Provenance-Based Refresh in Data-Oriented Workflows[*]

Robert Ikeda, Semih Salihoglu, and Jennifer Widom
Stanford University
{rmikeda,semih,widom}@cs.stanford.edu

ABSTRACT

We consider a general workflow setting in which input data sets are processed by a graph of transformations to produce output results. Our goal is to perform efficient *selective refresh* of elements in the output data, i.e., compute the latest values of specific output elements when the input data may have changed. Our approach is based on capturing one-level *data provenance* at each transformation when the workflow is run initially. Then at refresh time provenance is used to determine (transitively) which input elements are responsible for given output elements, and the workflow is rerun only on that portion of the data needed for refresh. Our contributions are to formalize the problem setting and the problem itself, to specify properties of transformations and provenance that are required for efficient refresh, and to provide algorithms that apply to a wide class of transformations and workflows. We have built a prototype system supporting the features and algorithms presented in the paper. We report experimental results on the overhead of provenance capture, and on the crossover point between selective refresh and full workflow recomputation.

1. INTRODUCTION

Consider a *workflow* in which input *data sets* are fed into an acyclic graph of *transformations* to produce output data sets. Examples include data warehouse ETL (*extract-transform-load*) processes [16], scientific data analyses [4, 11, 15], and information extraction pipelines [6]. Suppose the input data sets have been modified since the workflow was run, but the workflow has not been rerun on the modified input (for reasons outlined below). However, suppose we would like to *selectively refresh* one or more elements in the output data, i.e., compute the latest values of particular output elements based on the modified input data.

Selective refresh can be useful in any workflow that involves expensive, batch operations, for the following reasons:

- *Eager propagation*, i.e., rerunning the workflow whenever the input changes, may be prohibitively expensive, especially if the input data is modified frequently.
- Eager propagation may be overkill: Users may not care about each input update, or about all of the output data elements.

With selective refresh, we can avoid the expense of recomputing an entire output data set frequently, when all we need is a few up-to-date output values after the input has undergone some changes.

Our challenge is to perform selective refresh efficiently in a general setting. To avoid unnecessary work during the refresh of an output element, we must understand where the element came from in the input data, and how to rerun just the relevant computation to refresh the element. At the same time, we want to support refresh for workflows constructed from a wide class of possible transformations. Thus, we do not limit ourselves to well-understood transformations such as relational queries, or specific types of data elements. Rather, we define and consider *data-oriented workflows*, where all we require is some understanding of the behavior and relationships of individual data elements constituting the inputs and outputs of each transformation.

Clearly, data *provenance* (also called *lineage*) is highly relevant to our problem, since it captures where data came from and how it was derived [1, 4, 7, 15]. There has been a large body of work in lineage and provenance over the past two decades (Section 1.1), but we are unaware of any previous work that formalizes the refresh problem or exploits provenance for selective refresh in a general workflow environment. Our contributions are as follows:

- In Section 2, we present a formal foundation for individual transformations and their provenance.
- In Section 3, we formalize the refresh problem in terms of provenance. 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, we extend our formalization and refresh procedure to data-oriented workflows. We identify an additional provenance property necessary for the "transitive" workflow refresh procedure to produce correct refreshed data.
- In Sections 5 and 6, we extend our formalism and algorithms to support transformation types that were excluded, for presentation development, from Sections 3 and 4.
- In Section 7, we discuss how refresh can still be performed (albeit less efficiently) in workflows for which the provenance property identified in Section 4 does not hold.
- In Section 8, we describe the prototype system we have built that supports refresh in data-oriented workflows. We explain the features and limitations of the current system, how the user interacts with the system, and how provenance-based refresh is supported. We report experimental results on the overhead of provenance capture, and on the crossover point between selective refresh and full workflow recomputation.

The remainder of this section covers related work and introduces a running example. In Section 10 we conclude and describe future work.

1.1 Related Work

There has been a large body of work in lineage and provenance over the past two decades. Surveys are presented in, e.g., [4, 7, 15], and formal models for provenance are presented in, e.g., [1,

[*]This work is supported by the National Science Foundation under grants IIS-0414762 and IIS-0904497.

2, 5, 9, 10, 13]. Provenance in the context of schema mappings is studied in [8, 12, 17]. None of these papers exploits provenance for selective refresh in a general workflow environment.

There also has been a large body of work in *incremental view maintenance*: the efficient propagation of base data modifications, usually in a relational setting [3, 14]. 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.

Reference [12] 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 [12] are restricted to those that can be expressed in Datalog.

1.2 Running Example

We present a running example, designed to illustrate challenges and solutions throughout the paper. Consider *Jen*, a genetic counselor who runs a workflow, shown in Figure 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 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 $\text{risk} > 0.5$, producing final output HighPatientRisks.

Figure 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 2 and 5, 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, first one-level provenance is used to trace transitively from the output element to its relevant input elements. Starting with HighPatientRisks element #2, provenance 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 supports backward-tracing in general will be presented in Sections 3–6.

(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 3–6.

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

2. FOUNDATIONS

2.1 Transformations

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 can be seen in the running example, we do not limit ourselves to transformations expressible in relational algebra or SQL. 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 6. (*Multi-output* transformations do not introduce any interesting challenges, and are thus omitted.) For any 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.

2.2 Provenance

In general a transformation may inspect the entire input data set to produce each element in the output set, but in most cases there is a more fine-grained relationship between the input and output data elements: an output data element may have been derived from a small subset of the input data elements (maybe only one). 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.

There has been a great deal of work on defining and capturing provenance for specific understood types of transformations [5, 7, 10], and some work describing properties of "opaque" transformations and how they relate to provenance [9]. Our goal in this paper is not to expand or improve upon any of that work, but rather to develop a formal framework and algorithms for the refresh problem that can exploit those definitions and techniques.

For generality, we only require that provenance for each output data element can be obtained by applying a predicate on the input:

DEFINITION 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 for instance $T(I) = O$. □

We do not impose any restrictions on provenance predicates, except that they can be applied to choose elements from data sets. Also

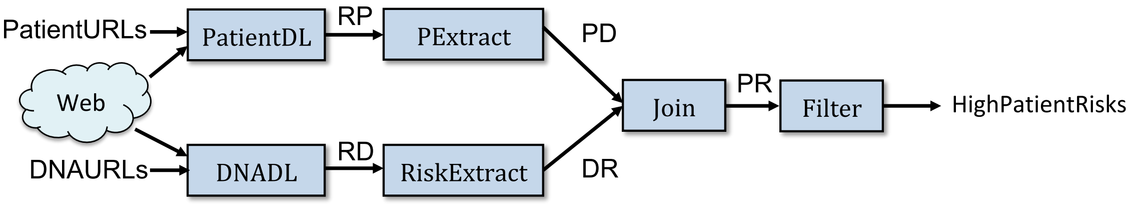

Figure 1: Genetic risk workflow example.

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’ $\wedge$ loc=1 $\wedge$ seq=‘aaa’, p^{DR}: loc=1 $\wedge$ seq=‘aaa’
2	Carl	liver	0.4	p^{PD}: name=‘Carl’ $\wedge$ loc=2 $\wedge$ seq=‘ttt’, p^{DR}: loc=2 $\wedge$ seq=‘ttt’
3	Denise	heart	0.8	p^{PD}: name=‘Denise’ $\wedge$ loc=2 $\wedge$ seq=‘aga’, p^{DR}: loc=2 $\wedge$ seq=‘aga’
4	Earl	lung	0.7	p^{PD}: name=‘Earl’ $\wedge$ loc=3 $\wedge$ seq=‘gcc’, p^{DR}: loc=3 $\wedge$ seq=‘gcc’

HighPatientRisks

	name	disease	risk	
1	Bob	heart	0.6	p: name = ‘Bob’ $\wedge$ disease = ‘heart’
2	Denise	heart	0.8	p: name = ‘Denise’ $\wedge$ disease = ‘heart’
3	Earl	lung	0.7	p: name = ‘Earl’ $\wedge$ disease = ‘lung’

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

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.

Predicates give us a very general notion of provenance, and a full treatment of how provenance predicates are obtained is beyond the scope of this paper. Note, however, that provenance in the form of pointers to specific data elements, as in [2] and other work, can be specified as predicates selecting on identifiers or keys. (In this case “evaluating a tracing query” would likely be implemented as simple pointer-chasing.) Provenance for basic relational operators [5, 7, 10] 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 [9] are amenable to the provenance-predicate approach. Although it’s possible some less conventional forms of provenance may not be captured by predicates, we believe this formalization is a useful and general starting point.

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 2.1. Our running example data in Figure 2 includes provenance predicates, denoted p. Table PatientDNA includes additional predicates labeled f, which will be explained in Section 5.

Table PatientRisks includes pairs of provenance predicates, since there are two input data sets to its transformation (Section 6). 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 the appropriate provenance for o. $\Box$

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.

3. ONE-TRANSFORMATION REFRESH

In this section, we specify a provenance-based refresh procedure for single transformations, and we identify two properties of the transformations and their provenance that are required for the procedure to work correctly. These properties also yield insights into the meaning of refresh in the presence of provenance.

But first, 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 this paper we assume refresh of a single output element produces either a single refreshed element or no element at all, but not several elements. We intend to relax this assumption in future work.) 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 3.1. Consider output directory RawDNAData in Figure 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. $\Box$

While immutable keys make for a convenient definition, unfortunately performing refresh based on immutable keys 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. For the remainder of the paper we assume each transformation T has a (possibly infinite) *input domain* $\mathbb{I}_T$, specifying T's allowable input sets.

PROCEDURE 3.1 (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. $\Box$

EXAMPLE 3.2. 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 3.1, Procedure 3.1 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. $\Box$

Example 3.2 is more efficient than Example 3.1, but does Procedure 3.1 always work? We identify two requirements on transformations and provenance under which it does.

The first restriction requires that each provenance predicate indeed captures the relevant input data subset for its output element.

REQUIREMENT 3.1 (PROVENANCE CORRECTNESS). Consider any transformation instance $T(I) = O$ for $I \in \mathbb{I}_T$, and any $\langle o, p\rangle \in O$. Then $T(\sigma_p(I)) = \{\langle o, p\rangle\}$. $\Box$

For *many-one* (and therefore *one-one*) transformations and their provenance, 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 5.

All of the transformations in our running example satisfy Requirement 3.1, with two exceptions: **PExtract** is a many-many transformation that requires the additional machinery introduced in Section 5, and **Join** is a multi-input transformation requiring some (minimal) additional machinery covered in Section 6.

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

REQUIREMENT 3.2 (PROVENANCE 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 $\langle o', p\rangle \in T(I')$. $\Box$

This requirement intuitively states that when input changes, even if there is a new value for an old output element based on its 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 as the immutable key that coordinates new and old values of output elements.

In our running example, and in other workflows we have looked at [9, 10], it is natural for transformations to output provenance predicates such that Requirement 3.2 is satisfied, and for provenance to act as an immutable key.

The following property, which follows directly from Requirement 3.1, further solidifies the key analogy by observing that provenance predicates are unique (under set semantics) within each output data set:

PROPERTY 3.1 (UNIQUE PROVENANCE). 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 $o' \neq o$. $\Box$

Now let us return to our refresh procedure and see how Requirements 3.1 and 3.2 guarantee that Procedure 3.1 works correctly, under the provenance-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 3.1 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 3.1, $T(\sigma_p(I^{new})) = \{\langle o', p\rangle\}$, contradicting the fact that $T(\sigma_p(I^{new}))$ is empty.

- Now suppose $T(\sigma_p(I^{new}))$ is non-empty. Requirement 3.2 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. REFRESH FOR WORKFLOWS

Consider any transformations T_1 and T_2. As usual, each transformation takes a data set from its domain as input, and it produces as output a data set annotated with provenance predicates. 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.

Composition is associative, so we denote the linear composition of n transformations as $T_1 \circ T_2 \circ \cdots \circ T_n$. We refer to such a composition as a *data-oriented workflow*, or *workflow* for short. In Section 6, we extend our formalism and algorithms to cover workflows where transformations may have multiple input data sets. (As mentioned in Section 2, transformations with multiple output sets do not introduce any complexities, and therefore are omitted from this paper.) Our running example in Figure 1 is a workflow composed of six transformations, with one of them taking multiple inputs.

Consider a workflow $T_1 \circ T_2 \circ \ldots \circ T_n$. A *workflow instance* is the application of the workflow to an 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 $(T_1 \circ T_2 \circ \ldots \circ T_n)(I_1) = I_{n+1}$.

Our goal is to selectively refresh an output element of an arbitrary workflow instance, when input data sets may have been modified. The following workflow refresh algorithm is a recursive extension of the single-transformation refresh Procedure 3.1.

ALGORITHM 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 $I_1^{new} \in \mathbb{I}_{T_1}$. 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 4.1. We revisit the original example from Section 1.2, 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 6), 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 1.2, 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 3 we identified properties of transformations and provenance required for the single-transformation refresh procedure to work correctly. Are the same properties sufficient for the recursive algorithm to work correctly? It turns out we need to restrict the composition of transformations and their provenance to ensure correctness; 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 4.1 (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 $\bigcup_{\langle o',p'\rangle \in \sigma_p(I_i)} T_{i-1}(\sigma_{p'}(I'_{i-1})) = \sigma_p(I'_i)$. □

With extensions to the safety definition to be introduced in Sections 5 and 6 for many-many and multi-input transformations, the workflow in our running example satisfies Requirement 4.1. However, there are some reasonable workflows where safety is not satisfied, as illustrated by the following example.

EXAMPLE 4.2. 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 generous 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 7 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 3.1, 3.2, and 4.1 all hold. The argument hinges on the following theorem.

THEOREM 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 4.1. 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 4.1, *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 4.1, 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 4.1 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 3 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 3.1 and 3.2, the arguments in Section 3 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.

5. 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 3.1). 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 5.1. 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 3.1 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 5.1, 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 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 paper thus far.) To refresh an element $\langle o, p, f\rangle \in O$, we add a third step to Procedure 3.1 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 3 and 4 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 3.2 (*Provenance 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. Full details of the extension for many-many transformations are given in Appendix A.

6. 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 paper 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 6.1. From our running example, to refresh (Carl,liver,0.4) (element #2) in PatientRisks, provenance predicate p^{PD} (name='Carl' $\wedge$ loc=2 $\wedge$ seq='ttt') is applied on PatientDNA to obtain element #2, and provenance predicate p^{DR} (loc=2 $\wedge$ 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 3–5 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 3.2 (*Provenance as Key*), we now treat the entire combination of $p_1, \ldots, p_m, f$ as the immutable key. In Requirement 4.1 (*Workflow Safety*), we require each transformation to be safe with respect to all of its predecessor transformations in concert. Full details are given in Appendix B.

7. UNSAFE WORKFLOWS

In Section 4 we introduced *safe workflows* (Requirement 4.1), and our refresh algorithms thus far require workflow safety. For now, 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.

8. SYSTEM PROTOTYPE

We have built a prototype system, which we call "Panda", that implements all features and algorithms presented in this paper. For example, the Panda system easily supports the workflow in our running example. For the time being, all data sets handled by Panda are encoded in relational tables, but as we have seen, our formal underpinnings and algorithms do not rely on the relational model.

The high-level architecture of the Panda system is shown in Figure 3. For now the main backend is a **SQLite** server, storing all data sets, relational (SQL) transformations, provenance, and workflow information. Panda also supports "opaque" transformations programmed in Python; they are stored separately in files.

Users interact with Panda through a simple command-line interface; we intend to build a GUI in the near future. There are three 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) Refresh commands. The *Panda Layer* processes all user commands: It stores workflow graphs and their transformations, creates and maintains auxiliary provenance tables, generates provenance predicates and forward filters for output elements, and runs the refresh algorithms.

In the remainder of this section we briefly discuss how Panda handles transformations specified in Python, transformations specified as SQL queries, and finally Refresh commands.

8.1 Python Transformations

Panda currently supports Python transformations that output provenance predicates with each output element. As an example, consider adding transformation **PatientDL** as we build up our running example workflow (Figure 1). The user writes a Python script

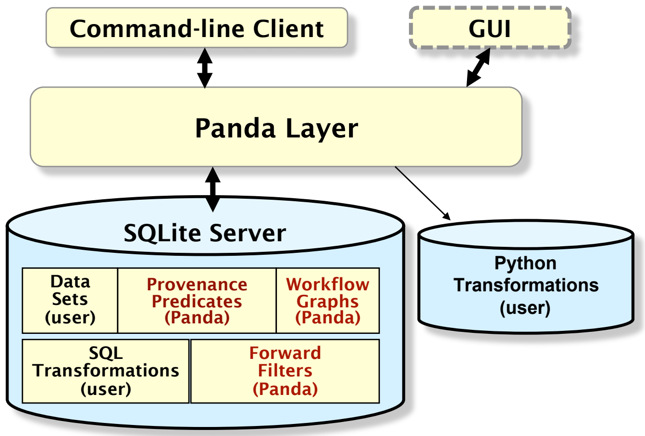

Figure 3: Architecture of the prototype Panda system.

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 Dataset RawPatientData
As Python 'patientdl.py' on PatientURLs
```

In response to this command, the Panda Layer: (1) Creates the new data set RawPatientData; (2) Inserts a record into the Workflow table connecting PatientURLs to RawPatientData via Python script **patientdl.py**; (3) Creates a provenance-predicate table RawPatientData_PP for RawPatientData; (4) Runs **patientdl.py** on PatientURLs, inserting the resulting elements and corresponding provenance-predicate records into tables RawPatientData and RawPatientData_PP, respectively.

For many-many transformations, the Python script also needs to create a forward filter for each output element. Then, step (3) above also creates a forward-filter table, and step (4) also inserts a record for the forward filter.

8.2 SQL Transformations

Panda also supports transformations specified as SQL queries, including queries/transformations involving multiple input tables. Provenance predicates are created automatically for SQL transformations, following known definitions and techniques [5, 7, 10]: 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 6, again relying on declared keys. Finally, **Group-by** queries generate provenance predicates based on the grouping attribute(s).

The command to create a new SQL transformation is similar to the command shown in Section 8.1, except **As** is followed by a SQL query, whose **From** clause must refer to already-defined tables. The steps performed by the Panda Layer are also similar to those outlined in Section 8.1; forward filters are never needed since SQL queries cannot produce many-many transformations.

8.3 Refresh

When workflows are created and run, Panda stores everything needed to support selective refresh: provenance predicates and intermediate data sets for backward-tracing; transformations and forward filters for forward-propagation. Panda assumes that all transformations, provenance, and workflows satisfy the requirements

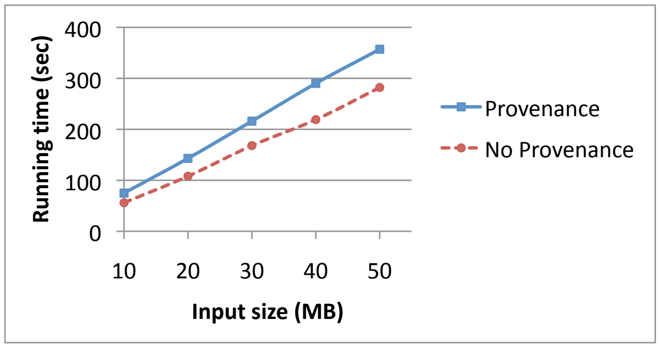

Figure 4: Time overhead of provenance capture, vary input size.

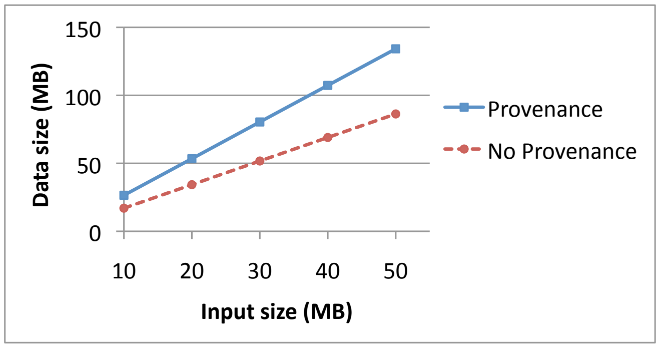

Figure 5: Space overhead of provenance capture, vary input size.

specified in this paper. Automatically detecting when the requirements are satisfied—particularly the most interesting requirement of workflow safety—is an important area of future work.

Under the assumption of all requirements being satisfied, Panda supports selective refresh using the exact algorithms given in this paper. When an output element o is refreshed, if a new value o' is produced then Panda 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.

9. EXPERIMENTS

The primary goal of our 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?

While eventually we plan to experiment with a suite of workflows, for this empirical study we used our running example workflow described in Section 1.2, 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 our experiments were run using the Panda system (Section 8) on a MacBook Pro laptop (2.4 GHz Intel Core 2 Duo, 4 GB memory, 150 GB storage, Mac OS X 10.4). 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.

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 30% time overhead and 56% space overhead for various input sizes to capture provenance while executing the workflow.
- 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 52% and 70% 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.

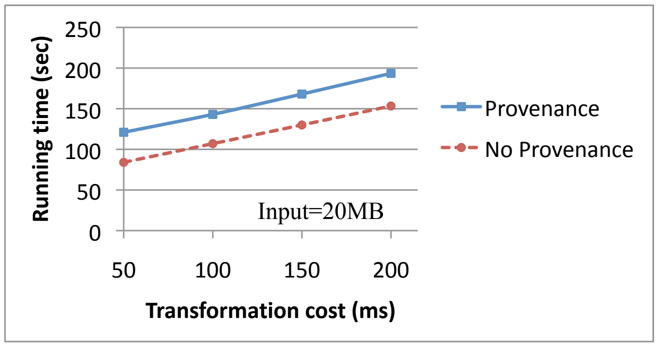

Figure 6: Time overhead of provenance capture, vary transformation cost.

9.1 Overhead of Provenance Capture

Figure 4 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 27% to 34%

Figure 5 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.

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 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 42% at 50ms vs. 26% at 200ms. Intuitively, when transformations are more expensive, provenance capture plays a smaller role.

9.2 Crossover Point for Refresh

Our next experiments identify the crossover point between selective refreshes (including provenance capture) and full recompu-

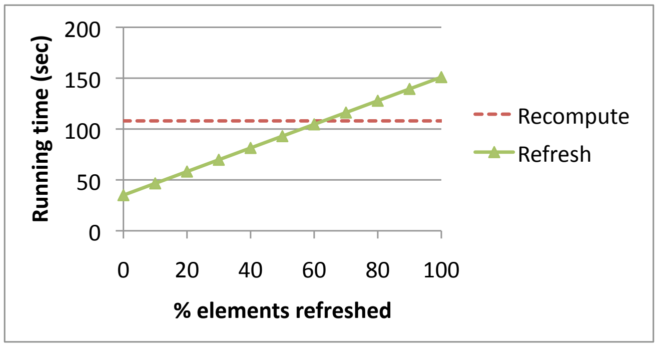

Figure 7: Recomputation and refresh costs.

tation (without provenance capture). For starters, the two lines in Figure 7 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 7 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 8 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 9 shows the impact of transformation costs on the crossover point. The input data size is 20 MB, and the transformation costs are varied as described above for Figure 6. Here the crossover point becomes more favorable for selective refresh as transformations get more expensive: we can refresh 52% at 50ms versus 70% 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.

9.3 Unsafe Workflows

All of the reported experiments were conducted using our running example, which is a *safe* workflow. Although we have not yet run experiments on unsafe workflows, until we develop more sophisticated techniques for handling them, we don't expect any significant surprises or insights. Recall from Section 7 that currently 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.

10. CONCLUSIONS AND FUTURE WORK

We presented a formal foundation and algorithms for efficient

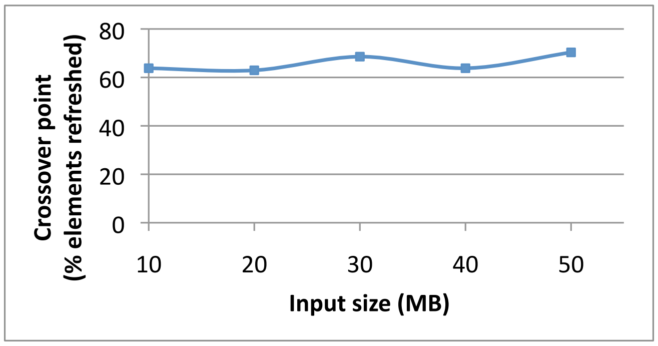

Figure 8: Crossover point between selective refresh and workflow recomputation, vary input size.

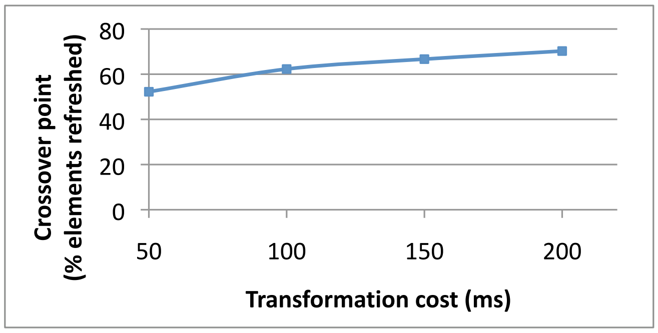

Figure 9: Crossover point between selective refresh and workflow recomputation, vary transformation cost.

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. We described the prototype system we have built that supports the features and algorithms presented in the paper, and we reported some experimental results. There are several directions for future work:

- **Verifying requirements:** Currently we rely on workflow creators to ensure that their transformations and workflows satisfy the properties required for correct refresh. We would like to develop static and/or dynamic tests to check the requirements for a given workflow. We also intend to explore more precisely the general classes of transformations, provenance, and workflows that are guaranteed to satisfy the requirements.
- **Relaxing requirements:** We would like to offer solutions when some of the requirements and assumptions made in this paper are not satisfied. For example, the rudimentary solution offered in Section 7 for unsafe workflows can probably be improved upon, and we don't yet have a formalism and algorithms that capture the case when refreshing a single value logically results in multiple values.
- **Automatic generation of provenance predicates:** As discussed in Section 2.2, there has been a large body of work on generating provenance predicates automatically for relational transformations [5, 7, 10]. However, automatic or semi-automatic generation of provenance predicates for general transformations is an interesting, largely open problem.
- **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. We hope to come up with an algorithm that can take advantage of all information obtained about new values during a refresh operation. More generally, we plan to expand our prototype into a system that can 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 can use this information to avoid unnecessary work.
- **Special settings:** In this paper we have considered a very general environment where provenance is tracked at the level of individual data elements and individual transformations. Our 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, often schema-level provenance relationships are known for transformations [9]. Currently, we require 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, with additional information, sometimes it is possible to "fold together" provenance for multiple transformations [9], eliminating the need for intermediate results and making refresh more efficient.

11. REFERENCES

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

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

[3] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

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

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

[6] C.-H. Chang, M. Kayed, M. R. Girgis, and K. F. Shaalan. A survey of web information extraction systems. *IEEE Trans. on Knowl. and Data Eng.*, 18(10), 2006.

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

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

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

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

[11] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, 2008.

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

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

[14] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2), 1995.

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

[16] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Conceptual modeling for ETL processes. In *DOLAP*, 2002.

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

APPENDIX

A. MANY-MANY TRANSFORMATIONS

A.1 One-Transformation Refresh

PROCEDURE A.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. $\square$

REQUIREMENT A.1 (PROVENANCE CORRECTNESS). Consider any transformation instance $T(I) = O$ for $I \in \mathbb{I}_T$, and any $\langle o, p, f \rangle \in O$. Then $\sigma_f(T(\sigma_p(I))) = \{\langle o, p, f \rangle\}$. $\square$

REQUIREMENT A.2 (PROVENANCE 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 $\langle o', p, f \rangle \in T(I')$. $\square$

PROPERTY A.1 (UNIQUE PROVENANCE). 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 $o' \neq o$. $\square$

We can show in a similar manner as we did at the end of Section 3 that Procedure A.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 A.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 A.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.

A.2 Workflow Refresh

ALGORITHM A.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 A.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 A.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 A.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 $S' = \bigcup_{\langle o_i,p_i,f_i\rangle \in S}$ *workflow_refresh*$(\langle o_i,p_i,f_i\rangle)$. By the inductive hypothesis, each *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 A.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. □

B. MULTI-INPUT TRANSFORMATIONS

B.1 One-Transformation Refresh

PROCEDURE B.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 B.1 (PROVENANCE 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 B.2 (PROVENANCE 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 B.1 (UNIQUE PROVENANCE). 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 3 that Procedure B.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 B.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 B.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.

B.2 Workflow Refresh

ALGORITHM B.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 a base data set **then** $S'_j = \sigma_{p^j}(I_j^{i\ new})$
        **else** { $S_j = \sigma_{p^j}(I_j^i)$;
            $S'_j = \bigcup_{\langle o',\bar{p}',f'\rangle \in S_j}$ *workflow_refresh*$(\langle o',\bar{p}',f'\rangle)$}
    **return** $\sigma_f(T_i(S'_1,\ldots,S'_{m_i}))$ }
```

REQUIREMENT B.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 $O_i = T_i(I_1^i,\ldots,I_{m_i}^i)$ 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 $(I_1'^j,\ldots,I_{m_j}'^j) \in \mathbb{I}_{T_j}$. Let $O'_j = T_j(I_1'^j,\ldots,I_{m_j}'^j)$. 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 $\bigcup_{\langle o',\bar{p}',f'\rangle \in \sigma_{p^j}(O_j)} \sigma_{f'}(T_j(\sigma_{p'_1}(I_1^{'j}), \ldots, \sigma_{p'_{m_j}}(I_{m_j}^{'j}))) = \sigma_{p^j}(O'_j)$. □

THEOREM B.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 B.3. Given any element $\langle o, (p_1, \ldots, p_{m_i}), f\rangle \in O_i$ for $i \geq 1$, *workflow_refresh*$(\langle o, (p_1, \ldots, p_{m_i}), f\rangle) = \sigma_f(T_i(\sigma_{p_1}(I_1^{i\ new}), \ldots, \sigma_{p_{m_i}}(I_{m_i}^{i\ new})))$.

Proof. Given i, let $l(i)$ be the length of the longest path from O_i to base data: if T_i has base 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_j^i are base data sets, $S'_j = \sigma_{p^j}(I_j^{i\ new})$. We return $\sigma_f(T_i(S'_1, \ldots, S'_{m_i}))$ = $\sigma_f(T_i(\sigma_{p_1}(I_1^{i\ new}), \ldots, \sigma_{p_{m_i}}(I_{m_i}^{i\ new})))$, 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_j^i is a base data set, $S'_j = \sigma_{p_j}(I_j^{i\ new})$. Else, *workflow_refresh* computes the following two sets: $S_j = \sigma_{p^j}(I_j^i)$ and $S'_j = \bigcup_{\langle o',\bar{p}',f'\rangle \in S_j}$ *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 B.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. □