Provenance for Generalized Map and Reduce Workflows*

Robert Ikeda, Hyunjung Park, and Jennifer Widom
Stanford University
{rmikeda,hyunjung,widom}@cs.stanford.edu

ABSTRACT

We consider a class of workflows, which we call *generalized map and reduce workflows* (*GMRWs*), where input data sets are processed by an acyclic graph of *map* and *reduce* functions to produce output results. We show how *data provenance* (also sometimes called *lineage*) can be captured for map and reduce functions transparently. The captured provenance can then be used to support *backward tracing* (finding the input subsets that contributed to a given output element) and *forward tracing* (determining which output elements were derived from a particular input element). We provide formal underpinnings for provenance in GMRWs, and we identify properties that are guaranteed to hold when provenance is applied recursively. We have built a prototype system that supports provenance capture and tracing as an extension to Hadoop. Our system uses a wrapper-based approach, requiring little if any user intervention in most cases, and retaining Hadoop's parallel execution and fault tolerance. Performance numbers from our system are reported.

1. INTRODUCTION

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. Such workflows are common in, e.g., scientific data processing [8, 14, 23] and information extraction [22]. A special case of such workflows is what we refer to as *generalized map and reduce workflows* (*GMRWs*), in which all transformations are either *map* or *reduce* functions [3, 15]. 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 data-oriented workflows, it can be useful to track *data provenance* (also sometimes called *lineage*), capturing how data elements are processed through the workflow [1, 8, 11, 23]. Provenance supports *backward tracing* (finding the input subsets that contributed to a given output element) and *forward tracing* (determining which output elements were derived from a particular input element). Backward tracing can be useful for, e.g., debugging and drilling-down, while forward tracing can be useful for, e.g., tracking error propagation. In addition, provenance can form the basis for *incremental maintenance* [16] and *selective refresh* [18].

*This work was supported by grants from the National Science Foundation (IIS-0904497) and Amazon Web Services.

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2011.
5th *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

In this paper, 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 define, 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 [3].

There has been a large body of work in provenance, including for general workflows (Section 1.1). 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. Provenance can be defined naturally for individual map and reduce functions, but it turns out to be a challenge to identify properties that hold when one-level provenance is applied recursively through a workflow. Also, 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 in Section 2, in Section 3 we define provenance for individual map and reduce functions. We then identify properties that hold when one-level provenance is applied recursively through a GMRW.
- Section 4 describes how provenance can be captured and stored during workflow execution, and it specifies backward and forward tracing procedures using provenance.
- We have built a system called *RAMP* (*Reduce And Map Provenance*) that implements the concepts in this paper. Section 5 describes the implementation of RAMP as an extension to Hadoop. RAMP wraps Hadoop components automatically, requiring little if any user intervention, and retaining Hadoop's parallel execution and fault tolerance.
- Section 6 reports performance results using RAMP on the time and space overhead of capturing provenance, and discusses the cost of provenance tracing in our current system.

In Section 7, we conclude and discuss future work, including how we can incorporate SQL processing into GMRWs with provenance.

1.1 Related Work

Obviously there has been tremendous interest recently in high-performance parallel data processing specified via map and reduce functions, e.g., [3, 15, 27]. In addition, higher-level platforms have been built on top of these systems to make data-parallel programming easier, e.g., [9, 20, 24]. 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, there has been a large body of work in lineage

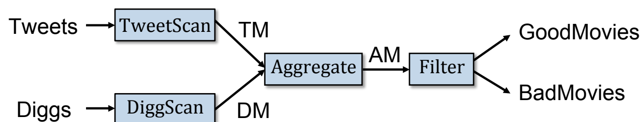


Figure 1: Movie sentiment workflow example.

and provenance over the past two decades, surveyed in, e.g., [8, 11, 23]. Provenance specifically in the data-oriented workflow setting is considered by [1, 2, 7, 10, 12, 17, 18, 19, 26], among others. However, 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 paper.

Reference [12]—our own work from the distant past—is perhaps most related. It 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 [12] allows acyclic graphs of transformations, it does not investigate behavioral properties when provenance is traced recursively through them. We show in this paper that recursive provenance tracing can yield ill-behaved results in certain subtle cases. Finally, in this paper we consider the overhead incurred gathering extra information during workflow execution to facilitate provenance tracing, a topic not considered in [12].

1.2 Running Example

As a simplified example GMRW that serves primarily to illustrate our definitions and techniques, consider the workflow shown in Figure 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 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*.

2. TRANSFORMATIONS & WORKFLOWS

Let a *data set* be any set of data *elements*. We assume every element has a unique identifier (discussed later). Thus, there are no duplicates in any 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. A *workflow* is a directed acyclic graph, where nodes are transformations, and each edge is annotated with a data set.

In *generalized map and reduce workflows*, the two types of transformations are *map functions* and *reduce functions*. For now, we consider map and reduce functions with just one input set and one output set; Section 2.2 explains how multiple input and output sets are handled.

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 this paper, 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 paper, we use $G_1, .., G_n$ to denote the key-based *groups* of a reduce function's input set I.

2.1 Transformation Properties

We now list some properties 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 this paper.

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*. 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*. In our running example, **Aggregate** is many-one.

Monotonicity. We say that a transformation T is *monotonic* if for any input sets I and I' with $I \subseteq I'$, then $T(I) \subseteq T(I')$. 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.

A thorough analysis of transformation properties in the context of provenance was developed in [12]. When we place the map and reduce functions we consider into the hierarchy of that paper, our provenance definitions (Section 3) are consistent with the definitions in [12].

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

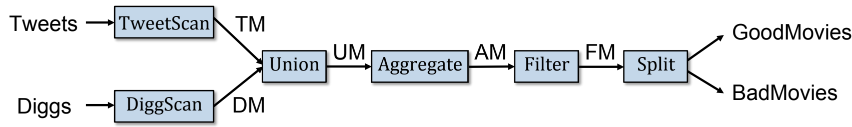


Figure 2: Movie workflow example with union and split.

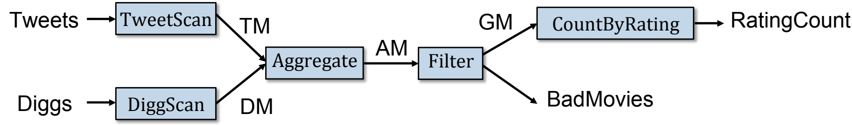


Figure 3: Modified movie workflow example with ill-behaved provenance.

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 analysis in Section 3, 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 their 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 $O_1 \cup \ldots \cup O_r = I$. 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.

Recall we assume all of our data sets have unique identifiers. We further assume identifiers are made globally unique, so $\cup$ in the above definitions is always disjoint union. Figure 2 adds union and split transformations to our running example.

3. PROVENANCE

Given a transformation instance $T(I) = O$ for a given input set I, and an output element $o \in O$, *provenance* should identify the input subset $I^* \subseteq I$ containing those elements that contributed to o's derivation. First we define provenance for each transformation type, then we show how this "one-level" provenance is used to derive workflow provenance.

Provenance for single transformations is straightforward and intuitive:

- **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 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 $i = o$.

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

Now suppose we have a GMRW, and we would like the provenance of an output element in terms of the initial inputs to the workflow. For our recursive definition, we more generally define the provenance of any data element involved in the workflow—input, intermediate, or output.

DEFINITION 3.1 (GMRW PROVENANCE). Consider a GMRW W with initial inputs $I_1, \ldots, I_m$ and any data element e. The 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 $P_W(e) = \{e\}$. Otherwise, let T be the transformation that output e. Let $P_T(e)$ be the one-level provenance of e with respect to T as defined above. Then $P_W(e) = \bigcup_{e' \in P_T(e)} P_W(e')$. □

Having defined GMRW provenance in the intuitive way, we would like to make sure it gives us something meaningful. Specifically, we desire the following "replay" property.

PROPERTY 3.1 (REPLAY PROPERTY). Consider an output element o, and let $P_W(o) = (I_1^*, \ldots, I_m^*)$ be the provenance of o in workflow W. If we run $I_1^*, \ldots, I_m^*$ through W, denoted $W(I_1^*, \ldots, I_m^*)$, then o is part of the result: $o \in W(I_1^*, \ldots, I_m^*)$. □

The replay property holds for our running example and for a very large class of GMRWs, but unfortunately it does not hold all the time. Suppose our running example is changed in the following two ways (shown in Figure 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.

Based on Definition 3.1, for the output element o = (rating:8,count:1) in RatingCount, we get $P_W(o) = \{t_1\}$, which agrees with our intuition since t_1 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 an element e = (*Twilight*,8) that is irrelevant to the provenance of the output element we're interested in. Such extraneous elements can be produced only by one-many map or many-many reduce functions. 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 workflow on the provenance of an output element o is not guaranteed to produce o. We prove the following theorem in Appendix A.

THEOREM 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 $o \in W(I_1^*, \ldots, I_m^*)$. □

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.

For workflows not satisfying Theorem 3.1, the recursive definition of provenance still yields an intuitive result. 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. (Provenance-based *selective refresh* [18], comprised of backward tracing followed by forward propagation, requires a similar property. Our approach to selective refresh for arbitrary workflows in [18] would deem this example workflow "unsafe" and disallow it.) 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, proved in Appendix B.

COROLLARY 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.[1] Then $o \in W^*(P_W(o))$. □

4. PROVENANCE CAPTURE & TRACING

In this section we describe, at an abstract level, how provenance according to our definitions can be captured and stored during workflow execution. We also give algorithms for forward and backward tracing. The next section will give details of our actual implementation as an extension to Hadoop. For now let us assume that all input, intermediate, and output data sets are persistent, although we will see in Section 5 that our Hadoop implementation discards certain intermediate data sets.

[1]We assume all intermediate/output data sets have been stored for provenance-tracing purposes; see Section 4.

Capturing and storing provenance according to our definitions is straightforward: For map functions, we extract the unique identifier (ID) of each input element that produces one or more output elements, and we add that ID to each of the output elements. For reduce functions, we keep track of the grouping key for each input group, and we add that key to each output element produced by the group. In Section 5, we describe in more detail how our Hadoop implementation wraps map and reduce functions automatically to emit these extra fields during execution. Recall that we introduced union and split operations in Section 2.2 for the purposes of analysis. In reality, these operations are incorporated into map and reduce functions and do not affect our capture and storage scheme.

Now consider backward tracing. The following algorithm implements the recursive definition of provenance given in Section 3.

ALGORITHM 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;
  if T is a map function
    then $E' \leftarrow$ input elements to T with ID that annotates an element in E;
  if T is a reduce function
    then $E' \leftarrow$ input elements to T with grouping key that annotates an element 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^*; }}

For forward tracing, the overall algorithm is simply the converse of backward tracing:

ALGORITHM 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;
  if T is a map function
    then $E' \leftarrow$ output elements from T with ID corresponding to an element in E;
  if T is a reduce function
    then $E' \leftarrow$ output elements from T with grouping key 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^*;}

5. RAMP SYSTEM

We have built a system called *RAMP* (*Reduce And Map Provenance*) implementing provenance for GMRWs as described in this paper. RAMP is built as an extension to Hadoop [3]. Because the basic execution unit in Hadoop is a *MapReduce job* consisting of one map function followed by one reduce function, in the workflows supported by RAMP, each transformation is a MapReduce job. Specifically, a GMRW is implemented as a *MapReduce workflow* in RAMP:

1. If the GMRW contains a map function followed by a reduce function, the two transformations are treated as a single MapReduce job in RAMP. In particular, no intermediate data is stored between the map and reduce functions.
2. Map functions in the GMRW that are not followed by a reduce function are treated as a MapReduce job without a reduce function.
3. Reduce functions in the GMRW that are not preceded by a map function are treated as a MapReduce job with the identity map function.

In other respects, RAMP captures and traces provenance as discussed in Section 4.

RAMP's approach to provenance capture is wrapper-based and transparent to Hadoop, retaining Hadoop's parallel execution and fault tolerance. Furthermore, users need not be aware of provenance capture while writing MapReduce jobs—wrapping is automatic, and RAMP stores provenance separately from the input and output data.

When input and output data sets are stored in files, RAMP provides efficient default schemes for assigning element IDs and storing provenance; these schemes are described in Section 5.3. For other settings, RAMP allows users to define custom ID and storage schemes.

We discuss how RAMP performs provenance capture and tracing in Sections 5.1 and 5.2, respectively. These sections do not assume RAMP's default implementation for file input and output, which is discussed in Section 5.3.

5.1 Provenance Capture

Recall from Section 4 that all intermediate data between transformations is stored, and provenance is captured one transformation at a time. As described above, in the MapReduce workflows supported by RAMP, each transformation is a MapReduce job. Thus, this section describes how provenance is captured for a single MapReduce job. Section 5.2 explains how RAMP uses the captured provenance to support provenance tracing through arbitrary workflows comprised of multiple MapReduce jobs.

In Hadoop, all data elements are assumed to be key/value pairs. When running a MapReduce job consisting of a map function and a reduce function, the map output elements are grouped by their key before being processed by the reduce function. Otherwise, keys are simply part of the data.

Hadoop users supply the following five components to define a MapReduce job [5]:

- *Record-reader:* Reads the input data and parses it into input key/value pairs for the mapper.
- *Mapper:* Defines the map function.
- *Combiner:* Defines partial aggregation by key (optional).
- *Reducer:* Defines the reduce function.
- *Record-writer:* Writes output key/value pairs from the reducer in a specified output format.

RAMP implements the provenance capture scheme from Section 4 by wrapping all of these components. For presentation purposes, we consider MapReduce jobs without a combiner; the extension for combiners is straightforward. We also assume all MapReduce jobs do have a reducer; RAMP's extension for map-only jobs is similarly straightforward.

5.1.1 Map functions

For map functions, RAMP adds to each map output element a unique ID for the input element that generated the output element. Specifically, RAMP annotates the *value* part of the map output element, allowing Hadoop to correctly group the map output elements by key for the reduce function.

The following procedure specifies how RAMP wraps the record-reader and mapper to perform ID annotation (Figure 4).

PROCEDURE 5.1 (WRAPPING A MAP FUNCTION).

1. The record-reader assigns a unique ID p to the input element (k^i, v^i) that it emits.
2. The record-reader's wrapper emits $(k^i, \langle v^i, p\rangle)$.
3. The mapper's wrapper takes $(k^i, \langle v^i, p\rangle)$ as input and feeds (k^i, v^i) to the mapper.
4. For each mapper output (k^m, v^m), the mapper's wrapper emits $(k^m, \langle v^m, p\rangle)$. □

No provenance is actually stored at this point; provenance storage is performed by the wrapped reducer and record-writer, as explained next.

5.1.2 Reduce functions

For reduce functions, RAMP stores the reduce provenance as a mapping from a unique ID for each output element (k^o, v^o) to the grouping key k^m that produced (k^o, v^o). It simultaneously stores the map provenance as a mapping from the grouping key k^m to the input element ID p_j's. By storing map provenance after the map output elements have been grouped, RAMP allows all input element IDs corresponding to the same grouping key to be stored together. Since the grouping key k^m merely joins the map and reduce provenance, k^m is replaced with an integer ID k^m_{ID}.

The following procedure describes how RAMP wraps the reducer and record-writer (Figure 5).

PROCEDURE 5.2 (WRAPPING A REDUCE FUNCTION).

1. The reducer's wrapper iterates over all annotated map output elements $(k^m, \langle v^m_j, p_j\rangle)$'s with the same key k^m and feeds each (k^m, v^m_j) to the reducer.
2. While feeding the reducer, the reducer's wrapper stores the map provenance (k^m_{ID}, p_j)'s.
3. For each reducer output (k^o, v^o), the reducer's wrapper emits $(k^o, \langle v^o, k^m_{ID}\rangle)$ to the record-writer's wrapper.
4. The record-writer's wrapper takes $(k^o, \langle v^o, k^m_{ID}\rangle)$ as input and feeds (k^o, v^o) to the record-writer.
5. The record-writer assigns a unique ID q to the output element (k^o, v^o) that it writes.
6. The record-writer's wrapper stores the reduce provenance (q, k^m_{ID}). □

An alternative scheme for provenance would be to store the input element ID p_j's directly for each output element ID. However, this scheme wastes space when the reduce function is many-many. Moreover, we cannot implement this scheme efficiently in Hadoop: we would need to collect all ID p_j's in advance by iterating over

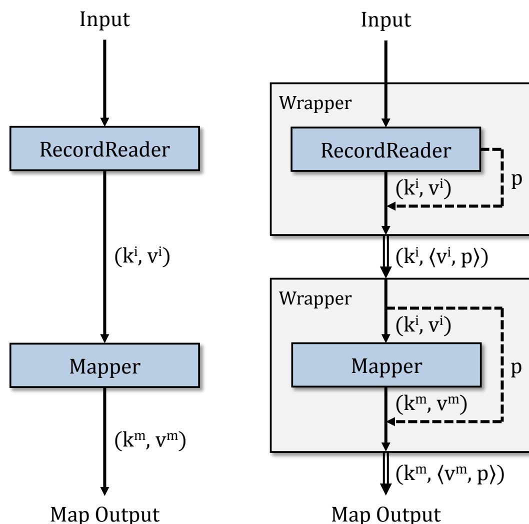


Figure 4: Wrapping a map function with RAMP.

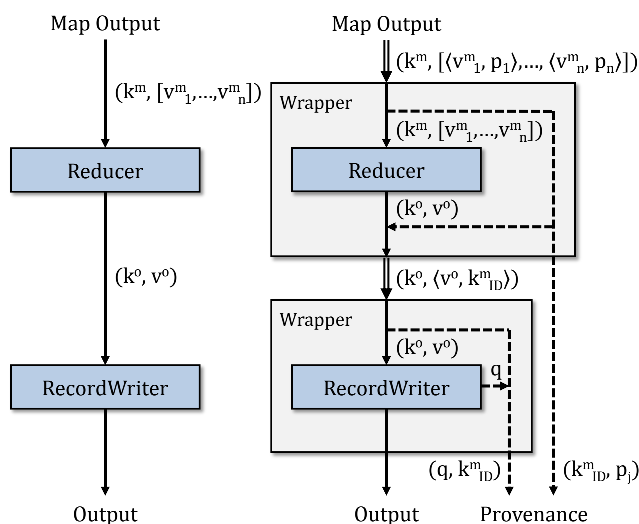


Figure 5: Wrapping a reduce function with RAMP.

all map output elements $(k^m, \langle v_j^m, p_j \rangle)$'s because the reducer can produce an output element (k^o, v^o) before all ID p_j's are seen by our wrapper. In contrast, RAMP stores the map and reduce provenance independently, joining them later during provenance tracing. One disadvantage of RAMP's scheme is that extraneous provenance data may be written if the reduce function does not produce any output elements for a particular grouping key.

5.2 Provenance Tracing

Implementing the *backward_trace* function of Algorithm 4.1 in the RAMP system is fairly straightforward, although our tracing scheme has not yet been made as efficient as possible.

Since each RAMP transformation is a MapReduce job as described above, a single backward tracing step for one output element proceeds as follows:

1. Given an output element ID q, RAMP accesses the reduce provenance as specified in the previous section to determine the corresponding grouping key ID k_{ID}^m.
2. Using k_{ID}^m, RAMP accesses the map provenance as specified in the previous section to retrieve all relevant input element ID p_j's.

The IDs returned by step 2 can either be used to fetch actual data elements, or they can be fed to recursive invocations of backward tracing until the initial input data sets are reached. Our current implementation traces recursively one element at a time, however we would expect efficiency to improve in some cases by tracing multiple elements together.

Notice that RAMP's provenance capture scheme is biased towards backward tracing, which we assume is a more frequent operation than forward tracing. In the forward-tracing setting, we are given a set of input element IDs, and we need to find all output elements that corresponds to these input element IDs. Without auxiliary structures, each forward-tracing step would require a complete scan of the map provenance, which is not sorted on input element IDs. Thus, as a first step to facilitate forward tracing, we certainly need to build indexes on the input element ID field. As will be seen in Section 6, our performance experiments have also focused on backward tracing. Enabling efficient forward tracing and measuring its performance is a next step in the RAMP system.

5.3 Data Sets in Files

Consider the specific workflow setting where all input, intermediate, and output data sets are stored in files, as is typical using Hadoop. In this setting, RAMP uses (*filename*, *offset*) as a default unique ID for each data element. For input element IDs, RAMP maps each filename to an integer ID, maintaining a dictionary with the actual filenames. For output element IDs, RAMP is able to omit the filename and use the offset alone: each provenance data file is associated with a particular output data file. Variable-length encoding for integers allows RAMP to implement this element ID scheme efficiently in terms of space overhead.

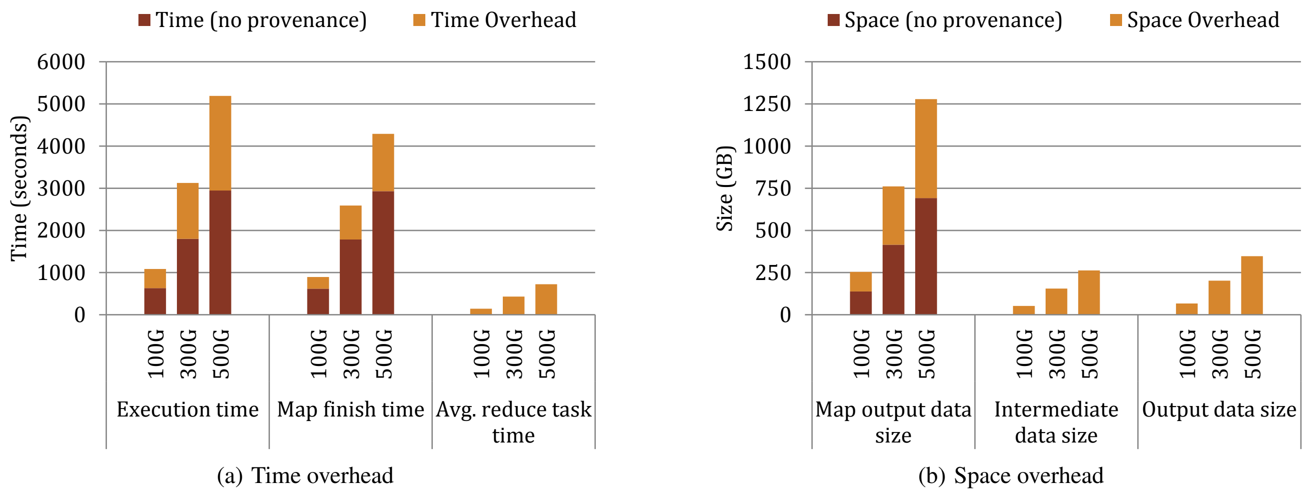
Notice that our scheme enables efficient backward tracing without special indexes: Output element IDs, which are the element's offset in the output data file, increase as each output element is appended. Thus, reduce provenance is automatically stored in ascending key order. Exploiting this order, RAMP performs binary search on the provenance data during backward tracing.

6. EXPERIMENTAL EVALUATION

We present performance experiments conducted using RAMP on two MapReduce workflows, each consisting of a single MapReduce job. (Since our experiments are focused more on capture than on tracing, a single MapReduce job is sufficient.) The two MapReduce jobs in our experiments were *Wordcount* and *Terasort* [21], included in the Hadoop distribution. Wordcount counts the number of occurrences of each word in a set of input text files; we used 100, 300, and 500 GB of input text generated randomly from 8000 distinct English words. Terasort sorts 100-byte records stored in files; we used 10^9, 3×10^9, and 5×10^9 random records as input (93, 279, and 466 GB respectively).

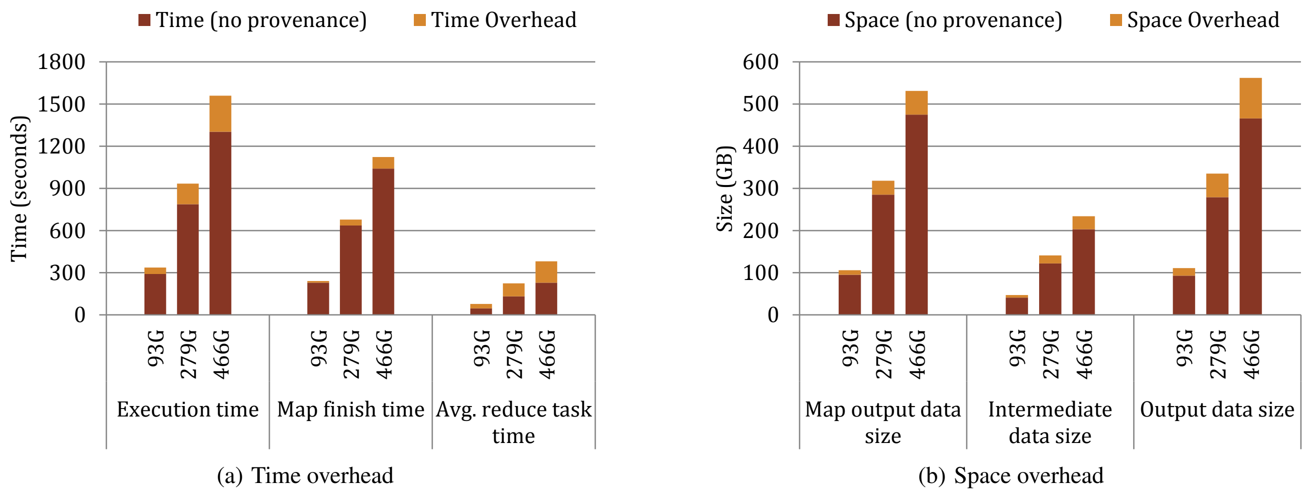
Note that Wordcount and Terasort are very different in terms of multiplicity. Wordcount is a many-one transformation, with a huge fan-in for large input data sizes. On the other hand, Terasort is a one-one transformation. We discuss how the multiplicity affects the time and space overhead of provenance capture in Section 6.1.

The cluster we used for our experiments consisted of 51 large Amazon Elastic Compute Cloud (EC2) instances, each with 7.5 GB memory, two virtual cores with 2 EC2 Compute Units each, and 850 GB instance storage. We launched all instances with 64-bit



(a) Time overhead (b) Space overhead

Figure 6: Overhead of provenance capture in Wordcount.



(a) Time overhead (b) Space overhead

Figure 7: Overhead of provenance capture in Terasort.

Amazon Linux AMI and installed Java 1.6.0_22 and the modified version of Hadoop 0.21.0. One instance served as the master node and acted as both *name node* and *job tracker*; the other 50 instances served as slave nodes. Each slave node was allowed to run two map tasks and two reduce tasks concurrently, and the number of reduce tasks was set to 100. We configured Hadoop following the guidelines for real-world cluster configurations [4]. The changes from the default configuration included increasing the heap size for task JVMs to 1 GB, compressing map output using LZO [25], and allowing task JVMs to be reused. We also increased the sort buffer size so that the entire output from each map task fit in the buffer without spilling to disk. Finally, the replication factor for output files was set to 1.

Our performance results are summarized as follows:

- We first measured time and space overhead of provenance capture. For our two experiments, provenance capture incurred 20-76% time overhead. For Terasort, the space overhead incurred by provenance capture was 21%; for Wordcount, the space overhead can be made arbitrarily large. Details are reported in Section 6.1.
- We then measured the time to backward-trace output elements after provenance has been captured. Backward-tracing one element took as little as 1.5 seconds in Terasort, but again can be made arbitrarily large in Wordcount. Details are reported in Section 6.2.

6.1 Performance: Capture

We report the time and space overhead associated with capturing provenance in our experiments. For each input data size, we ran Wordcount five times and Terasort three times, with and without capturing provenance; we report the average of the trials, which had little variance.

Figures 6(a) and 7(a) report the time overhead by showing the time taken without provenance capture (dark bar) and the additional time with capture (light bar).

- *Execution time:* Time for the entire MapReduce job to complete.
- *Map finish time:* Time until all map tasks are complete.
- *Average reduce task time:* Average time for an individual reduce task.

Figures 6(b) and 7(b) similarly report the space overhead of provenance.

- *Map output data size:* The size of the map output data (before applying the combiner).
- *Intermediate data size:* The size of the intermediate data after applying the combiner and LZO compression.
- *Output data size:* The size of the final output data.

Observe in Figures 6 and 7 that time and space overhead are closely related. A larger output data size increases the average reduce task time, because the output data is written by the reduce tasks. Similarly, a larger intermediate data size increases the map finish time as well as the average reduce task time, because map tasks store intermediate data temporarily in local disk, and reduce tasks sort the intermediate data set. In our experiments, the map output data size had little impact on the execution time, because the entire map output data set fit in the sort buffer.

For Wordcount, provenance capture incurred 72-76% time overhead (Figure 6(a)). Because Wordcount is a many-one transformation, each intermediate and output data element is annotated with many input element IDs. Moreover, the number of annotations per data element increases with input size, because we have the same fixed number of words across all data sizes. As a result, the space overhead percentage grows linearly with input size, and can be made arbitrarily large. The significant increase in both intermediate and output data sizes (shown in Figure 6(b), where the dark bars are not even visible) correlates with the large time overhead of provenance capture.

For Terasort, provenance capture incurred 16-20% time overhead and 19-21% space overhead (Figures 7(a) and 7(b), respectively). Because Terasort is a one-one transformation, each intermediate and output data element is annotated with exactly one input element ID. The moderate increase in both intermediate and output data sizes (shown in Figure 7(b)) is consistent with the small time overhead.

6.2 Performance: Tracing

For both experiments, we measured the time to backward-trace output elements after provenance has been captured, for varying input data sizes. In our experiments we did not fetch the actual input data elements as part of tracing; we just identified their element IDs. Overall, we have not yet focused our work on making backward tracing as efficient as possible; we report the tracing performance based on our current implementation.

For Wordcount, tracing one element took approximately 1, 3, and 5 minutes, for 100, 300, and 500 GB input data sizes respectively. Note that even when we trace a single output element, the data sizes processed become quite large: For 100 GB input data, the average number of occurrences for each word is about 1,289,000. As discussed in the previous section, because we have a fixed number of words across all input data sizes, the provenance of individual output elements grows linearly with input size. This behavior explains the linear growth of tracing time.

For Terasort, tracing one element took approximately 1.5 seconds for all input data sizes. Since the data sizes processed are very small—each element traced produces one element as a result—the binary search (Section 5.3) tends to dominate tracing time. We suspect that deploying an appropriate index could improve backward tracing time by at least factor of two. The use of indexes in general is an immediate area of future work.

7. CONCLUSIONS AND FUTURE WORK

This paper defines provenance for map and reduce functions, and it identifies properties that hold when one-level provenance is applied recursively in arbitrary GMRWs. We have built a prototype system as an extension to Hadoop that supports provenance capture and tracing; performance numbers are reported.

As described in Section 5, implementing efficient backward and forward tracing is an important next step in the RAMP system. For both types of tracing, building appropriate indexes will be a key component of our approach. In addition, we plan to measure the performance of provenance capture and tracing for MapReduce workflows consisting of multiple jobs. We have thus far successfully captured provenance for MapReduce workflows compiled by Pig [20] using RAMP. For future experiments, the PigMix [6] benchmarks seem like a good starting point.

One obvious general avenue for future work is to incorporate SQL processing into our workflows. 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 our framework. Other SQL queries may not fit the map or reduce paradigm precisely, but do have known, well-understood provenance [11, 13] that can be incorporated via extensions to our framework. Finally, several recent systems (e.g., Hive [24]) compile SQL queries into MapReduce jobs. We intend to compare provenance captured and traced using previous methods for SQL against using our system on the compiled GMRWs.

8. REFERENCES

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

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

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

[4] Apache. Hadoop cluster setup. *http://hadoop.apache.org/common/docs/r0.21.0/cluster_setup.html*.

[5] Apache. Mapreduce tutorial. *http://hadoop.apache.org/mapreduce/docs/r0.21.0/mapred_tutorial.html*.

[6] Apache. Pigmix benchmarks. *http://wiki.apache.org/pig/PigMix*.

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

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

[9] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI*, 2010.

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

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

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

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

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

[15] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[16] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007.
[17] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *SIGMOD*, 2008.
[18] R. Ikeda, S. Salihoglu, and J. Widom. Provenance-based refresh in data-oriented workflows. Technical report, Stanford University InfoLab, March 2010.
[19] P. Missier, K. Belhajjame, J. Zhao, M. Roos, and C. Goble. Data lineage model for Taverna workflows with lightweight annotation requirements. In *IPAW*, 2008.
[20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.
[21] O. O'Malley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. *http://l.yimg.com/a/i/ydn/blogs/hadoop/yahoo2009.pdf*, 2009.
[22] S. Sarawagi. Information extraction. *Found. Trends databases*, 1:261–377, March 2008.
[23] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3), 2005.
[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. In *VLDB*, 2009.
[25] K. Weil. hadoop-lzo. *http://www.github.com/kevinweil/hadoop-lzo/*.
[26] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, 1997.
[27] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified relational data processing on large clusters. In *SIGMOD*, 2007.

APPENDIX

A. PROOF OF THEOREM 3.1

Theorem: 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 $o \in W(I_1^*, \ldots, I_m^*)$. □

A.1 Proof of Part 1

We 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 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 $I_1^U, \ldots, I_p^U$. Let $P_U(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. For any set $\mathbb{I}$ that combines subsets of U's input sets $I_1^U, \ldots, I_p^U$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_p)$, where each $I'_k = \mathbb{I} \cap I_k^U$. 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, $W(P_W(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$. □

A.2 Proof of Part 2

Lemma 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: 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 $I_1^U, \ldots, I_p^U$. Let $P_U(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. For any set $\mathbb{I}$ that combines subsets of U's input sets $I_1^U, \ldots, I_p^U$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_p)$, where each $I'_k = \mathbb{I} \cap I_k^U$. 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 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: 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 $I_1^U, \ldots, I_p^U$. 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 $I_1^U, \ldots, I_p^U$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_p)$, where each $I'_k = \mathbb{I} \cap I_k^U$. 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, $W(P_W(\{o_1, ..., o_n\})) \subseteq O$. □

Theorem: We prove a stronger property: Let $o_1, \ldots, o_n$ be elements of O. Then $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$. The proof is by induction on the structure of W. The base case and inductive step proofs are exactly the same as those for Lemma 1, with the exception of the Reduce inductive step.

Inductive step for Reduce: Suppose $W = W' \circ R$. If R is monotonic, then the Reduce inductive step proof from Lemma 1 suffices. Suppose R is nonmonotonic. Then W' has no nonmonotonic reduce functions. Let $P_R(\{o_1, ..., o_n\}) = (G_1 \cup ... \cup G_n)$, where each group G_j produces o_j. Let O^* denote $W'(P_{W'}(G_1 \cup \ldots \cup G_n)$. Lemma 1 on W' tells us that $O^* \supseteq G_1 \cup \ldots \cup G_n$. Let O' denote the output of W'. Lemma 2 on W' tells us that $O^* \subseteq O'$. 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. $R(O^*) \supseteq \bigcup_{j=1}^{n} R(G_j)$, which implies $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$. □

B. PROOF OF COROLLARY 3.1

Corollary: 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. Then $o \in W^*(P_W(o))$.

Proof: 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 1 and 2 of Appendix A, 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, $O \supseteq W^*(P_W(\{o_1, ..., o_n\}))$. $\square$