The Best Bang for Your Buck
When SQL Debugging and Data Provenance Go Hand in Hand
Benjamin Dietrich Tobias Müller Torsten Grust
Universität Tübingen Tübingen, Germany
[ b.dietrich, to.mueller, torsten.grust ]@uni-tuebingen.de

ABSTRACT
We report on our ongoing effort to develop observational debuggers for SQL. This debugging paradigm—in which the evaluation of selected subexpressions may be “spied on”—fits the nature of query languages, but may lead to observations whose size can overwhelm users. Here, we tackle this challenge with the help of data provenance analysis. The analysis identifies exactly those input rows that are material in producing suspect query outputs. Running the debugger on such a minimized input will exclusively yield observations that are indeed relevant in understanding the bug.

1. SPYING ON SQL EVALUATION
SQL queries are prone to bugs much like code written in conventional programming languages. The present work investigates debugging paradigms that fit the declarative nature of SQL and, in particular, shield users from low-level internals (like execution plans, for example). We argue that observational debugging [5], an idea rooted in the logic and functional programming communities, is one such paradigm: users mark the “suspect” subexpressions—ranging from simple arithmetics to entire subquery blocks—of a buggy SQL query to observe their value at runtime. Seeing the difference between the expected and observed evaluation of a subexpression has turned out to be an effective tool in uncovering subtle SQL bugs [3].

A sample debugging scenario is depicted in Figure 1. Tables cities and roads jointly model a road network in which only selected cities host fueling stations (label § in Figure 1(a), 0/1 in column fuel of table cities). Which cities can we reach from Alton if our car has a maximum range of 100 km before it needs to be refueled? An attempt to answer this question is the recursive SQL query of Figure 3. The query emits table hops(city, range) in which a row (c, r) indicates that we can reach city c with a residual range of r (see Figure 2). The result looks suspicious, though: we are able to reach Olney although the city is farther from the last fueling station (in Lewes) than our maximum reach.

Pursuing the observational debugging paradigm, users mark parts of the buggy query (■■■■■ in Figure 3) to learn about the evaluation of selected subexpressions. Markings typically start out large and then gradually zoom in on query details until the source of the bug can be observed directly. In keeping with the relational data model, the debugger presents observations in tabular form (Figure 4). Given the particular markings § to § of Figure 3, one row in the observation table shows our current location (column §) and the range available (possibly after refueling, column §) before we travel r.dist kilometers (column §).

Figure 1: A debugging scenario: a relational model of cities and their connecting roads, parts of the instance hidden behind (see Figure 2). Which cities can we reach from Alton?

Figure 2: Final (but incorrect) hops table. The | identifies one questionable output: we did not expect to reach Olney.
WITH RECURSIVE hops(city, range) AS (  
VALUES('Alton ', 0)
UNION ALL
SELECT r.there AS city , h.range + c.fuel * 100 >= r.dist AS range
FROM cities AS c, roads AS r, hops AS h
WHERE h.city = c.city
AND h.city = r.there
AND h.range + c.fuel * 100 >= r.dist
)

(3) Remove unlabeled input rows and run the observational debugger on the minimized database instance.

This input minimization will, in general, lead to significantly smaller observations: in the road network scenario, any city or road that does not lie on the path from Alton to Olney will be removed (see Figure 5 which features a mere 12 rows and hides nothing). Most importantly, the reduced input will still trigger the bug and any observation made will be relevant in identifying the bug’s cause—in a sense, after minimization the query will focus on producing the buggy output. We claim that this focus is just what is needed to effectively debug data-intensive computations.

2. MAKE EVERY OBSERVATION COUNT

Observations may be sizeable, however, and it can be a true challenge to spot enlightening details like \( \hat{d} \) in Figure 4. The sheer size of the input tables as well as the marking of subexpressions that are evaluated before aggregates or filters reduce data volume may lead to huge observations that do not reveal much. Indeed, in Figure 4 the lion’s share of our observations hides behind the ellipses (\( ··· \)), the majority of which contribute nothing to the understanding of the bug.

It is here where we propose to join two strands of work that have evolved independently until now. We build on a variant of data provenance analysis \([1,4]\) as follows:

(1) In the query output, users identify one or more suspect cells or rows (see Figure 2 where we use the mouse to identify the questionable city Olney).

(2) Provenance analysis infers those input table cells that are material in computing the value Olney (where provenance, cells labeled \( \hat{\bullet} \) in Figure 1(a)) as well as all rows that were inspected to decide that Olney is part of the query’s result (why provenance, label \( \hat{\circ} \)).

(3) Remove unlabeled input rows and run the observational debugger on the minimized database instance.

This input minimization will, in general, lead to significantly smaller observations: in the road network scenario, any city or road that does not lie on the path from Alton to Olney will be removed (see Figure 5 which features a mere 12 rows and hides nothing). Most importantly, the reduced input will still trigger the bug and any observation made will be relevant in identifying the bug’s cause—in a sense, after minimization the query will focus on producing the buggy output. We claim that this focus is just what is needed to effectively debug data-intensive computations.

Hand in Hand: Debugging and Provenance Analysis. The practical relevance of this research hinges on the ability to embrace expressive SQL dialects—featuring language constructs like correlation, grouping, window functions, recursion, as well as built-in and user-defined SQL functions. It is these rich queries that are potential sources of obscure bugs.

Our recent work on the efficient value-less interpretation of programs \([4]\) provides a means to derive where- and why-provenance for such real-world SQL dialects. We are underway to connect this analysis with the Habitat observational debugger for SQL \([2]\) and are positive to be able to make a significant step towards truly declarative and scalable query debugging.

3. REFERENCES


