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

Tobias Müller Torsten Grust

Universität Tübingen
Tübingen, Germany

[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/aggregation, 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 tracking data lineage [4] as well as the understanding and debugging of queries [1]. 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 that *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 ❶ and ❷ reveal that the values of the selected output cells depend on (here: were copied from) column `name` of input rows t_5 and t_6, respectively. Such *where*-dependencies are highlighted via . Color-coding identifies exactly which input row participated in the computation of the

Proceedings of the VLDB Endowment, Vol. 8, No. 12

agencies

	name	based_in	phone
t_1	BayTours	San Francisco	415-1200
t_2	HarborCruz	Santa Cruz	831-3000

externaltours

	name	destination	type	price
t_3	BayTours	San Francisco	cable car	$50
t_4	BayTours	Santa Cruz	bus	$100
t_5	BayTours	Santa Cruz	boat	$250
t_6	BayTours	Monterey	boat	$400
t_7	HarborCruz	Monterey	boat	$200
t_8	HarborCruz	Carmel	train	$90

Figure 1: Travel agency scenario. The t_i denote row ids. *Where*- and *why*-dependencies are marked by and , respectively (indicates a combination of both).

```
SELECT e.name, a.phone
FROM   agencies AS a,
       externaltours AS e
WHERE  a.name = e.name
AND    e.type = 'boat'
```

(a) Which travel agencies offer boat tours?

output

name	phone
HarborCruz	831-3000
BayTours	415-1200
BayTours	415-1200

(b) Inspecting the provenance of two output table cells.

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

individual output cells. Abstract interpretation thus pinpoints *individual witnesses* [2], a notion not supported by [4], for example. We also learn that it is *not* t_1.name of table agencies that contributed the 'BayTours' value (in the query's SELECT clause, note that the developer used the column reference e.name but not the equivalent a.name).

All input cells that were inspected to decide the presence of the selected outputs are highlighted via (*why*-dependency). We find the expected dependency on the type cells of rows $t_{5,6}$. Row t_1 served as the single join partner for t_5 and t_6 based on the matching 'BayTours' values in their name columns.

Symmetrically, click ❸ on input row t_1 discloses that its phone cell value has contributed to the construction of two joined rows in the output (see the 415-1200 *where*-dependencies identified in Figure 2(b)).[1] Such input influences may be explored just like output dependencies in a bidirectional fashion.

Group construction and aggregate contributions. The SQL GROUP BY clause and its associated aggregation functions collapse input rows and thus, inherently, lose information. Provenance can help to unravel how groups were formed and identify those values that contributed to an aggregate. The SQL query of Figure 3(b) uses grouping to find boat tour destinations of high average price.

Click ❹ on the aggregate price $300 identifies *why*-dependencies on the rows $t_{6,7}$ in table externaltours: (1) the type cells of both rows were read (WHERE clause), (2) their destination cells were found to agree on value 'Monterey' such that t_6 and t_7 form one group, and (3) inside that group all cells in column price were inspected (HAVING). The additional *where*-dependency on $t_{6,7}$ indicates that the price cells were also used to compute the average of $300 (a new value not present in the input database).

Click ❺ on the Monterey output uncovers the known *why*-dependencies in table externaltours: the 'Monterey' group is in focus again and the same WHERE/HAVING conditions had to be checked to produce the output cell. We additionally see that t_6's destination cell has been used to form the group's key—by definition, all destination values in the group are equal but it was the particular row t_6 whose key was picked.

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 $C_6H_5O_7^{3-}$—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.

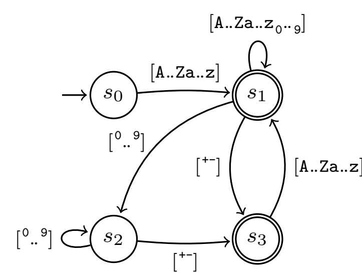

Figure 4: Formula syntax FSM.

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.

[1] *Why*-dependencies not shown to avoid clutter.

rs

	ne	destination	type	price
t_3		San Francisco	cable car	$50
t_4		Santa Cruz	bus	$100
t_5		Santa Cruz	boat	$250
t_6		Monterey	boat	$400
t_7		Monterey	boat	$200
t_8		Carmel	train	$90

rs

ne	destination	type	price
	San Francisco	cable car	$50
	Santa Cruz	bus	$100
	Santa Cruz	boat	$250
	Monterey	boat	$400
	Monterey	boat	$200
	Carmel	train	$90

(a) Excerpt of input table externaltours (two copies to aid presentation).

```
SELECT   e.destination,
         AVG(e.price)
FROM     externaltours AS e
WHERE    e.type = 'boat'
GROUP BY e.destination
HAVING   AVG(e.price) > $250
```

(b) SQL query text.

output

destination	AVG(·)
Monterey	$300

(c) Output table.

Figure 3: In/output dependencies for a grouping query: What are the pricey boat tour destinations? (Query adapted from [2].)

compounds

	compound	formula
t_9	citrate	$C_6H_5O_7{}^{3-}$
t_{10}	glucose	$C_6H_{12}O_6$
t_{11}	hydronium	H_3O^+

fsm

	source	labels	target	final
t_{12}	0	A..Za..z	1	false
t_{13}	1	A..Za..z0..9	1	true
t_{14}	1	0..9	2	true
t_{15}	1	+-	3	true
t_{16}	2	0..9	2	false
t_{17}	2	+-	3	false
t_{18}	3	A..Za..z	1	true

(a) Input and transition tables.

```
WITH RECURSIVE
run(compound, step, state, formula) AS (
  SELECT compound, 0, 0, formula
  FROM   compounds
 UNION ALL
  SELECT this.compound,
         this.step + 1 AS step,
         edge.target AS state,
         RIGHT(this.formula, -1) AS formula
  FROM   run AS this,
         fsm AS edge
  WHERE  LENGTH(this.formula) > 0
  AND    this.state = edge.source
  AND    STRPOS(edge.labels,
                LEFT(this.formula, 1)) > 0
)
SELECT r.step, r.state, r.formula
FROM   run AS r
WHERE  r.compound = 'citrate'
```

(b) SQL query (drives the FSM).

run

compound	step	state	formula
citrate	0	0	$C_6H_5O_7{}^{3-}$
glucose	0	0	$C_6H_{12}O_6$
hydronium	0	0	H_3O^+

⋮

run

compound	step	state	formula
citrate	3	1	${}_5O_7{}^{3-}$
glucose	3	1	${}_{12}O_6$
hydronium	3	1	${}^+$

(c) Intermediate parser states.

output

step	state	formula
0	0	$C_6H_5O_7{}^{3-}$
1	1	${}_6H_5O_7{}^{3-}$
2	1	$H_5O_7{}^{3-}$
3	1	${}_5O_7{}^{3-}$
4	1	$O_7{}^{3-}$
5	1	${}_7{}^{3-}$
6	1	${}^{3-}$
7	2	-
8	3	

(d) Parsing trace.

Figure 5: Deriving in/output dependencies for a SQL:1999 recursive common table expression (FSM parses chemical formulae).

Just like before, we select individual table cells to inspect the data dependencies created by the recursive query. Click ❻ inquires on the intermediate parsing state when all but the residual formula '$O_7{}^{3-}$' has already been read. We see that this suffix exclusively *where*-depends on the formula cells of compound 'citrate' (see the run tables as well as the original input row t_9 in table compounds). The *why*-dependencies reveal that only the FSM states s_0 and s_1 were activated to parse the formula prefix so far: the source, labels, and target cells of rows t_{12} and t_{13} of table fsm were inspected—up to this point in the recursion, no further row was touched.

Tracking input influence can help to assess query correctness. To illustrate, turn to column step which the developer introduced to watch recursion depth—otherwise this column is not meant to affect the computation. Indeed, with click ❼ on 0 in table run we find that this step cell exclusively influences other step cells: neither are further cells "polluted" with step values (no other *where*-dependencies), nor does the query inspect step to guide the actual parsing (no *why*-dependencies). We conjecture that such assertions may turn out valuable in the understanding and debugging of complex queries.

```
def ascending(xs):
  peak = xs[0]
  i = 1
  while i < len(xs):
                                put(log_cf, True)
                                put(log_ix, i)
    x = xs[i]
    if x >= peak:
                                put(log_cf, True)
      peak = x
    else:
                                put(log_cf, False)
      return (peak,False)
    i = i+1
                                put(log_cf, False)
  return (peak,True)
```

(a) Original code and its instrumentation.

written/read sequentially

log_{cf}	log_{ix}
⟨True,	⟨1,
True,	2,
True,	3⟩
True,	
True,	
False⟩	

(b) Log contents.

Figure 6: Python function `ascending()` and final log contents after execution of `ascending([3,4,7,5,1,2])`.

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 $\log_{cf}$ and $\log_{ix}$ are lean streams of Booleans and indices, respectively, and are only appended to (using *put*(), see Figures 6(a) and 6(b)).

```
def ascending(xs):
  peak = xs[0]
  while get(log_cf):
    x = xs[get(log_ix)]
    if get(log_cf):
      peak = x ∪ why(x) ∪ why(peak)
    else:
      peak = x ∪ why(x) ∪ why(peak)
      return (peak, why(x) ∪ why(peak))
  return (peak, ∅)
```

Figure 7: Abstract variant of `ascending()`. $\overline{\text{Variables}}$ hold dependency sets, not values. When `ascending()` is entered, its argument *where*-depends on itself, *i.e.*, $\overline{\texttt{xs[}i\texttt{]}} = \{\texttt{xs[}i\texttt{]}\}$.

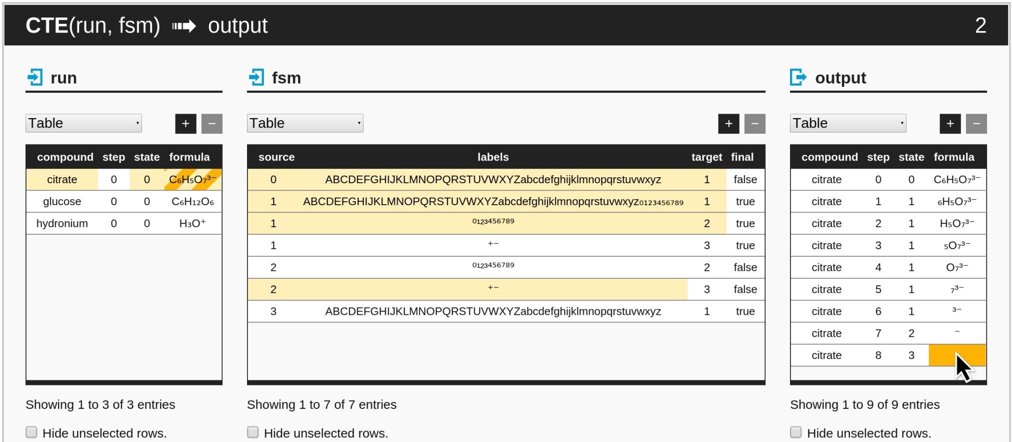

Figure 8: Provenance analysis for a full run of the SQL finite state machine of Figure 5(b). *Why*-dependencies highlight all state machine transitions that have been activated. The selected fully parsed string *where*-depends on the input $C_6H_5O_7^{3-}$ string only.

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* $\overline{x}$ for all variables x:

- $\overline{y} \subseteq \overline{x}$ if x's value has been computed based on variable y,
- $why(\overline{y}) \subseteq \overline{x}$ 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

`ascending([3,4,7,5, 1 , 2 ])` $\rightsquigarrow$ `(7,False).`

We observe that `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 representation, held by the database backend itself) 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 run and run 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 Bettinger built the browser-based interactive visualization of input/output dependencies.

5. REFERENCES

[1] P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A Characterization of Data Provenance. In *Proc. ICDT*, 2001.
[2] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4), 2007.
[3] P. Cousot and R. Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In *Proc. POPL*, 1992.
[4] Y. Cui, J. Widom, and J. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM TODS*, 25(2), 2000.
[5] B. Glavic and G. Alonso. Provenance for Nested Subqueries. In *Proc. EDBT*, 2009.
[6] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-Level language. In *Proc. VLDB*, 2014.
[7] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *Proc. VLDB*, 2011.
[8] *The PostgreSQL Relational Database System.* `postgresql.org`.
[9] *The Python Programming Language.* `python.org`.
[10] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4), 1984.
[11] A. Woodruff and M. Stonebraker. Supporting Fine-Grained Data Lineage in a Database Visualization Environment. In *Proc. ICDE*, 1997.

[2]We use $why(\overline{y})$ to convert set $\overline{y}$ of *where*-dependencies into a set of *why*-dependencies.