Provenance-Based Debugging and Drill-Down in Data-Oriented Workflows*

Robert Ikeda, Junsang Cho, Charlie Fang, Semih Salihoglu, Satoshi Torikai, Jennifer Widom

Stanford University

{rmikeda,ks038c,charlief,semih,storikai,widom}@cs.stanford.edu

I. Introduction

The amount of publicly available data has grown in recent years, in part due to initiatives from governments [2], [5] and scientific journals [7], [8]. Users who download and analyze this data often execute a number of processing steps, including data extraction, integration of multiple sources, aggregation, and ad-hoc transformations and queries. Such analyses can be managed using *data-oriented workflows* [13], in which input data sets are processed by a graph of *processing nodes* to produce output data.

Although existing workflow systems such as [14], [16] are useful for building and executing these types of analyses, additional functionality can further enhance their usefulness:

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

Provenance, which captures where data came from and how it is processed through the workflow, can be used to support both debugging and drill-down.

There has been plenty of past work on provenance in workflows, but most approaches either track coarse-grained (schema and/or processing-node level) provenance (e.g., [9], [11]), which is insufficient to support record-level debugging and drill-down, or track the provenance of individual data records with *physical provenance*—requiring each output record to be annotated with some type of identifier for the contributing input records before debugging or drill-down (e.g., [10], [15]).

We demonstrate *Panda* (for *Provenance and Data*), a system for data-oriented workflows that supports debugging and drill-down using *logical provenance*—provenance information stored at the processing-node level. For many processing nodes, including a large subset of SQL, logical provenance can be derived from the node's specification, and it captures exactly the same information as physical provenance but in a much more compact fashion (one specification per processing node rather than one per data record). Furthermore, for these nodes, logical provenance incurs no overhead at workflow-execution time. For some processing nodes, Panda must "augment" the output to support logical provenance. In the worst case, the augmenting process unavoidably degenerates to the equivalent of storing physical provenance.

In our demonstration we use Panda to integrate, process, and analyze actual education data from multiple sources. Panda is a complete system providing the following functionality:

- **Data-oriented workflows**: Panda supports the creation and execution of arbitrary acyclic workflows containing both relational (SQL) processing nodes and opaque processing nodes programmed in Python.
- **Generating provenance:** Logical provenance specifications in Panda consist of *attribute mappings* and *filters*. For SQL processing nodes, Panda generates logical provenance automatically. For Python nodes, the user has the option to specify attribute mappings. Otherwise, Panda automatically augments the node's output to support logical provenance.
- **Provenance operations:** Panda supports *backward tracing* (finding the input records that contributed to a given output record) and *forward tracing* (determining which output records were derived from a given input record). Panda also supports *incremental maintenance* and *selective refresh* [12], however the focus of this demo is on using Panda's tracing capabilities for debugging and drill-down.
- **Comprehensive GUI:** Panda's graphical user interface (Figure 1) allows users to view a workflow graph; browse and query input, intermediate, and output data sets; and perform tracing operations. Since our focus is not about workflow development tools (many such tools already exist), workflow graphs are created and executed in Panda by issuing a sequence of commands.

II. System Description

Panda supports user interaction through its command-line interface and its GUI. There are three basic types of user commands: (1) creating or modifying input data sets; (2) creating processing nodes that generate newly-defined data sets from existing ones, to build up workflows; (3) provenance tracing. Currently, all data sets handled by Panda are encoded in relational tables, and all records are given a globally unique

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

ID. The main backend is **SQLite**, which stores all data sets, SQL processing nodes, provenance, and workflow information. Python processing nodes are stored separately in files.

A. Preliminaries

For each processing node in a workflow, Panda generates logical provenance specifications consisting of *attribute mappings* and *filters*.

Definition 2.1 (Attribute Mapping): Let P be a processing node with input sets $I_1, \ldots, I_n$ and output set O. Let A be an attribute in the schema of I_k, and let B be an attribute in the schema of O. P has an *attribute mapping* from input attribute $I_k.A$ to output attribute $O.B$, denoted $I_k.A \leftrightarrow O.B$, if for all $x \in domain(B)$ and all possible input set instances $I'_1, .., I'_n$:
$\sigma_{B=x}(P(I'_1, .., I'_n)) = \sigma_{B=x}(P(I'_1, .., \sigma_{A=x}(I'_k), ..I'_n))$. □

In words, attribute mapping $I_k.A \leftrightarrow O.B$ states that given an output tuple t with $t.B = x$, t is unaffected by all records in I_k except those where $A = x$.

Definition 2.2 (Filter): Let P be a processing node with input sets $I_1, \ldots, I_n$ and output set O. P has a *filter condition* C on I_k if for all possible input set instances $I'_1, .., I'_n$:
$P(I'_1, .., I'_n) = P(I'_1, .., \sigma_C(I'_k), ..I'_n)$. □

In words, filter C states that output tuples are never affected by records in I_k except those satisfying C.

From these definitions we can see informally how attribute mappings and filters are used to define (and find) the record-level provenance of individual data records.

B. SQL Processing Nodes

Panda supports processing nodes specified as SQL queries, currently limited to SELECT blocks with optional grouping and aggregation. As an example, consider a data set Scores(schoolID, county, mean, #studs, year) containing test score averages for schools in California. Suppose the user wants to create a processing node that aggregates test scores in 2011 by county. The following Panda command does the job:

```
Create Table CountyAgg As
Select county,
 (sum(#studs*mean)/sum(#studs)) as mean,
From Scores
Where year = 2011
Group By county
```

In response to this command, Panda:

1. Runs the SQL query to create the new data set CountyAgg
2. Inserts a record into the Workflow metadata table connecting Scores to CountyAgg via the above SQL query
3. Stores logical provenance for CountyAgg

Panda generates logical provenance through syntactic analysis of SQL queries. For this query, Panda generates the attribute mapping Scores.county ↔ CountyAgg.county and the filter condition `year = 2011` on Scores.

Now suppose the above query did not contain the county attribute in its SELECT clause. No attribute mappings would hold, so logical provenance would contain only the filter condition, which is not very precise. In this case Panda augments the query automatically in support of logical provenance: Panda adds the county attribute to the SELECT clause for the purpose of provenance tracing, then creates a SQL view for the output data set to hide the extra attribute.

C. Python Processing Nodes

Currently, Panda only supports Python processing nodes that are one-one or one-many transformations—i.e., the processing node operates on one record at a time—although this restriction can be lifted with some additional work. Consider an input data set Raw(school, rawdata), containing school test score data encoded in a string format. Suppose the user wants to extract the information into a more structured record format, stored in a table Clean(school, ...). The user writes a Python script **extract.py** that takes as its argument one record from the input data set, and returns zero or more output records suitable for insertion into the output data set. The user appends the Python processing node to the workflow with the following command, which includes an optional attribute mapping specification:

```
Create Table Clean(<schema>)
As Python 'extract.py' on Raw
Attrmap school to school
```

In response to this command, Panda:

1. Executes Python script **extract.py** on each record in Raw, inserting the resulting record(s) into new data set Clean
2. Inserts a record into the Workflow metadata table connecting Raw to Clean via the Python script **extract.py**
3. Stores the specified attribute mapping as the logical provenance for Clean.

If no user-specified attribute mappings are provided, Panda augments each output record with the ID of the corresponding input record (calling it src_id in the output) and generates the attribute mapping I.ID ↔ O.src_id. Note in this case augmenting the node's output effectively degenerates to physical provenance.

D. Provenance Tracing

Provenance tracing is performed through Panda's GUI; see Figure 1. (The application captured in this screenshot is our demo scenario; see Section III.) The left panel contains the workflow graph, while selected data is currently shown in the right panel. Panda enables provenance to be traced between any data sets in the workflow graph, either forwards or backwards. More generally, our interface allows users to specify a *tracing path*: a *source data set*, and one or more *destination data sets* reachable via a single path either forwards or backwards (but not both) from the source. In Figure 1, the source data set is CountyData at the bottom (light shading), while the destinations are the other three (more

darkly) shaded data sets. Once the tracing path is specified, the user can trace provenance from any record in the source data set to the set of relevant records in the nearest destination data set. Any of these records can then be traced further to the next destination along the path, and so on (right panel in the screenshot; the buttons labeled with arrows next to the records are for provenance tracing).

Given a record in source data set S, to perform a single (backward or forward) tracing step, Panda computes the join of all tables contained in the path from S to the destination data set D, where the join and other conditions are based on the logical provenance (attribute mappings and filters) for the relevant processing nodes, and the unique ID of the tuple being traced. As an example, the following query traces the CountyScores record with ID=15 to destination RawScores:

```
Select Distinct R.*
From CountyScores C, SchoolScores S,
  RawScores R
Where R.ID = S.src_id
  And S.county = C.county And C.ID = 15
```

Although the tracing query by default joins all tables along the path, sometimes we can "chain" together attribute mappings to eliminate intermediate tables.

III. Demonstration

To illustrate the debugging and drill-down capabilities enabled by Panda's provenance features, our demo runs through a scenario involving data from the following sources:

1. *California Department of Education*: Average test scores by school on the California STAR exams [4]
2. *Wikipedia*: California per-capita income by county [3]
3. *RAND California*: Per-pupil spending by county [1]

Using Panda, we build a workflow (Figure 1), containing both SQL and Python processing nodes, to analyze how test scores are correlated to per-capita income and per-pupil spending. Initial processing of the three data sources produces:

1. CountyScores: Average test scores of California schools (by subject and grade level) are available in two data files containing score and school information respectively. The workflow computes average 2nd-grade English scores by county: it extracts the data from the two files, joins the two extracted data sets, filters the data for 2nd-grade English, and finally aggregates the test scores by county.
2. IncomeData: A Python processing node extracts per-capita income by county from the Wikipedia page.
3. PupilSpend: From a data file downloadable from RAND's website, a Python processing node extracts per-pupil spending by county.

The workflow uses a SQL join query to combine the three data sets, followed by a last "cleaning" step to produce output table CountyData(county_name, mean_score, income, pupil).

We demonstrate creating and running the workflow over the actual data, with Panda generating provenance automatically for every node. We then show with specific real examples how Panda's provenance tracing operations can be used for both debugging and drilling down.

A. Debugging Scenario

After running the workflow, we use a special plug-in that allows us to visualize the output data in two dimensions: we scatter-plot test scores against per-capita income. Predictably, the plot shows a positive correlation between test scores and income. However, there are a few counties with unexpectedly low test scores, suggesting that some part of the workflow might contain an error. We use provenance to investigate.

Trinity County is an example with surprisingly low scores, so we trace the provenance of the Trinity CountyData record, as shown in Figure 1. By backward-tracing this record's provenance to intermediate table SchoolScores, we see that some Trinity County schools have an average test score of 0, strongly suggesting a workflow or data error. Tracing the provenance for these schools, we see in the raw input data (RawScores) that their average test scores are missing, replaced by *. It turns out that due to privacy concerns, schools with fewer than 10 test-taking students do not release average test scores. In the workflow, the Python node that extracted the test scores from the raw data mistakenly interpreted the *'s as 0's.

Having discovered this error, we correct the workflow by modifying the Python code to drop scores reported as *. We rerun the workflow and see that Trinity, as well as the other suspect counties, now fall into the expected curve.

B. Drill-Down Scenario

To illustrate drill-down, we use our visualizer to scatter-plot test scores against per-pupil spending. Although we previously saw a strong correlation between test scores and income, here we do not see as much of a correlation: most of the points appear to be randomly distributed within a fairly compact region. However, Alpine County is a clear outlier, with very high per-pupil spending (and not especially high test scores).

By backward-tracing the Alpine CountyData record to the data set containing test scores by school (RawScores), we see that only 15 students altogether in Alpine County took the test. A quick browse through other records in the same table reveals that most counties have hundreds or thousands of test-takers. This small number of students suggests that Alpine's per-pupil spending might be high partly because there are few students to support. Upon further research, a report from the Legislative Analyst's Office confirms this hypothesis: Alpine's high per-pupil spending is due to the county's high property values relative to the number of students [6]. Thus, we see that in addition to tracking down erroneous values or processing, provenance tracing can also be used to explore the source of surprising but correct results.

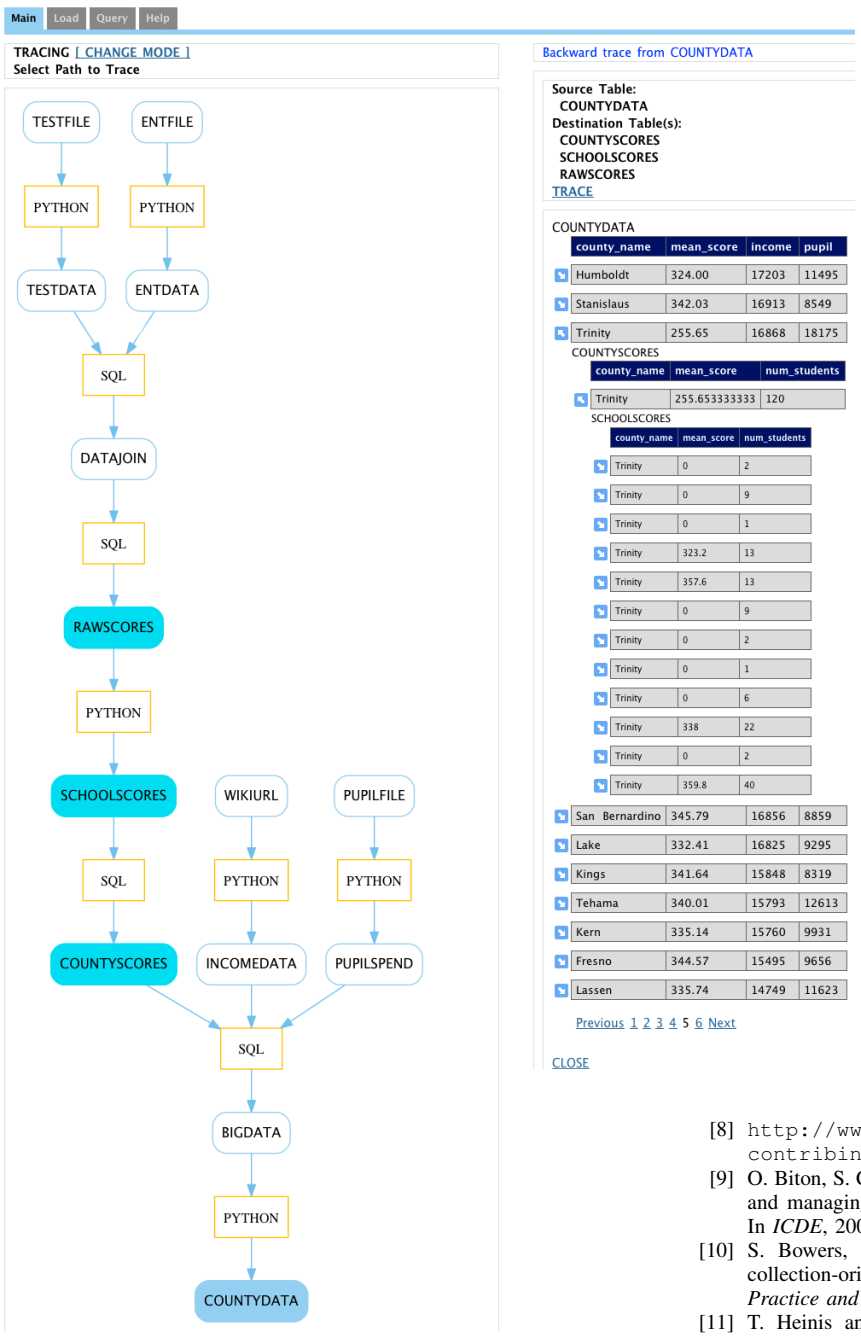

Fig. 1. Screenshot of Panda.

REFERENCES

[1] http://ca.rand.org/stats/education/perpupil_co.html.

[2] http://data.gov.uk.

[3] http://en.wikipedia.org/wiki/California_locations_by_per_capita_income.

[4] http://star.cde.ca.gov/star2009/ResearchFileList.asp.

[5] http://www.data.gov.

[6] http://www.lao.ca.gov/1996/082196_prop_taxes/property_tax_differences_pb82196.html.

[7] http://www.nature.com/authors/policies/availability.html.

[8] http://www.sciencemag.org/site/feature/contribinfo/prep/gen_info.xhtml#dataavail.

[9] 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.

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

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

[12] R. Ikeda, S. Salihoglu, and J. Widom. Provenance-based refresh in data-oriented workflows. Technical report, Stanford University InfoLab, March 2010.

[13] R. Ikeda and J. Widom. Panda: A system for provenance and data. *IEEE Data Engineering Bulletin*, September 2010.

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

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

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