ldots, A_r \leftrightarrow B_r\}$
                suppose $M = \{B_1 \leftrightarrow D_1, \ldots, B_s \leftrightarrow D_s\}$, $r \leq s$
                $M := \{A_1 \leftrightarrow D_1, \ldots, A_r \leftrightarrow D_r\}$
            else:

$CPath := \langle I_i, M, F_i \rangle + CPath$
$M := M_i$
$CPath := \langle I_1, M, F_1 \rangle + CPath$
return $CPath$

// Trace combined logical provenance in *CPath* returned by *CombinePath*
GenerateQuery($CPath, o \in O$) :
suppose $CPath = \langle I_1, M_1, F_1, I_2, M_2, F_2, \ldots, I_{n'}, M_{n'}, F_{n'} \rangle$
for i in $[1, n']$:
suppose $M_i = \{A_1 \leftrightarrow B_1, \ldots, A_r \leftrightarrow B_r\}$
let $C_i = ((A_1 = B_1) \wedge \ldots \wedge (A_r = B_r) \wedge F_i)$
let $\mathbf{I}_1$ be the schema of I_1
return $\pi_{\mathbf{I}_1}(\sigma_{C_1 \wedge \ldots \wedge C_{n'} \wedge O.ID = o.ID}(I_1 \times \ldots \times I_{n'} \times O))$

As an example, consider a LaptopProfit record (ID=15 say) that we want to trace to destination CustSales (recall Figure 1.1 in Section 1.1). The path between LaptopProfit (LP) and CustSales (CS) contains intermediate data set ItemCountryProfit (IC): $Path = \langle \mathsf{CS}, \mathsf{IC}, \mathsf{LP} \rangle$. The attribute mappings between CS and IC are $M_1 = \{(\mathsf{CS.item\text{-}id} \leftrightarrow \mathsf{IC.item\text{-}id}), (\mathsf{CS.country} \leftrightarrow \mathsf{IC.country})\}$, and the attribute mappings between IC and LP are $M_2 = \{(\mathsf{IC.item\text{-}id} \leftrightarrow \mathsf{LP.item\text{-}id}), (\mathsf{IC.country} \leftrightarrow \mathsf{LP.country}), (\mathsf{IC.brand} \leftrightarrow \mathsf{LP.brand}), (\mathsf{IC.profit} \leftrightarrow \mathsf{LP.profit})\}$. Since all mappings in M_1 have corresponding mappings in M_2, *CombinePath* combines M_1 and M_2 to produce the following logical specification (M, F) between CS and LP: $M = \{\mathsf{CS.item\_id} \leftrightarrow \mathsf{LP.item\_id}, \mathsf{CS.country} \leftrightarrow \mathsf{LP.country}\}$, $F = \{\}$. Panda then uses *GenerateQuery* to generate the following tracing query:

```
Select Distinct CS.*
From CustSales CS, LaptopProfit LP
Where LP.item-id = CS.item-id
  And LP.country = CS.country And LP.ID = 15
```

Figure 6.2: Health insurance data workflow.

Panda uses `Distinct` in its tracing queries so that tuples that would otherwise appear multiple times in the tracing result appear only once.

6.3 Logical Provenance 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 6.3.1), and also enables more efficient provenance tracing by combining provenance across transformations (Section 6.3.2). Overhead and tracing performance may vary considerably depending on the particular workflow and data. Our experiments were not designed to explore these variations, 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 the real-world workflow shown in Figure 6.2. The workflow processes health insurance claim data from the Heritage Health Prize competition [4]. 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

The workflow involves the following transformations:

- **ExtractYear** extracts attribute year from ClaimData.data, producing data set YearData(memberID, providerID, diseaseID, data, year), abbreviated YD. **ExtractYear** is a Python transformation with logical specification (M, F):

M = {(ClaimData.memberID ↔ YD.memberID), (ClaimData.providerID ↔ YD.providerID), (ClaimData.diseaseID ↔ YD.diseaseID), (ClaimData.data ↔ YD.data)}, $F = \{\}$.

- **LookupType** joins YearData and DiseaseData on attribute diseaseID (and drops diseaseID), producing data set DiseaseType(memberID, providerID, data, year, type), abbreviated DT. **LookupType** is a SQL transformation that is augmented to keep join attribute diseaseID (recall Section 6.2.1). The logical specification for the augmented output DTA is (M, F): M = {(YD.memberID ↔ DTA.memberID), (YD.providerID ↔ DTA.providerID), (YD.diseaseID ↔ DTA.diseaseID),(YD.data ↔ DTA.data), (YD.year ↔ DTA.year), (DiseaseData.diseaseID ↔ DTA.diseaseID), (DiseaseData.type ↔ DTA.type)}, $F = \{\}$.
- **SelectYear** applies filter year=3 to DiseaseType and drops year, producing data set Year3Data(memberID, providerID, data, type), abbreviated Y3. **SelectYear** is a SQL transformation with logical specification (M, F): M = {(DT.memberID ↔ Y3.memberID), (DT.providerID ↔ Y3.providerID), (DT.data ↔ Y3.data), (DT.type ↔ Y3.type)}, F = {year=3}.
- **LookupDays** joins Year3Data and VisitData on attribute memberID, producing DaysData(memberID, providerID, data, type, days), abbreviated DD. **LookupDays** is a SQL transformation with logical specification (M, F): M = {(Y3.memberID ↔ DD.memberID), (Y3.providerID ↔ DD.providerID), (Y3.data ↔ DD.data), (Y3.type ↔ DD.type), (VisitData.memberID ↔ DD.memberID), (VisitData.days ↔ DD.days)}, $F = \{\}$.
- Finally, **AggDays** computes the average number of days spent in the hospital for each (type, providerID) group, producing data set AverageTime(type, providerID, avgdays). **AggDays** is a SQL transformation with logical specification (M, F): M = {(DD.type ↔ AverageTime.type), (DD.providerID ↔ AverageTime.providerID)}, $F = \{\}$.

All of the experiments in this section were run on a MacBook Air laptop (1.8 GHz

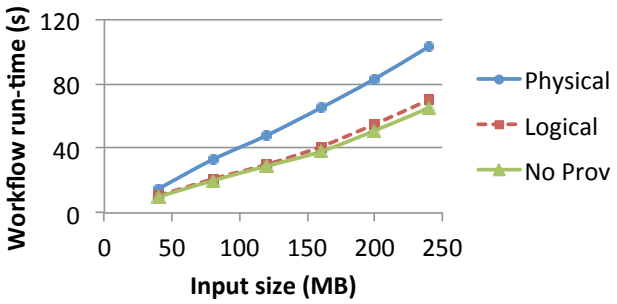

Figure 6.3: Time overhead of provenance capture.

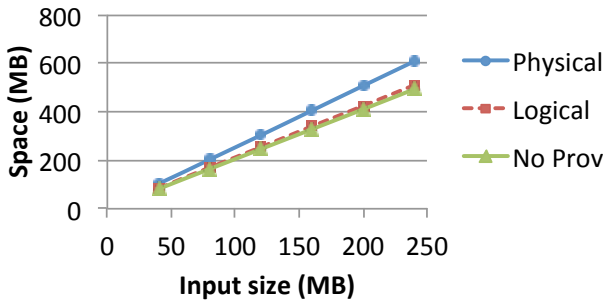

Figure 6.4: Space overhead of provenance capture.

Intel Core i7, 4 GB memory, 250 GB storage, Mac OS X 10.7). The time performance numbers we report are averages taken over 5 runs; we observed little variation between runs. To measure the performance of physical provenance, we used a modified version of Panda that stores physical provenance in separate provenance tables.

6.3.1 Time and Space Overhead

Figure 6.3 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 8% for logical provenance, and from 50% to 71% for physical provenance. The time overhead for logical provenance is due to the additional output data produced by augmentation.

Figure 6.4 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

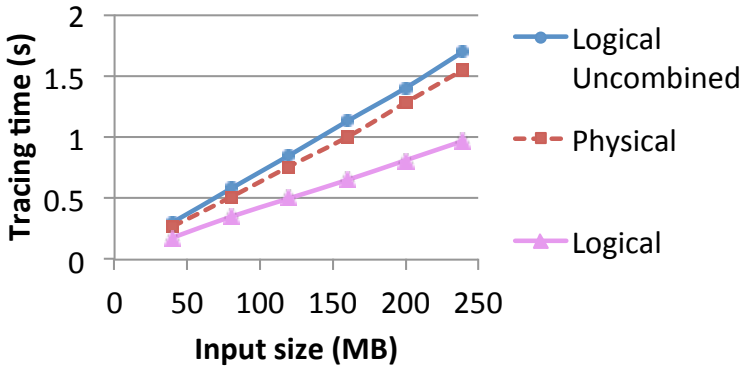

Figure 6.5: Provenance tracing time.

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.

6.3.2 Tracing Time

We measured the time to trace output tuples through our workflow to input data set ClaimData, after the workflow was run collecting physical or logical provenance. We report two times for logical provenance (Figure 6.5):

1. **Logical:** Time to trace one output tuple using Panda's tracing algorithm as presented in Algorithm 6.2.1
2. **Logical Uncombined:** Time to trace one output tuple using a tracing algorithm that does not combine provenance across transformations

For our workflow, Algorithm 6.2.1 combines provenance twice, eliminating intermediate data sets YearData and Year3Data from the tracing query (recall Section 5.3.3). 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 6.5, tracing using uncombined logical provenance is 10-14% 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 Algorithm 6.2.1 enables faster tracing; tracing using combined logical provenance is 32-37% faster than using physical provenance.

6.4 Supporting Provenance-Based Refresh

In this section, we describe an earlier version of Panda that was used to experiment with provenance-based refresh. We call this version *Panda*^{R}. This section references the provenance-based refresh material presented in Chapter 4. We describe how Panda^R obtains provenance predicates for SQL transformations (Section 6.4.1) and Python transformations (Section 6.4.2). We then describe how Panda^R performs provenance-based refresh (Section 6.4.3). Performance experiments will be in Section 6.5.

6.4.1 SQL Transformations

For SQL transformations, Panda^R creates provenance predicates automatically, following known definitions and techniques [16, 19, 22]: Single-table **Select** statements are one-one, so their output provenance predicates can select on declared keys from the input data set. Multi-table **Select** statements generate provenance predicates for each input table separately as described in Section 4.5, again relying on declared keys. Finally, **Group-by** queries generate provenance predicates based on the grouping attribute(s).

As an example of a SQL transformation, consider adding transformation **Join** as we build up our genetic risk workflow example from Section 4.1.1 (Figure 4.1). The user appends the transformation to the workflow with the following command:

```
Create Table PatientRisks As
Select name, disease, risk
From PatientDNA, DNARisks
Where PatientDNA.loc = DNARisks.loc And PatientDNA.seq = DNARisks.seq
```

In response to this command, Panda^R creates the new table PatientRisks, and creates an additional table to store the provenance predicates. Forward filters are never needed since SQL queries cannot produce many-many transformations.

6.4.2 Python Transformations

To support refresh, Panda^R requires that Python transformations output provenance predicates with each output element. As an example, consider transformation **PatientDL** from our genetic risk workflow example (Figure 4.1). The user writes a Python script **patientdl.py** that takes as its argument a list of URLs, and returns the set of XML documents located at the URLs, with corresponding provenance predicates. The user then appends transformation **PatientDL** to the workflow with the following command:

```
Create Table RawPatientData
As Python 'patientdl.py' on PatientURLs
```

In response to this command, Panda^R creates the new table RawPatientData, as well as an additional table to store the provenance predicates. For many-many transformations, the Python script would also need to create a forward filter (recall Section 4.4) for each output element.

6.4.3 Refresh

When workflows are created and run, Panda^R stores everything needed to support selective refresh: provenance predicates and intermediate data sets for backward-tracing; transformations and forward filters for forward-propagation. Panda^R assumes that all transformations, provenance, and workflows satisfy the requirements specified in Chapter 4. Automatically detecting when the requirements are satisfied—particularly the most interesting requirement of workflow safety (Requirement 4.3.1)—is an important area of future work.

Under the assumption of all requirements being satisfied, Panda^R supports selective refresh using the exact algorithms given in Chapter 4. When an output element

o is refreshed, if a new value o' is produced then Panda^R automatically replaces o with o'. If the refresh results in an empty set, then o is deleted. However, we leave a visible "tombstone" for o with its associated provenance predicate (and forward filter if present). Then, if desired we can refresh the tombstone at a later time and possibly discover that further input modifications have created a new value for o.

6.5 Refresh Experiments

The primary goal of this empirical study was to determine, for varying workflow characteristics, when it is advantageous to perform selective refresh as opposed to rerunning the entire workflow. Specifically, how many refreshes can we perform before their aggregate running time—including the extra time spent to capture provenance—exceeds that of complete recomputation?

For this empirical study we used the example workflow described in Section 4.1.1 (Figure 4.1), with fabricated data. Transformations **Join** and **Filter** use SQL, while transformations **PatientDL**, **DNADL**, **PExtract**, and **RiskExtract** were coded in Python. We indexed the intermediate data sets so that our provenance predicates could be evaluated efficiently. All of the experiments in this section were run using Panda^R on a MacBook Air laptop (1.8 GHz Intel Core i7, 4 GB memory, 250 GB storage, Mac OS X 10.7). To study how different workflow characteristics impact the overhead of provenance capture and the performance of refresh, we ran experiments for varying data sizes and varying transformation costs. The time performance numbers we report are averages taken over 5 runs; we observed little variation between runs.

Our performance results are summarized as follows:

- The time and space overhead of provenance capture is proportional to the time and space required by workflow execution, which in turn is proportional to the amount of input data. We observed roughly 25% time overhead and 56% space overhead for various input sizes to capture provenance while executing the workflow.

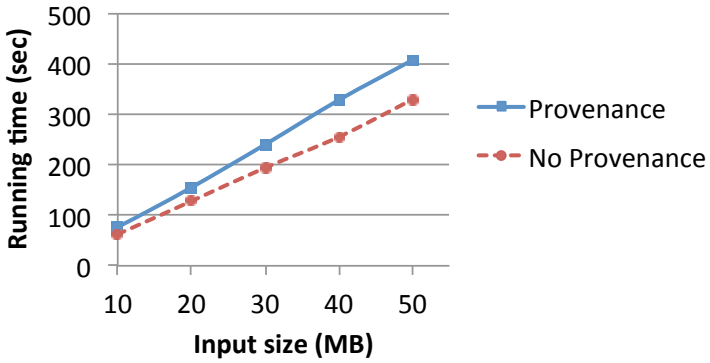

Figure 6.6: Time overhead of provenance capture, vary input size.

- Not surprisingly, the relative time overhead of provenance capture and provenance tracing decreases as individual transformations get more expensive. Thus, selective refresh is most advantageous for workflows with relatively expensive transformations.
- In our experiments, we were able to refresh between 47% and 71% of the data elements in the output data set before the running time (including provenance capture) exceeded that of complete recomputation. The crossover point was independent of input data size, but became more favorable for selective refresh as transformations got more expensive.

6.5.1 Overhead of Provenance Predicates

Figure 6.6 shows the time overhead of provenance capture for varying input data sizes. Comparing the running times of workflow computation with and without provenance, we see that the time overhead is roughly proportional to the amount of input data, ranging from 21% to 29%

Figure 6.7 shows the space overhead of provenance capture for varying input data sizes. We totaled all data involved in the workflow, including intermediate data, with and without provenance. Storing provenance, which includes provenance predicates and forward filters, incurred a 56% overhead across all input data sizes. Note the space overhead would be considerably lower for "wider" data elements; our fabricated data elements are relatively small.

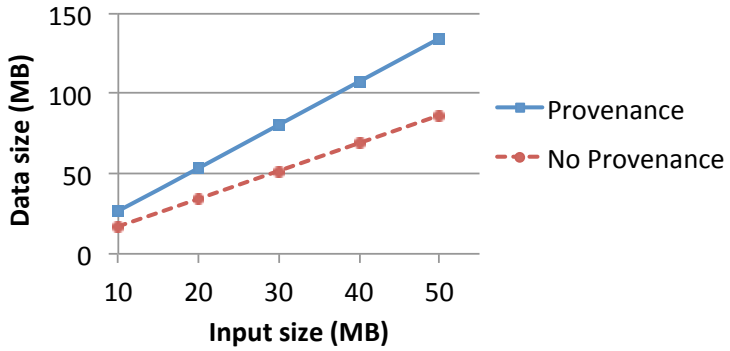

Figure 6.7: Space overhead of provenance capture, vary input size.

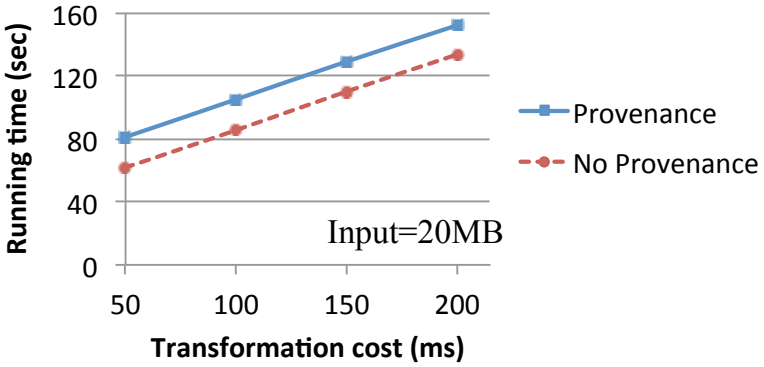

Figure 6.8: Time overhead of provenance capture, vary transformation cost.

Next we measured the impact of transformation cost on time overhead of provenance capture. We modified the two transformations that download data from the web, **PatientDL** and **DNADL**, to perform local downloads instead, and we instrumented them with a configurable delay. Figure 6.8 shows the time overhead of provenance capture, varying the costs of (i.e., the delays in) transformations **PatientDL** and **DNADL**. We can see that the larger the transformation costs, the lower the relative time overhead of provenance capture; e.g., time overhead is 32% at 50ms vs. 14% at 200ms. Intuitively, when transformations are more expensive, provenance capture plays a smaller role.

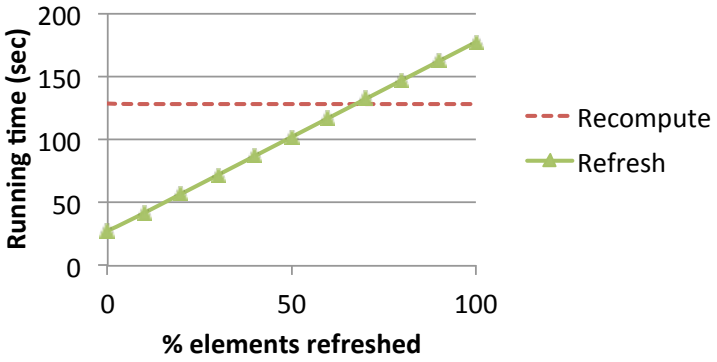

Figure 6.9: Recomputation and refresh costs.

6.5.2 Crossover Point for Refresh

Our next experiments identify the crossover point between selective refreshes (including provenance capture) and full recomputation (without provenance capture). For starters, the two lines in Figure 6.9 plot the running times of:

1. **Recompute:** Time required to fully (re)compute the workflow.
2. **Refresh:** Total time to selectively refresh a varying fraction of the output data elements. To make a fair comparison, we added the time overhead of provenance capture to the total time.

In this experiment we used an input size of 20 MB and the original transformation costs for **PatientDL** and **DNADL**. From Figure 6.9 we see that we can capture provenance and then refresh over 60% of the data elements in the output data set before the total running time exceeds that of recomputation. Obviously, selective refresh is most advantageous when only a small subset of the output elements are refreshed, however in our experiment a significant fraction of output elements needed to be refreshed before we observed no advantage at all.

Figure 6.10 shows the crossover point (in terms of percentage of refreshed elements) between selective refresh and full workflow recomputation when we vary the input size. This line is approximately constant, indicating that the relationship between selective refresh and workflow recomputation is independent of input data size. Figure 6.11 shows the impact of transformation costs on the crossover point. The

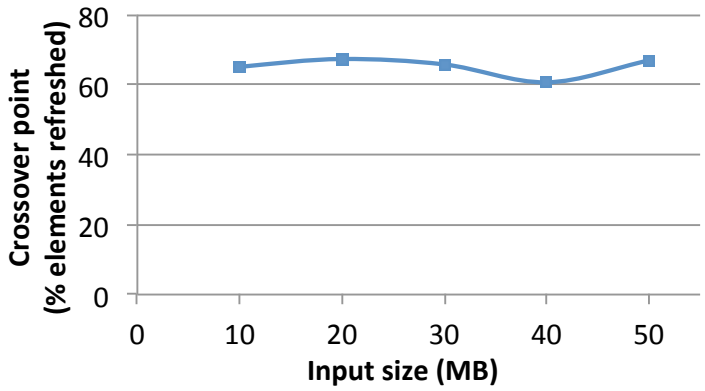

Figure 6.10: Crossover point between selective refresh and workflow recomputation, vary input size.

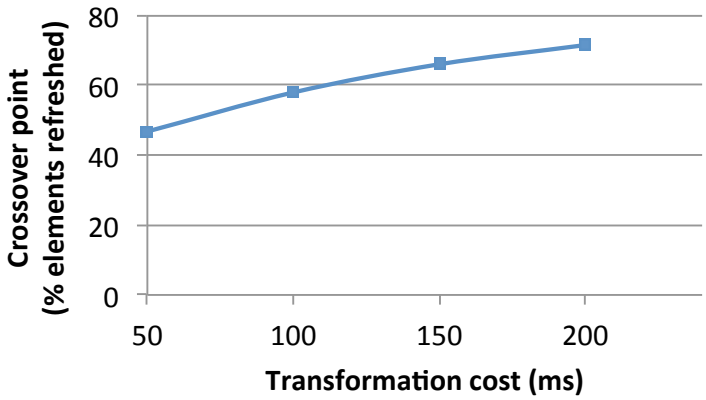

Figure 6.11: Crossover point between selective refresh and workflow recomputation, vary transformation cost.

input data size is 20 MB, and the transformation costs are varied as described above for Figure 6.8. Here the crossover point becomes more favorable for selective refresh as transformations get more expensive: we can refresh 47% at 50ms versus 71% at 200ms before refresh time exceeds recomputation. Like with provenance capture, as transformations get more expensive, we are not surprised to see a decrease in the relative cost of provenance tracing.

6.5.3 Unsafe Workflows

All of the reported experiments were conducted using our workflow example from Section 4.1.1 (Figure 4.1), which is a *safe* workflow. Recall from Section 4.6 that we propose handling unsafe workflows with a simple hybrid approach: perform workflow computation for the unsafe portion of the workflow, and selective refresh for the rest. Since this solution combines the two computations we've measured, we would expect a hybrid result: the crossover point for unsafe workflows will be less favorable towards selective refresh, with the amount of crossover "shift" determined by the percentage of the workflow that is unsafe.

6.6 Conclusions

We described the Panda system, which supports data-oriented workflows, with provenance capture and tracing. There have been multiple versions of the Panda system as we have developed our work. We gave an overview of the final version of Panda, which is based on logical provenance. We described how Panda generates logical provenance specifications and executes our tracing algorithms, and we presented some performance results for logical provenance. We described an earlier version of Panda, which we called Panda^R, that was used to experiment with provenance-based refresh. We reported experimental results based on Panda^R that consider the overhead of provenance capture, and the crossover point between selective refresh and full workflow recomputation. Even including the time overhead of provenance capture, we were able to selectively refresh over 60% of the output elements before the total running time exceeded that of recomputation.

Chapter 7

Summary and Future Work

This thesis studied the problem of managing provenance in data-oriented workflows. Overall, our work provided a comprehensive foundation, set of algorithms, and prototype system for capturing and exploiting provenance. We summarize our work in Section 7.1, and we discuss future directions in Section 7.2.

7.1 Thesis Summary

Chapter 1 gave an overview of provenance, showing through a motivating example how provenance can support the overall functions of explanation, verification, and recomputation in data-oriented workflows. Chapters 2-6 contained the main contributions of the thesis.

- **Foundations.** In Chapter 2 we laid the formal foundations for defining provenance in data-oriented workflows. We gave a new general definition of provenance, introducing the notions of correctness, precision, and minimality. After we defined workflow provenance in the intuitive recursive way, we then discussed its theoretical properties, identifying when the provenance properties of correctness, minimality, and weak correctness carry over from individual transformations to the workflow as a whole.
- **Generalized Map and Reduce Workflows.** In Chapter 3 we explored provenance for forward and backward tracing in generalized map and reduce workflows

(GMRWs). We showed that the special case of workflows where all transformations are map or reduce functions allows us to capture and exploit provenance more easily and efficiently than for general data-oriented workflows. We identified properties that hold for workflow provenance in GMRWs. We described how provenance can be captured for both map and reduce functions transparently using wrappers in the Hadoop open-source MapReduce framework. We built a prototype system as an extension to Hadoop that supports provenance capture and tracing, and we reported performance numbers on the overhead of provenance capture and the cost of provenance tracing.

- **Provenance-Based Refresh.** In Chapter 4 we considered the problem of selectively refreshing one or more output elements in a workflow where the input data sets had been modified since the workflow was run, but the workflow had not been rerun on the modified input. We presented a formal foundation and algorithms for selective refresh in workflows where each transformation has provenance captured in the form of provenance predicates. We identified properties of transformations, provenance, and workflows that are required for the algorithms to perform refresh correctly, and we discussed how the algorithms can be adapted to handle workflows that do not satisfy these properties.
- **Logical Provenance.** In Chapter 5 we considered logical provenance: provenance information stored at the transformation level. Logical provenance can often capture exactly the same element-level provenance information as physical provenance, but in a much more compact fashion. Thus, for workflows in which provenance can be captured logically without losing information, logical provenance is strictly preferable to physical provenance. We described a simple logical-provenance specification language consisting of attribute mappings and filters. We provided algorithms for backward tracing in workflows where logical provenance for each transformation is specified using our language. We also considered logical provenance specifically in the relational setting, showing that for a class of Select-Project-Join (SPJ) transformations, logical provenance specifications encode minimal provenance.

- **Panda System.** In Chapter 6 we described the Panda system, which supports data-oriented workflows, with provenance capture and tracing. There have been multiple versions of the Panda system as we have developed our work. We gave an overview of the final version of Panda, which is based on logical provenance. We described how Panda generates logical provenance specifications and executes our tracing algorithms, and we presented some performance results for logical provenance. We described an earlier version of Panda, referred to as Panda^R, that was used to experiment with provenance-based refresh. We reported experimental results based on Panda^R that consider the overhead of provenance capture, and the crossover point between selective refresh and full workflow recomputation.

7.2 Future Directions

We divide our discussion of future directions by topic. Section 7.2.1 discusses generalized map and reduce workflows, Section 7.2.2 discusses provenance-based refresh, and Section 7.2.3 discusses logical provenance.

7.2.1 Generalized Map and Reduce Workflows

- **Efficient tracing.** Our work on the RAMP system focused primarily on transparent wrapping and efficiency of provenance capture. A natural next step would be to focus on efficiency of both backward and forward tracing. For both types of tracing, building appropriate indexes is sure to be a key component.
- **Incorporating SQL processing.** A general avenue for future work is to incorporate SQL processing into GMRWs. SQL nodes interspersed with map and reduce functions can form a rich and interesting environment. Some SQL queries are map or reduce functions already, allowing them to slot right into the GMRW framework. Other SQL queries may not fit the map or reduce paradigm precisely, but do have provenance that can be incorporated via extensions to the GMRW framework. Finally, several systems (e.g., Hive [38]) compile SQL queries into

MapReduce jobs, which may provide a natural way to incorporate SQL processing into GMRWs.

7.2.2 Provenance-Based Refresh

- **Verifying requirements.** As discussed in Chapter 4, correct selective refresh requires that the transformations and workflows satisfy certain properties. Currently, workflow creators must ensure that their transformations and workflows satisfy the requirements. It may be possible to develop static and/or dynamic tests that can automatically check these requirements for some classes of workflows.
- **Integrating refresh with eager propagation.** In its current form, the workflow refresh algorithm may unnecessarily repeat work that may be shared between individual refreshes invoked during the recursion. A more sophisticated algorithm might be able to take advantage of all information obtained about new values during a refresh operation. More generally, an expanded system might perform a combination of eager and lazy refresh in response to user needs.
- **Stability guarantees.** Currently we assume that input data may change at any time, so selective refresh must always perform full backward tracing. Furthermore, we always perform forward propagation, under the assumption the data has changed. If the system monitors changes and can make guarantees about data stability—both input data and, when possible, intermediate data—we could use this information to avoid unnecessary work.
- **Special settings.** We considered a very general environment where provenance is tracked at the level of individual data elements and individual transformations. The goal was to lay the foundations for selective refresh for a wide class of data-oriented workflows. In some settings, the refresh problem might be simplified and/or made more efficient when special properties hold. For example, as discussed in Chapter 5, often schema-level provenance relationships are known for

transformations. Currently, the refresh algorithm requires schema-level provenance to be encoded as provenance predicates on each data element. This approach works, but it is unnecessarily cumbersome when all predicates take the same form. Also, as in Chapter 5, sometimes it is possible to "fold together" provenance for multiple transformations, eliminating the need for intermediate results and making refresh more efficient.

7.2.3 Logical Provenance

- **Specialized tracing algorithms.** Recall from Chapter 5 that there are a range of possible provenance-tracing algorithms for workflows with logical specifications at each transformation. 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. 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. A natural next step is to more comprehensively explore the performance tradeoffs between logical and physical provenance, which likely depend on the properties of the workflow and data.
- **Specification languages.** In Chapter 5 we presented a specific language for expressing logical provenance, based on attribute mappings and filters. It would be interesting to explore other possible logical-provenance specification languages. The language we presented was designed to be expressive yet simple, but there may exist other specification languages that work especially well for workflows in particular domains.

Bibliography

[1] `http://data.gov.uk`.

[2] `http://i.stanford.edu/panda`.

[3] `http://www.data.gov`.

[4] `http://www.heritagehealthprize.com`.

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

[6] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig: enabling database-style workflow provenance. In *VLDB*, 2012.

[7] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT*, 2010.

[8] Apache. Hadoop. *http://hadoop.apache.org/*.

[9] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[10] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. In *VLDB*, 2004.

[11] Olivier Biton, Sarah Cohen-Boulakia, Susan B. Davidson, and Carmem S. Hara. Querying and managing provenance through user views in scientific workflows. In *ICDE*, 2008.

[12] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

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

[14] Klaas Bosteels. Dumbo. *http://wiki.github.com/klbostee/dumbo/*.

[15] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *SIGMOD*, 2006.

[16] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[17] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[18] Adriane P. Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *SIGMOD*, 2008.

[19] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009.

[20] Laura Chiticariu and Wang-Chiew Tan. Debugging schema mappings with routes. In *VLDB*, 2006.

[21] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal*, 12(1), 2003.

[22] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 25(2), 2000.

[23] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, 2008.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[25] Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, 2009.

[26] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007.

[27] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, 2007.

[28] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2), 1995.

[29] Thomas Heinis and Gustavo Alonso. Efficient lineage tracking for scientific workflows. In *SIGMOD*, 2008.

[30] Jiansheng Huang, Ting Chen, Anhai Doan, and Jeffrey F. Naughton. On the provenance of non-answers to queries over extracted data. In *VLDB*, 2008.

[31] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18:1039–1065, August 2006.

[32] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. In *VLDB*, 2011.

[33] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and

Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[34] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[35] Hyunjung Park, Robert Ikeda, and Jennifer Widom. RAMP: A system for capturing and tracing provenance in MapReduce workflows. In *VLDB*, 2011.

[36] Christopher Ré and Dan Suciu. Approximate lineage for probabilistic databases. *VLDB*, 2008.

[37] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3), 2005.

[38] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. In *VLDB*, 2009.

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

[40] Yannis Velegrakis, Renee J. Miller, and John Mylopoulos. Representing and querying data transformations. In *ICDE*, 2005.

[41] Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *Managing and Mining Uncertain Data*, 2008.

[42] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, 1997.

[43] Hung-Chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-Reduce-Merge: Simplified relational data processing on large clusters. In *SIGMOD*, 2007.

[44] Mingwu Zhang, Xiangyu Zhang, Xiang Zhang, and Sunil Prabhakar. Tracing lineage beyond relational operators. In *VLDB*, 2007.