Provenance for SQL through Abstract Interpretation: Value-less, but Worthwhile

Tobias Müller  
Universität Tübingen  
Tübingen, Germany

Torsten Grust  
[to.mueller, torsten.grust]@uni-tuebingen.de

ABSTRACT

We demonstrate the derivation of fine-grained where- and why-provenance for a rich dialect of SQL that includes recursion, (correlated) subqueries, windows, grouping/agggregation, and the RDBMS’s library of built-in functions. The approach relies on ideas that originate in the programming language community—program slicing and abstract interpretation, in particular. A two-stage process first records a query’s control flow decisions and locations of data access before it derives provenance without consultation of the actual data values (rendering the method largely “value-less”). We will bring an interactive demonstrator that uses this provenance information to make input/output dependencies in real-world SQL queries tangible.

1. PROVENANCE AWAY FROM THE LAB

We demonstrate the derivation of fine-grained provenance for a full-featured dialect of SQL. Data provenance for SQL uncovers the—sometimes intricate—dependencies between the output and inputs of a given query [2]:

- Exactly which parts of the input were used to compute this particular piece of the output? [where-provenance]
- Which parts of the input were inspected to decide that this piece is present in the output? [why-provenance]

Provenance has long been identified as a valuable tool in the language community—program slicing and abstract interpretation coincide with established notions of provenance [1,2]. A considerable gap, however, yawns between the languages for which provenance has been studied and those queries actually found in the field. “In the lab,” the principal objects of study have been the (positive) relational algebra and its SQL equivalent, possibly augmented with aggregation, over sets of tuples. The language subsets grew over time [5] but significant restrictions regarding the data model, acceptable query constructs, or tractable query classes (e.g., invertible queries only [11]) remained.

Data provenance for SQL queries “away from the lab.” Here, we explore an approach that embraces SQL constructs like grouping, aggregation, window functions, (correlated) subqueries, recursive common table expressions, as well as built-in and user-defined functions. We stay true to SQL’s tables-of-rows data model. Where- and why-provenance is derived in the granularity of individual table cells.

The present method translates SQL queries into program code (a subset of Python [9] for this demonstration, but any imperative language would do). We then build on ideas developed by the programming language community—program slicing [10] and abstract interpretation [3], specifically. In a nutshell, we (1) instrument the code to write a log of control flow decisions as well as data access locations, then (2) interpret selected aspects of the code to derive where and why-dependencies. The latter, abstract interpretation of the query code is “value-less,” i.e., it is entirely based on the succinct logs and does not consult or compute actual data.

This alternative route to provenance derivation might lead to a general framework that can cope with rich query languages, types, and function libraries. In the present case of full-featured SQL, the results certainly are promising. We will bring a fully functional implementation of the technique, hosted on top of PostgreSQL v9 [8]. An interactive demonstrator allows to examine the input/output dependencies of canned and ad hoc SQL queries.

2. IN/OUTPUT DEPENDENCIES FOR SQL

We continue to briefly explore three SQL queries and the dependencies they establish between their input and output table cells. This also provides a flavor of what the demo audience will encounter on-site.

Let us focus on a deliberately simple example first. Two base (input) tables represent travel agencies and the external tours they advertise (Figure 1). Among these agencies, the SQL query of Figure 2(a) finds those that offer boat trips. The scenario has been directly copied from [2] to manifest where-/why-dependencies found by abstract interpretation coincide with established notions of provenance [1,2].

Output dependency and input influence. Mouse clicks in the input and output tables inspect the provenance of the selected cell (indicated by in Figure 1 and Figure 2(b)). Clicks 1 and 2 reveal that the values of the selected output cells depend on (here: were copied from) column name of input rows t3 and t4, respectively. Such where-dependencies are highlighted via . Color-coding identifies exactly which input row participated in the computation of the
Figure 1: Travel agency scenario. The $t_i$ denote row ids. Where- and why-dependencies are marked by $\square$ and $\nabla$, respectively ($\square$ indicates a combination of both).

Figure 2: Finding output dependencies in a SQL join query.

Provenance in recursive common table expressions. Deriving provenance through abstract interpretation has the potential to embrace expressive query languages. We turn to a recursive SQL query (based on SQL:1999’s WITH RECURSIVE) to make this point. The query is designed to parse chemical formulae—like $C_6H_{12}O_6^{m}$—held in input table compounds (see Figure 5(a)). Formula syntax is given in terms of the finite state machine (FSM) shown on the right and encoded in table fsm in relational form. The query of Figure 5(b) parses all formulae “in parallel.” For each compound, the recursively defined run table holds the current FSM state as well as the residual input formula to parse. Note how the query relies on built-in string functions (LEFT, RIGHT, STRPOS, LENGTH [8]) to process its inputs and drive the FSM. The final output table of Figure 5(d) contains a trace of the (partially) parsed citrate formula as well as the FSM states and transitions that were activated during the parse.

SQL:1999 iteratively evaluates the common table expression’s body, yielding table run in each step, until run is found to be empty. Abstract interpretation can record provenance for each step, providing insight into how the recursive computation progressed. Figure 5(c) shows two such run tables at different time instants.

(a) Excerpt of input table externaltours (two copies to aid presentation).
(b) SQL query text.
(c) Output table.

Figure 3: In/output dependencies for a grouping query: What are the pricey boat tour destinations? (Query adapted from [2].)
with recursive
run(compound, step, state, formula) AS (  
  SELECT compound, 0, 0, formula  
  FROM compounds  
  UNION ALL  
  SELECT this.compound,  
    CASE  
      WHEN this.step + 1 AS step,  
        edge.target AS target state,  
        trim(this.formula, ‘\-’) AS formula  
    END from AS this  
    WHERE  
      not run AS this,  
      formula AS edge  
    AND  
      this.state = edge.source  
    AND  
      strpos(edge.label,  
        left(this.formula, 1)) > 0  
  )  
  SELECT r.step, r.state, r.formula  
  FROM run AS r  
  WHERE r.compound = ‘citrate’

Figure 7: Abstract variant of ascending(). Variables hold dependency sets, not values. When ascending() is entered, its argument where-depends on its own, i.e., xs[i] = {xs[i]}.  

3. VALUE-LESS, BUT WORTHWHILE

Behind the scenes, a given SQL query is translated into equivalent procedural code. This demonstration employs a subset of Python—featuring atomic values, dictionaries, lists, key lookups and index accesses, variable assignment, if-else conditionals, and while loops—chosen to faithfully represent the semantics of SQL as well the database host’s library of built-ins. (Sub)queries are mapped into functions that receive input tables and correlated row variables as arguments.

We do not hinge on Python. In fact, the current wave of work on compiling queries into program code perfectly complements the approach: what is described here would fit with, for example, the Scala source produced by LegoBase [6] or the LLVM code emitted by HyPer [7].

To keep this exposition brief, we sketch the two-phase approach using the archetypical Python function of Figure 6(a): ascending(xs) finds the largest element peak in a monotonically ascending prefix of list xs and returns (peak,True) if peak is the last element of xs (or else returns (peak,False)).

Phase 1 instruments the code such that (1) its control flow decisions (“Has a while loop been entered/left?”), “Which branch of this if-else was taken?”), and (2) the location of data structure accesses (non-constant list indices or dictionary keys) are logged. The resulting logs logf and logx, are lean streams of Booleans and indices, respectively, and are only appended to (using put(), see Figures 6(a) and 6(b)).
Phase 2 replays the behavior of the function solely based on one sequential scan of both logs (via get(), see Figure 7)—no actual database values are consumed or manipulated. Since dependencies are the only aspect of interest during provenance analysis, it suffices to execute a simplified value-less abstraction of the program [3]. The abstracted function computes dependency sets \( \pi \) for all variables \( x \):

1. \( \pi \subseteq \tau \) if \( x \)'s value has been computed based on variable \( y \),
2. \( \text{why}(\tau) \subseteq \pi \) if \( y \) has been used to decide whether \( x \)'s value is computed in the if or else branch of a conditional.\(^2\)

The lightweight logging of Phase 1 and the value-less Phase 2 both aid the non-intrusive and scalable implementation of provenance analysis. For the example of Figure 6, the phases derive the output dependency

\[
\text{ascending}([3, 4, 7, 5, 1, 2]) \Rightarrow (7, \text{False}).
\]

We observe that \text{ascending}() inspects the list only up to element 5 where it finds the monotonically ascending prefix to end; the preceding peak 7 is used to construct the output pair.

4. DEMONSTRATION SETUP

The on-site demonstration features a comprehensive implementation of provenance derivation for SQL, resting on top of a PostgreSQL (version 9) backend. Given a SQL query, we reach into the database host’s log to extract a sanitized and typed parse tree before the translation into procedural form is initiated. The demonstrator permits to review the Python code that is generated under the hood. Abstract interpretation then derives dependency sets (in a tabular form is initiated. The demonstrator permits to review the Python code that is generated under the hood. Abstract interpretation then derives dependency sets (in a tabular form) as described in Section 3.

To make data provenance tangible, we will bring an interactive frontend that renders dependencies much like we did in Figures 3 and 5 of Section 2. Clicks on table cells highlight all where- and why-dependent pieces in the input or output (see the screenshot of Figure 8). For recursive common table expressions, the intermediate states of the recursion may be inspected as well (not shown in the screenshot, but recall tables \( \text{run} \) and \( \text{fsm} \) of Figure 5(c)). Tables whose contents is better understood in terms of scatter/line mosaic plots or histograms can be rendered in alternative forms. These visualizations, too, allow a point-and-click exploration of the derived dependencies.

The demonstration will be live: we will bring a canned set of interesting data provenance scenarios—ranging from obvious to “tricky”—but the audience is invited to also formulate and analyze SQL queries in an ad hoc fashion.

Acknowledgments. Janek Bettiger built the browser-based interactive visualization of input/output dependencies.

5. REFERENCES