RAMP: A System for Capturing and Tracing Provenance in MapReduce Workflows*

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

ABSTRACT

RAMP (*Reduce And Map Provenance*) is an extension to Hadoop that supports provenance capture and tracing for workflows of MapReduce jobs. RAMP uses a wrapper-based approach, requiring little if any user intervention in most cases, while retaining Hadoop's parallel execution and fault tolerance. We demonstrate RAMP on a real-world MapReduce workflow generated from a Pig script that performs sentiment analysis over Twitter data. We show how RAMP's automatic provenance capture and tracing capabilities provide a convenient and efficient means of drilling-down and verifying output elements.

1. INTRODUCTION

MapReduce [3] has become a very popular framework for large-scale data processing. Some data-processing tasks are too complex for a single MapReduce job, so individual jobs may be composed in acyclic graphs to form *MapReduce workflows*. In addition to workflows constructed by hand, MapReduce workflows are the target of higher-level platforms built on top of Hadoop [2], such as Pig [5], Hive [7], and Jaql [1].

Debugging MapReduce workflows can be a difficult task: their execution is batch-oriented and, once completed, leaves only the data sets themselves to help in the debugging process. *Data provenance*, which captures how data elements are processed through the workflow, can aid in debugging by enabling *backward tracing*: finding the input subsets that contributed to a given output element. For example, erroneous input elements or processing functions may be discovered by backward-tracing suspicious output elements. Provenance and backward tracing also can be useful for drilling-down to learn more about interesting or unusual output elements.

We propose to demonstrate *RAMP* (*Reduce And Map Provenance*), an extension to Hadoop that captures and traces provenance in any MapReduce workflow. RAMP uses a wrapper-based approach to capture fine-grained provenance transparently, while retaining Hadoop's parallel execution and fault tolerance. In previous work [4], we showed that RAMP imposes reasonable time and space overhead during provenance capture. Moreover, RAMP's default scheme for storing provenance enables efficient backward tracing without requiring special indexing of provenance information.

In the remainder of this demonstration proposal, we:

- Provide foundations of provenance for MapReduce workflows, summarizing material from [4] (Section 2)
- Explain how RAMP captures and traces provenance (Section 3)
- Describe the MapReduce workflow and data sets to be used in the demonstration, and walk through real-world debugging and drill-down scenarios (Section 4)

2. FOUNDATIONS

The MapReduce framework involves *map functions* and *reduce functions*:[1]

Map Functions. 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. The RAMP system currently assumes pure map functions.

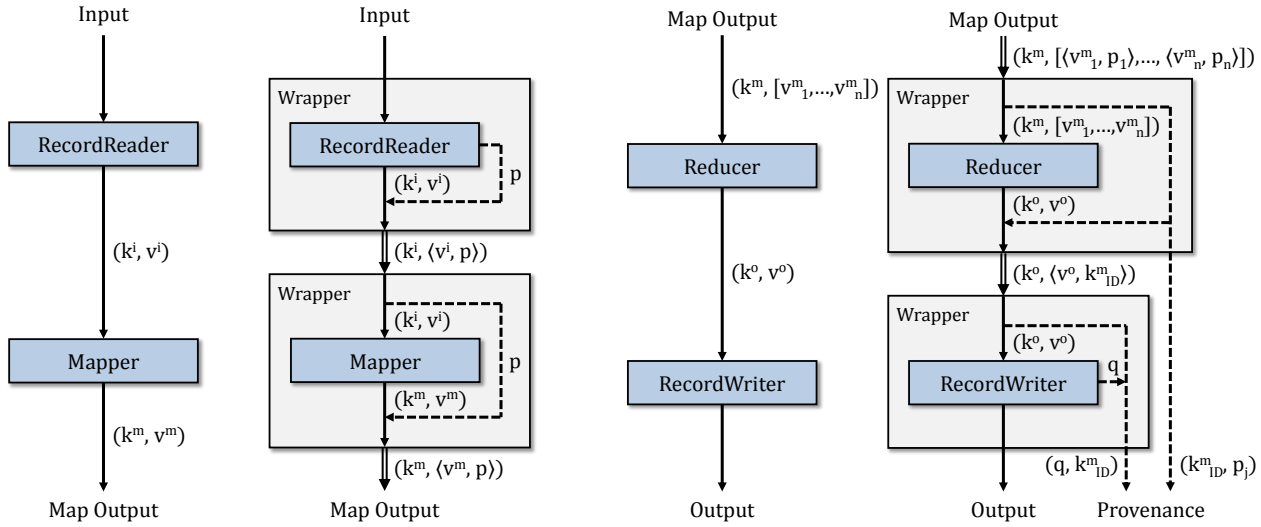
Reduce Functions. A *reduce function* R takes an input data set I in which each element is a key-value pair, and returns zero or more output elements independently for each group of elements in I with the same key: Let $k_1, \ldots, k_n$ be all of the distinct keys in I. Then $R(I) = \bigcup_{1 \leq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

Let transformation T be either a map or a reduce function. 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 function type, then we show how this "one-level" provenance is used to define workflow provenance.

*This work was supported by the National Science Foundation (IIS-0904497), the Boeing Corporation, KAUST, and an Amazon Web Services Research Grant.

Proceedings of the VLDB Endowment, Vol. 4, No. 12

[1]This section assumes workflows are arbitrary compositions of separate map and reduce functions. Our implementation also handles traditional MapReduce jobs that combine a map and a reduce function into a single transformation.



(a) Wrapping a map function

(b) Wrapping a reduce function

Figure 1: Provenance capture in RAMP.

Provenance for single functions 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., $o \in R(G_j)$.

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 MapReduce workflow: an arbitrary acyclic graph composed of map and reduce functions. 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 2.1 (MAPREDUCE PROVENANCE). Consider a MapReduce workflow W with initial input I and any data element e. The provenance of e in W, denoted $P_W(e)$, is a set $I^* \subseteq I$. If e is an initial input element, i.e., $e \in I$, 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 $P_W(e) = \bigcup_{e' \in P_T(e)} P_W(e')$. □

This recursive definition is quite intuitive: If the "one-step" provenance of an output element o through the map or reduce function that produced o is the set E of intermediate elements, then o's provenance is (recursively) the union of the provenance of the elements in E. Additional formal material on provenance in MapReduce workflows appears in [4].

3. SYSTEM OVERVIEW

RAMP is built as an extension to Hadoop. It consists of three main components: a generic wrapper implementation for capturing provenance, pluggable schemes for assigning element IDs and storing provenance, and a stand-alone program for tracing provenance. Our current implementation is compatible with the Hadoop 0.20 API (also known as the "new" API).

RAMP captures provenance by wrapping the Hadoop components that define a MapReduce job: the *record-reader*, *mapper*, *combiner* (optional), *reducer*, and *record-writer*. This wrapper-based approach is transparent to Hadoop, retaining Hadoop's parallel execution and fault tolerance. Furthermore, in many cases 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.

Since RAMP stores provenance as mappings between input and output element IDs, RAMP requires schemes for assigning element IDs and storing provenance. When input and output data sets are stored in files, RAMP uses (*filename*, *offset*) as a default unique ID for each data element, so user intervention is not needed. RAMP also has a default provenance storage scheme for file input and output; details are in [4]. For other settings, RAMP allows users to define custom ID and storage schemes.

In previous work [4], we conducted performance experiments on two standard MapReduce jobs: *Wordcount* and *Terasort*. For these experiments, which were run on a 51-machine Hadoop cluster with 500GB of input data, provenance capture incurred 20-76% time overhead. Backward-tracing one element from the full data set took as little as 1.5 seconds without special indexes.

3.1 Provenance Capture

Although our formalism in Section 2 was based on individual map and reduce functions, our implementation is based on MapReduce jobs. We assume our workflows combine adjacent map and reduce functions into MapReduce jobs; all remaining independent map and reduce functions are treated as MapReduce jobs with identity reduce or map components, respectively. For presentation purposes, we consider MapReduce jobs without a combiner; the extension for combiners is straightforward.

For map functions, RAMP adds to each map output element (k^m, v^m) a unique ID p for the input element (k^i, v^i) that generated (k^m, v^m) (Figure 1(a)). 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.

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

```
1  raw_movie = LOAD 'movies.txt' USING PigStorage('\t') AS (title: chararray, year: int);
2  movie = FOREACH raw_movie GENERATE LOWER(title) as title;
3
4  raw_tweet = LOAD 'tweets.txt' USING PigStorage('\t') AS (datetime: chararray, url, tweet: chararray);
5  tweet = FOREACH raw_tweet GENERATE datetime, url, LOWER(tweet) as tweet;
6  rated = FOREACH tweet GENERATE datetime, url, tweet, InferRating(tweet) as rating;
7  ngramed = FOREACH rated GENERATE datetime, url, flatten(GenerateNGram(tweet)) as ngram, rating;
8  ngram_rating = DISTINCT ngramed;
9
10 title_rating = JOIN ngram_rating BY ngram, movie BY title USING 'replicated';
11 title_rating_month = FOREACH title_rating GENERATE title, rating, SUBSTRING(datetime, 5, 7) as month;
12
13 grouped = GROUP title_rating_month BY (title, rating, month);
14 title_rating_month_count = FOREACH grouped GENERATE flatten($0), COUNT($1);
15
16 november_count = FILTER title_rating_month_count BY month eq '11';
17 december_count = FILTER title_rating_month_count BY month eq '12';
18 outer_joined = JOIN november_count BY (title, rating) FULL OUTER, december_count BY (title, rating);
19 result = FOREACH outer_joined GENERATE (($0 is null) ? $4 : $0) as title, (($1 is null) ? $5 : $1) as
       rating, (($3 is null) ? 0 : $3) as november, (($7 is null) ? 0 : $7) as december;
20 STORE result INTO '/sentiment-analysis-result' USING PigStorage();
```

Figure 2: Pig script for Twitter data analysis.

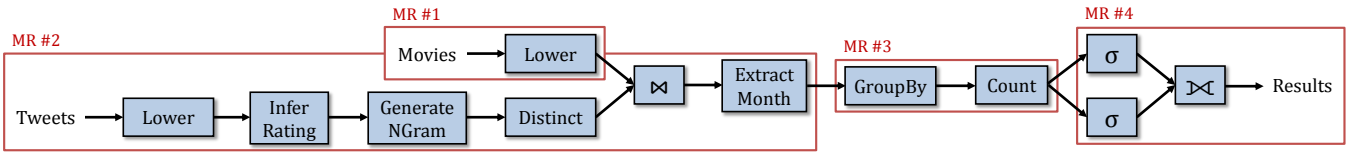


Figure 3: MapReduce workflow compiled from Pig script.

the map provenance as a mapping from the grouping key k^m to the input element ID p_j's (Figure 1(b)). 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}.

3.2 Provenance Tracing

Since RAMP captures provenance for each MapReduce job's output elements with respect to the job's inputs, 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 above to determine the corresponding grouping key ID k^m_{ID}.
2. Using k^m_{ID}, RAMP accesses the map provenance as specified above 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.

Carefully crafted schemes for assigning element IDs and storing provenance can improve the efficiency of provenance tracing. For example, RAMP's default schemes for file input and output always store provenance in ascending key order so that RAMP can exploit binary search on the provenance data during backward tracing.

4. DEMONSTRATION

Since it's difficult to give an engaging demonstration of the automatic wrapping process, our demonstration will primarily show how RAMP's provenance capture and tracing capabilities are useful for drilling-down and verifying output elements in a realistic MapReduce workflow setting.

4.1 Workflow Description

We demonstrate RAMP on a MapReduce workflow for movie sentiment analysis using Twitter data. The workflow is compiled from a Pig script (Figure 2) that takes two data sets as input:

- Tweets collected over several months in 2009 [8]
- 478 highest-grossing movies from the Internet Movie Database (`http://www.imdb.com/boxoffice/alltimegross`)

The Pig script infers movie ratings from the Tweets as follows:

1. For each Tweet, it invokes the UDF `InferRating()`, which uses sentiment analysis to infer a 1–5 overall sentiment rating. It uses a Naive Bayes classifier trained with a sentence polarity dataset [6], using unigrams as features.
2. For each rated Tweet from Step 1, it invokes the UDF `GenerateNGram()`, which generates all possible n-grams from each Tweet. (Currently we limit n to 3, thereby missing a few movies with longer names.) It then joins the generated n-grams with the movies from IMDb to find all movie titles (if any) mentioned in the Tweet.

Lastly, the script counts the number of instances of each rating for each movie, separating November and December (2009). A portion of the final output can be seen in the background of Figure 4 (columns are *movie*, *rating*, *#november*, and *#december*).

4.2 Running the Workflow

We begin the demonstration by compiling the Pig script into a MapReduce workflow; the result consists of four MapReduce jobs, as shown in Figure 3. RAMP automatically wraps the generated MapReduce jobs, so when the workflow is executed in Hadoop, we see that provenance files have been created in addition to the output files.

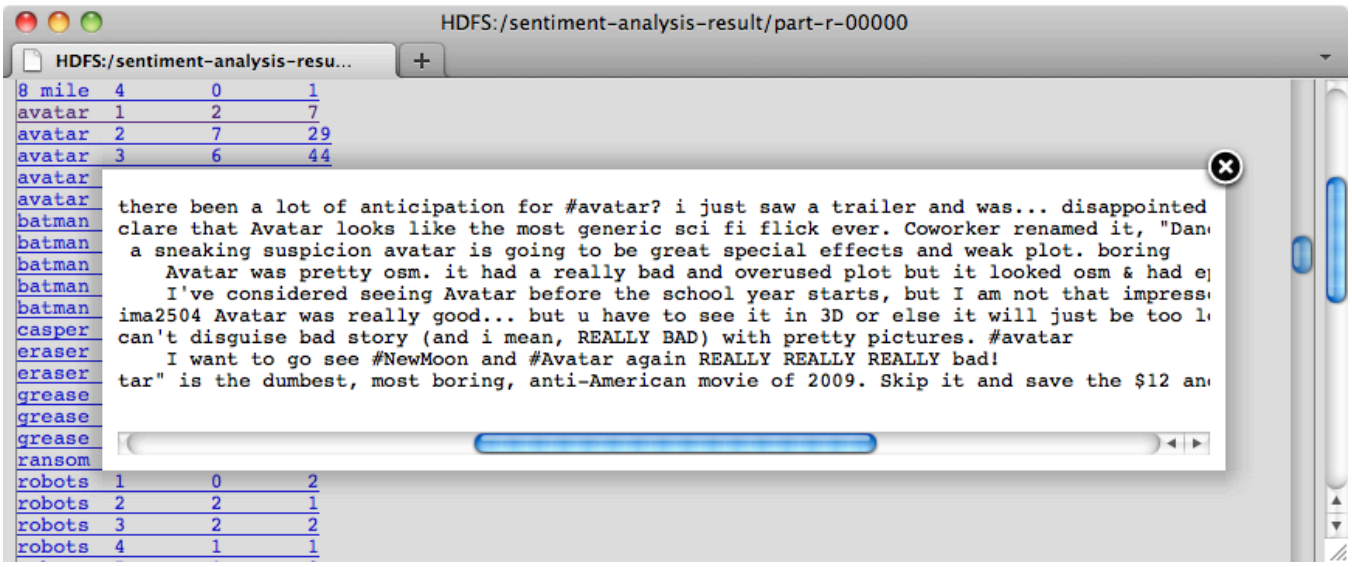


Figure 4: Provenance of the output element `(title:avatar, rating:1, nov:2, dec:7)`.

4.3 Provenance Tracing

After executing the workflow, we use RAMP's backward-tracing feature to investigate suspect or otherwise unusual output elements. Note that without the provenance captured by RAMP, it would be difficult and time-consuming—perhaps even impossible—to find the input Tweets from which a particular movie rating was inferred.

RAMP's interface, shown in Figure 4, is a modified version of the *HDFS* (*Hadoop Distributed File System*) web interface. RAMP's interface allows the user to browse input and output elements as well as backward-trace output elements. Initially, RAMP shows the set of output elements with hyperlinks, visible in the background of Figure 4. Clicking any output element causes a window to appear showing its provenance, visible in the foreground of Figure 4.

Drill-Down Scenario: Avatar

Suppose we would like to know why some people did not like the movie *Avatar*. We investigate the output element `(title:avatar, rating:1, nov:2, dec:7)`. When we click on it to view its provenance, we find the nine Tweets that resulted in a 1 rating for Avatar, shown in Figure 4. (Scrolling to the left more clearly delineates the nine separate Tweets.) Of the Tweets that criticize the film, one mentions a "bad and overused plot" while another complains about the "bad story." Thus, it seems that Avatar's plot is a point of criticism. But we also discover through backward-tracing that not all nine Tweets in the provenance actually contain negative opinions; for example, one Tweet expresses the desire to see Avatar "really bad."

Drill-Down Scenario: New Moon

Conversely, suppose we would like to know why some people *did* like the movie *New Moon*. We backward-trace the output element `(title:new moon, rating:5, nov:54, dec:13)` to obtain the relevant Tweets, but we don't immediately see a pattern or problem. We then notice that the Twitter data contains an attribute called `url`, pointing to the user account that created each Tweet. Inspecting the user accounts, we see that the Tweets comprising the provenance of the 5 ratings for New Moon were written almost exclusively by teenage girls.

Debugging Scenario: Eclipse

Finally, we backward-trace output element `(title:eclipse, rating:5, nov:9, dec:15)` to understand why people liked the movie *Eclipse*. We find that most of the Tweets in the element's provenance have no mention of the movie Eclipse; some Tweets discuss a lunar eclipse, while others rave about the Eclipse IDE. Of the Tweets that are about the movie, all simply express a desire to see the film. It turns out that Eclipse was not released until June 2010, while all of the Tweets in our data set are from 2009. To avoid speculative ratings in future runs of the workflow, we could modify the Pig script to filter out movies not yet released at the time of the Tweet.

5. REFERENCES

[1] Jaql. *http://code.google.com/p/jaql/*.

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

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[4] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *CIDR*, pages 273–283, 2011.

[5] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[6] B. Pang and L. Lee. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *ACL*, pages 115–124, 2005.

[7] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive: A petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[8] J. Yang and J. Leskovec. Patterns of temporal variation in online media. In *WSDM*, pages 177–186, 2011.