A SQL Debugger Built from Spare Parts

Turning a SQL:1999 Database System into Its Own Debugger

Benjamin Dietrich Torsten Grust

Universität Tübingen
Tübingen, Germany

[b.dietrich, torsten.grust]@uni-tuebingen.de

ABSTRACT

We demonstrate a new incarnation of Habitat, an **observational debugger for SQL**. In observational debugging, users highlight parts of a—presumably faulty—query to observe the evaluation of SQL subexpressions and learn about the query's actual runtime behavior. The present version of Habitat has been redesigned from scratch and employs a query instrumentation technique that exclusively relies on the SQL facilities of the underlying RDBMS. We particularly shed light on new features like (1) the debugging of recursive SQL queries and (2) the observation of row groups (before and after aggregation). Habitat can turn any reasonably modern SQL:1999 RDBMS into its own language-level SQL debugger.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*Query languages*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms: Algorithms, Languages

Keywords: SQL; debuggers; observational debugging

1. AN OBSERVATIONAL SQL DEBUGGER

We demonstrate a completely rethought and redesigned version of Habitat, an **observational debugger for SQL**. Given the text of a—presumably buggy—SQL query, users mark subexpressions of arbitrary size and position for runtime observation. Much like the judicious placement of `printf()` calls in the debugging of procedural code, Habitat's observations help to gain detailed insight into the query evaluation process. Habitat uses the user's markings to transform and re-evaluate the instrumented query, collects the desired observations on the way, and finally prepares a tabular display of these observations and the context in which they occurred.

The logic and functional programming language communities pioneered this observational style of debugging [7, 9] which proves to be a good fit for the declarative SQL language as well. With Habitat, the debugging of logical (*not*: performance) SQL bugs happens on the level of the user-facing query language and its tabular data model. Users are not exposed to engine internals or query plans (they do not need to: the plans are okay—it is the query logic that needs fixing). The RDBMS host itself produces and collects the observations while it evaluates the instrumented query against the original database instance, *i.e.*, in the presence of built-in or user-defined functions, views, and types (the query is observed in "its own habitat".) Habitat's debugging model is orthogonal to those of existing SQL stored procedure debuggers [6] in which the evaluation of a query constitutes a monolithic and opaque action.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

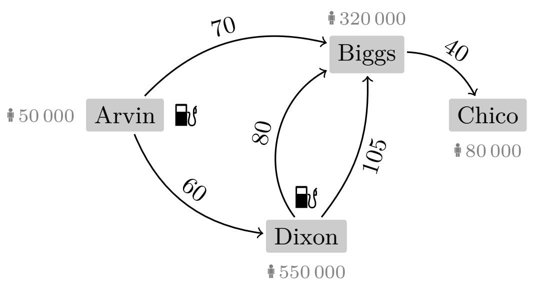
SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2735358.

A new Habitat, assembled from the spare parts around you. Earlier, we have described an implementation of the Habitat idea that relied on a rather complex and non-standard translation of SQL into an intermediate table algebra [3, 4]. With it came the requirement of an external code generator that would turn instrumented algebraic queries back into executable SQL text. Users were constrained in what kind of subexpressions could be observed and where markings could be placed.

The present work demonstrates a new incarnation of Habitat that has been *exclusively built using the host DBMS's own SQL facilities.* We focus on PostgreSQL [8] for this demonstration but the design immediately applies to other SQL:1999 database systems. The new Habitat debugger

- admits the free placement of markings and can observe subexpressions including atomic values, arithmetic and Boolean expressions, groups of rows before and after aggregation, (correlated) subqueries as well as recursive SQL queries,
- places a set of observations into a common context to illustrate the runtime interplay of individual query pieces,
- supports "*what if?*"–style debugging that allows to tinker with (sub)query results before a final fix is committed to,
- collects observations in relational tables as a side-effect during regular query evaluation, and
- comes with a visual browser-based frontend that supports subexpression marking, an interactive observation display, and allows to establish and keep focus on key observations.

The demonstration will feature a fully functional and live version of the new Habitat debugger running atop a PostgreSQL 9.3 backend. Here, we continue to briefly explore two SQL debugging scenarios. Our aim is to provide a flavor of the methodology of observational SQL debugging and to give an impression of what the demo audience can experience on-site.



(a) Road network with travel distances between cities, locations of fueling stations (⛽) and population size (🧍).

cities

	city	population	fuel
c_1	Arvin	50 000	1
c_2	Biggs	320 000	0
c_3	Chico	80 000	0
c_4	Dixon	550 000	1

roads

	here	dist	there
r_1	Arvin	70	Biggs
r_2	Arvin	60	Dixon
r_3	Biggs	40	Chico
r_4	Dixon	80	Biggs
r_5	Dixon	105	Biggs

(b) Tables cities and roads.

Figure 1: Simplified relational model of cities and their connecting road network.

```
 1 WITH RECURSIVE hops(city, gauge) AS (
 2     VALUES ('Arvin', 0)
 3   UNION ALL
 4     SELECT r.there AS city,
 5            h.gauge + c.fuel * 100 - r.dist AS gauge
 6     FROM   cities AS c, roads AS r, hops AS h
 7     WHERE  h.city = c.city
 8     AND    h.city = r.here
 9     AND    h.gauge + c.fuel * 100 >= r.dist
10 )
11 SELECT *
12 FROM   hops;
```

Figure 2: Which cities can be reached from Arvin? (Buggy.)

```
 1 WITH RECURSIVE hops(city, gauge) AS (
 2     VALUES ('Arvin', 0)
 3   UNION ALL
 4     SELECT r.there AS city,
 5            h.gauge + c.fuel * 100④ - r.dist③ AS gauge
 6     FROM   cities AS c, roads AS r, hops AS h
 7     WHERE  h.city = c.city                 ②
 8     AND    h.city = r.here
 9     AND    h.gauge + c.fuel * 100 >= r.dist
10 )                                               ①
11 SELECT *
12 FROM   hops;
```

Figure 4: Habitat allows the placement of markings (⬚) to observe the evaluation of selected SQL subexpressions.

2. UNRAVELING AND FIXING SQL BUGS

Let us focus on two debugging showcases that go beyond the capabilities of the initial Habitat version: (1) a recursive (or iterative) road network exploration, formulated using SQL's `WITH RECURSIVE` construct, and (2) a SQL query that revolves around the grouping and subsequent aggregation of rows. In both cases we obtain unexpected (read: incorrect) results and we will use Habitat to unravel the queries and then fix their SQL bugs.

The simple two-table database of Figure 1 serves as the scenario for both showcases. We model an acyclic network of unidirectional roads that connects four cities. In selected cities, marked with a ⛽ symbol in Figure 1(a), drivers have the opportunity to refuel their cars (tank volume: 100 ℓ). Traveling along a road $\xrightarrow{d}$ costs d liters of fuel. Figure 1(b) shows how this road network is represented in terms of relational tables. Table roads encodes the network itself while table cities holds city details: the presence of a gas station (0/1 in column fuel) as well as the current population size. (Ignore the c_i and r_j row identifiers for now.)

2.1 Recursion in SQL

Road network exploration. Start in Arvin with an empty tank, refill, then travel along any road requiring no more than 100 ℓ of fuel. Upon arrival, take note of the destination reached (city) as well as the fuel that remains (gauge). Refill at the destination if possible, then continue the journey. Which cities can we reach from Arvin?

The recursive SQL query reproduced in Figure 2 computes table hops(city, gauge) to answer this question. The base case starts us off in Arvin with an empty tank (line 2). In the recursive case, the hop(s) we reached last are available in row variable `h`. We join with tables cities and roads to learn about the destination city `c` and the roads `r` that leave from `c` (lines 6 to 8). Should the remaining fuel `h.gauge` plus a possible refill in `c`—if there is no gas station in `c`, `c.fuel * 100` = 0—bridge the distance `r.dist` (line 9), we have established `r.there` as a possible next hop. To complete this step of the recursion, we enter `r.there` along with the then remaining fuel into the hops table (lines 4 and 5). Evaluating this recursive query against the database of Figure 1(b) yields the hops table of Figure 3. The result looks suspicious. Row ⟨`Chico`, `20`⟩, in particular, is unexpected: there is no option to reach Biggs—where we cannot refill—with the residual 40 ℓ of fuel needed to bridge the distance to Chico. Row ⟨`Biggs`, `60`⟩ thus is questionable as well.

city	gauge
Arvin	0
Biggs	30
Dixon	40
Biggs	35
Biggs	60
Chico	20

Figure 3: Final hops table. (Incorrect.)

Debugging a recursive query. Since we are not sure where the problem with the query might lie, we start by *marking* the entire body of the `WITH RECURSIVE` clause. For reference, Figure 4 depicts the markings that we place during this debugging session (we have just placed Marking ①; Markings ② to ④ will be added in the process). Habitat instruments the SQL query according to Marking ①, runs the modified query, and responds with a tabular display of *observations* made at runtime (Figure 5).

Each cell in a Habitat observation reflects one evaluation (of many, in general) of a marked SQL subexpression. The observation of Figure 5 shows that the base case of the `WITH RECURSIVE` clause ran once while the recursive part was evaluated an additional three times (cf. the rows numbered 0 to 3). A cell marked ▨ indicates that an observed

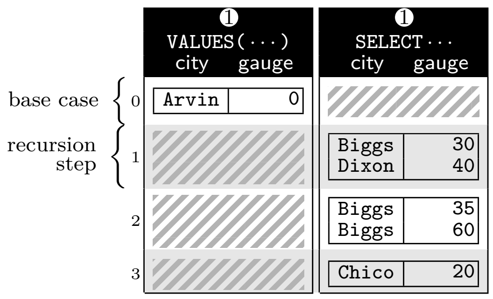


Figure 5: Observation for Marking ①. Left column relates to the `VALUES(···)` expression, right column shows the evaluations of the recursive `SELECT` clause (after `UNION ALL`).

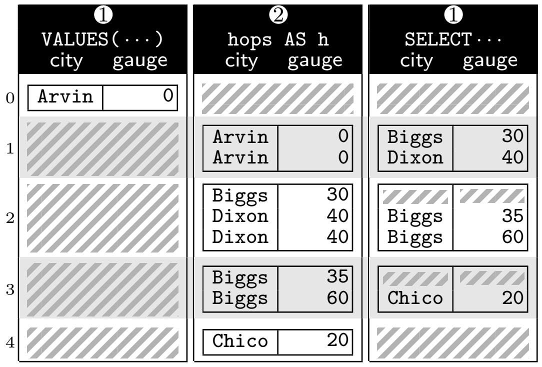


Figure 6: Observations for Markings ① and ② in context.

expression has not been evaluated in that particular recursive step (or iteration). Note that both, the `VALUES` expression as well as the recursive `SELECT`, yield *tables* of rows on each evaluation—Habitat renders such tabular values inside boxes ▭ for clarity. We see that Chico has been reached in the third (final) step of the recursion—as is expected since the city is located at the fringe of the road network.

To understand how we (erroneously) got to Chico, we place Marking ② which brings the hops themselves into context. Habitat now prepares the merged display of Figure 6 which relates the observations for Markings ① and ②. In recursion step 1 we reach both Biggs and Dixon coming from Arvin. In step 2, we do not get anywhere from Biggs with a tank of 30 ℓ. However, we can refill in Dixon to reach Biggs on two different roads. The remaining fuel levels in Biggs of 35 ℓ and 60 ℓ look suspiciously high, though. In fact, the residual 60 ℓ allows us to reach Chico in step 3. (Note that step 4 indicates that we get nowhere from Chico: row ⟨`Chico`, `20`⟩ does not find any join partner such that the `SELECT` clause of Marking ① is never evaluated; this indicates the end of the recursion according to the semantics of `WITH RECURSIVE` [1].)

Do we take the proper travel distances into account? We place Marking ③ to observe the atomic subexpression `r.dist`. Habitat grows the display once more (refer to the column labeled ③ in Figure 7). Checking with table roads, the distances look as expected. Does the fuel level calculation work correctly? We highlight expression `h.gauge + c.fuel * 100` to obtain Marking ④ and its associated observation (Figure 7). Here is the bug! Apparently, a refill may exceed the tank volume of 100 ℓ (in Dixon we fill up from 40 to 140 ℓ).

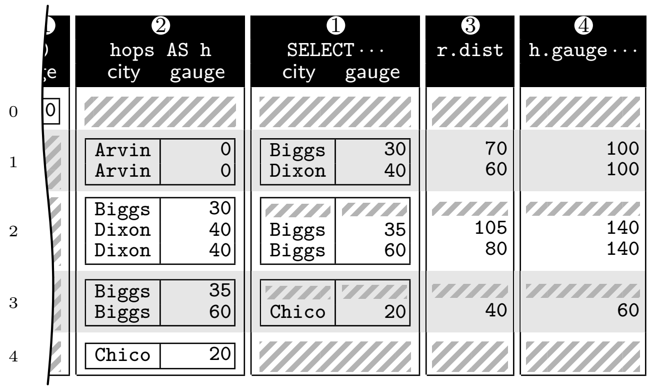


Figure 7: Display of observations for Markings ① to ④.

```
WITH RECURSIVE hops(city, gauge) AS (
   VALUES ('Arvin', 0)
  UNION ALL
   SELECT r.there AS city,
          LEAST(100, h.gauge + c.fuel * 100) - r.dist AS gauge
   FROM   cities AS c, roads AS r, hops AS h
   WHERE  h.city = c.city
   AND    h.city = r.here
   AND    LEAST(100, h.gauge + c.fuel * 100) >= r.dist
)
SELECT *
FROM   hops;
```

Figure 8: Repaired road network exploration query (fixes).

One obvious fix is to properly cap the fuel level on a refill. The SQL query of Figure 8 uses `LEAST(·,·)` to implement this. (Section 3 illustrates how Habitat allows to experiment with a capped fuel level *before* such a concrete query fix is applied.) Table hops of Figure 9 holds the correct result of the road network exploration.

city	gauge
Arvin	0
Dixon	40
Biggs	30
Biggs	20

Figure 9: Table hops. (Corrected.)

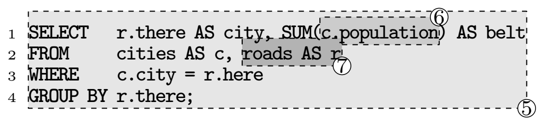
2.2 Observing Groups Before Aggregation

The `GROUP BY` construct and the groups of rows it produces are, in a sense, second-class concepts in SQL: queries can compute aggregates of a group but may not return the *group itself* (a restriction due to the flat 1NF nature of the relational data model). Nevertheless, a look at such groups of rows can be insightful during the debugging of queries featuring `GROUP BY`.[1] It can also help learners to understand the otherwise opaque behavior of `GROUP BY`. This second debugging session illustrates how the new Habitat treats groups as being first-class.

[1]Queries can be rewritten to reveal group contents but this manual process is error-prone [2, 5] and does not preserve the original query intent.

```
SELECT    r.there AS city, SUM(c.population) AS belt
FROM      cities AS c, roads AS r
WHERE     c.city = r.here
GROUP BY r.there;
```

Figure 10: Computing commuter belt sizes. (Buggy.)



```
SELECT   r.there AS city, SUM(c.population) AS belt
FROM     cities AS c, roads AS r
WHERE    c.city = r.here
GROUP BY r.there;
```

Figure 12: Markings placed to debug the query of Figure 10.

	❺ SELECT ···		❻
	city	belt	c.population
1	Biggs	1 150 000	50 000 550 000 550 000
2	Chico	320 000	320 000
3	Dixon	50 000	50 000

Figure 13: Observations made for Markings ⑤ and ⑥. Nested boxes in column ⑥ represent groups.

	❺ SELECT ···		❻	❼ roads AS r		
	city	belt	c.population	here	dist	there
1	Biggs	1 150 000	50 000 550 000 550 000	Arvin Dixon Dixon	70 80 105	Biggs Biggs Biggs
2	Chico	320 000	320 000	Biggs	40	Chico
3	Dixon	50 000	50 000	Arvin	60	Dixon

Figure 14: Complete display of observations for Markings ⑤ to ⑦. The double Dixon in iteration 1 hints at the bug.

```
SELECT   r.there AS city, SUM(c.population) AS belt
FROM     cities AS c,
         (SELECT DISTINCT here, there FROM roads) AS r
WHERE    c.city = r.here
GROUP BY r.there;
```

Figure 15: Patched commuter belt query (fix).

Commuter belt size. For each city with incoming roads, compute the sum of the population sizes of the city's direct neighbors. The result serves as a measure for the expected commuter traffic (useful input in road planning).

The query of Figure 10 uses tables cities and roads (Figure 1(b)) to implement this commuter belt computation in SQL. For each city with incoming roads—found in column there in table roads—we find all neighboring cities c (lines 2 and 3), collect them in a common group (line 4), and finally sum the neighbors' populations (line 1) to find the belt size (belt). The resulting two-column table is shown in Figure 11 but its contents look dubious: the commuter belt size of Biggs alone exceeds the overall population (1 000 000) of the entire region.

city	belt
Biggs	1 150 000
Chico	320 000
Dixon	50 000

Figure 11: Commuter belts. (Incorrect.)

Debugging with first-class groups. Once more we start the session by highlighting the entire SQL query text (Marking ⑤ in Figure 12). The resulting observation merely mirrors the query's result. It would be more insightful to know which cities' population sizes contribute to the obviously incorrect belt size aggregates. We thus place the Marking ⑥. Note that we highlight the subexpression `c.population` but omit the enclosing `SUM` aggregate function. Habitat instruments and re-runs the SQL query, then renders the observation display of Figure 13. In each of the three iterations of the query's `SELECT` clause, marked subexpression `c.population` evaluates to a *group of rows*: column ⑥ thus uses nested boxes ▭ to visualize the group contents.

The group associated with Biggs (iteration 1) indicates incoming traffic of 550 000 twice. Is that okay? Marking ⑦ on `roads as r` lets us observe the associated roads `r` that carry the traffic. The augmented display of Figure 14 indicates that we counted the population of Dixon twice when we explored the neighborhood of Biggs (two roads lead from Dixon to Biggs).

city	belt
Biggs	600 000
Chico	320 000
Dixon	50 000

Figure 16: Corrected commuter belts.

The `DISTINCT` fix of Figure 15 makes the SQL query disregard such multi-road connections between cities and establishes the intended notion of neighborhood. Figure 16 shows the corrected commuter belt sizes.

3. BEHIND THE SCENES

Instrumentation. Habitat's operation is based on the *instrumentation* of the SQL query under debugging. In a nutshell, a marked subexpression e is replaced by a function call $\mathtt{h}_i(\ldots, e)$. SQL function $\mathtt{h}_i$ inserts the value of e into Habitat–created table $\mathtt{obs}_i$ and otherwise acts as the identity by returning e, allowing the evaluation of the instrumented query to proceed normally.

Figure 17 shows the fully instrumented variant of the recursive road exploration query. The four markings of Figure 4 turn into calls to functions $\mathtt{h}_{1,\ldots,8}$ (if e is table-valued, the columns of e are instrumented separately). Function $\mathtt{h}_6$ and its associated observation table $\mathtt{obs}_6$ are shown in Figures 18 and 19, respectively (in these $\mathtt{obs}_i$ tables, the observed value of e is found in column observation). Note that $\mathtt{h}_i$ is declared as a `VOLATILE` SQL function to announce its side effect on the $\mathtt{obs}_i$ table, forcing the RDBMS to reevaluate $\mathtt{h}_i$ on every invocation.

Observations in context. Once the evaluation of the instrumented query is complete, Habitat's frontend reads from the observation tables $\mathtt{obs}_i$ to prepare the displays discussed in Section 2. Recall that these displays relate multiple observations by merging them into a coherent whole—a requirement of the observational debugging paradigm in which markings are added over time to establish a chain of cause and effects, eventually leading to the detection of the bug [7]. Along with the observed value of e, Habitat thus collects information that identifies the *context* in which e was evaluated. A display relates observations that share the same context.

Habitat derives context information from three sources (as required and applicable):

- for any SQL row variable v in scope, the identifier $\mathtt{TID}(v)$ of the row v is currently bound to,
- the values of the criteria $c_1, \ldots, c_n$ ($n \geqslant 1$) that uniquely identify each group built by a `GROUP BY` $c_1, \ldots, c_n$ clause (*e.g.*, $c_1 \equiv$ `r.there` in the query of Figure 12), and
- the depth $\mathtt{DEPTH()}$ of the current recursion ($\mathtt{DEPTH()} \in \{0, \ldots, 4\}$ for the example of Figures 6 and 7).

```
 1 WITH RECURSIVE hops(city, gauge) AS (
 2    SELECT  hops.city, hops.gauge
 3    FROM    (VALUES (DEPTH())) AS _(rec),
 4    LATERAL (SELECT h1(rec, TID(v), v.city) AS city, h2(rec, TID(v), v.gauge) AS gauge ①
 5             FROM   (VALUES ('Arvin', 0)) AS v(city, gauge)
 6            ) AS hops
 7  UNION ALL
 8    SELECT  hops.city, hops.gauge
 9    FROM    (VALUES (DEPTH())) AS _(rec),
10    LATERAL (SELECT h7(rec, TID(c), TID(r), h.id, r.there) AS city,
11                    h8(rec, TID(c), TID(r), h.id,
12                      h6(rec, TID(c), TID(r), h.id, h.gauge + c.fuel * 100) ④ - h5(rec, TID(c), TID(r), h.id, r.dist) ③) AS gauge ①
13             FROM   cities AS c, roads AS r,
14                    (SELECT TID(h) AS id, h3(rec, TID(h), h.city) AS city, h4(rec, TID(h), h.gauge) AS gauge ②
15                     FROM   hops AS h) AS h
16             WHERE  h.city = c.city
17             AND    h.city = r.here
18             AND    h.gauge + c.fuel * 100 >= r.dist
19            ) AS hops
20 )
21 SELECT hops.*
22 FROM   hops;
```

Figure 17: Full instrumentation of the recursive SQL query of Figure 4.

```
1 CREATE FUNCTION h6(rec BIGINT, c TID, r TID, h BIGINT,
2                    observation INT) RETURNS INT AS
3 $$
4   INSERT INTO obs6 VALUES (rec, c, r, h, observation);
5   SELECT observation; -- return value of e (act as identity)
6 $$
7 LANGUAGE SQL VOLATILE;
```

Figure 18: Function $\mathtt{h}_6$ implements Marking ④, observing expression $e \equiv$ `h.gauge + c.fuel * 100` of type `INT`.

$\mathtt{obs}_6$

rec	TID(c)	TID(r)	TID(h)	observation
1	c_1	r_1	1	100
1	c_1	r_2	1	100
2	c_4	r_4	1	140
2	c_4	r_5	1	140
3	c_2	r_3	2	60

Figure 19: Observation table $\mathtt{obs}_6$. Populated by function $\mathtt{h}_6$ when the instrumented query of Figure 17 is evaluated.

Contemporary SQL RDBMSs provide a variety of options to obtain these bits of context:

TID(v): If v ranges over a base table, identify rows by their system-provided row ids[2] (with PostgreSQL TID(v) $\equiv$ v`.ctid` and Oracle TID(v) $\equiv$ v`.rowid`, for example). Otherwise (v ranges over an intermediate subquery result), generate surrogate row ids via a SQL:1999 row number generator: TID(v) $\equiv$ `ROW_NUMBER() OVER ()`.

DEPTH(): Create a temporary SQL sequence generator s and draw successive numbers $0, 1, \ldots$ from s: DEPTH() $\equiv$ `NEXT VALUE FOR` s (for PostgreSQL: `nextval('`s`')`).

To illustrate, function $\mathtt{h}_6$ receives the current recursion depth as well as the identifiers of the rows bound to variables `c`, `r`, and `h` as context. Observation table $\mathtt{obs}_6$ holds the recorded context along with the result of the observed expression: value `60` was observed in the third recursion step when variables `c`, `r`, and `h` were bound to the rows with ids c_2, r_3, and 2, respectively (the row ids c_i, r_j are found in Figure 1(b); `h` ranges over `hops`, an intermediate result table—Habitat implements TID(h) (line 14 in Figure 17) in terms of surrogate row identifiers of type `BIGINT`).

[2]It suffices that these row ids are stable while the instrumented query is evaluated.

```
5 SELECT CASE WHEN (rec, c, r, h) IN ((2,c4,r4,1), (2,c4,r5,1))
             THEN 100
             ELSE observation
        END;
```

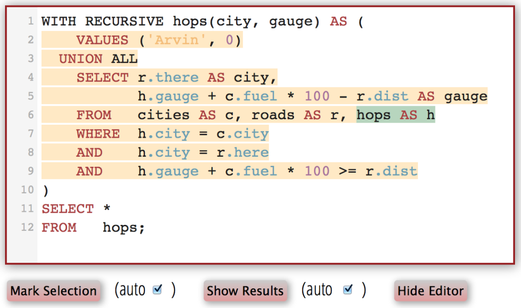
Figure 20: "What if?" debugging: replacement for line 5 in function $\mathtt{h}_6$ (Figure 18).

Instrumentation and (non-)interference. Instrumentation involves the risk of interfering with the evaluation of the debugged subject query. Habitat's query instrumentation diligently avoids such interference:

- If possible, context information is generated inside separate `LATERAL` subqueries: in `FROM` … e_c, `LATERAL` e, expression e_c can compute context information which Habitat may use to instrument expressions in subquery e (see lines 3 and 9 in Figure 17 where the current recursion depth is made available in column `rec`).
- If Habitat needs to introduce additional columns to hold context information (cf. column `id` in line 14), care is taken to not interfere with query evaluation: this relates to the expansion of `SELECT *`, duplicate elimination via `DISTINCT`, or the semantics of set operations (*e.g.*, `UNION`), for example.

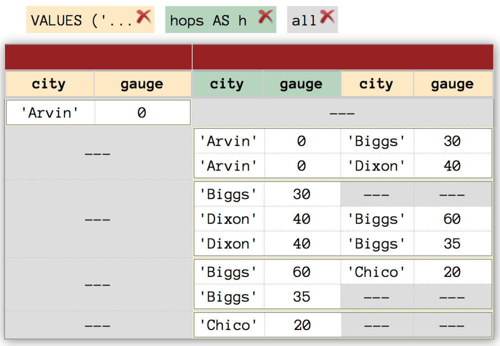
"*What if?*" debugging. The presence of evaluation context information enables Habitat to offer a "*what if?*"–approach to debugging in which users can tentatively alter the behavior of the subject query before a corresponding fix needs to be implemented. Reconsider the road exploration query where we have found the fuel gauge expression marked ④ to erroneously exceed the tank volume. *Would the query work if these values had not exceeded 100 ℓ?* To explore this hypothesis, users select the two values `140` in the display of Figure 7 and instead provide the desired value of `100`. When the subject query is re-evaluated, the marked subexpression will yield `100` for the two highlighted instances (but otherwise evaluate as before).

Under the hood, the frontend passes the context information associated with the selected occurrences of `140` to Habitat which then alters the definition of $\mathtt{h}_6$ (see Figure 20). Based on the evaluation context, the new (non-identity) function $\mathtt{h}_6$ will return `100` instead of the original observation made, effectively creating the desired "*what if?*" scenario.



```
WITH RECURSIVE hops(city, gauge) AS (
   VALUES ('Arvin', 0)
  UNION ALL
   SELECT r.there AS city,
          h.gauge + c.fuel * 100 - r.dist AS gauge
   FROM   cities AS c, roads AS r, hops AS h
   WHERE  h.city = c.city
   AND    h.city = r.here
   AND    h.gauge + c.fuel * 100 >= r.dist
)
SELECT *
FROM   hops;
```

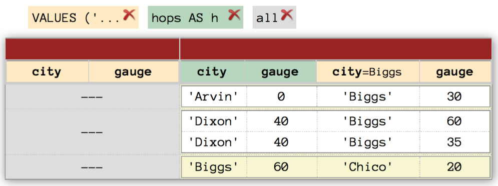
(a) Markings ① (🟧) and ② (🟩) placed in the road exploration query (refer to Figure 4).



city	gauge	city	gauge	city	gauge
'Arvin'	0	---			
---		'Arvin'	0	'Biggs'	30
		'Arvin'	0	'Dixon'	40
---		'Biggs'	30	---	---
		'Dixon'	40	'Biggs'	60
		'Dixon'	40	'Biggs'	35
---		'Biggs'	60	'Chico'	20
		'Biggs'	35	---	---
---		'Chico'	20	---	---

(b) Debugger display with observations ① and ②. Dashes (---) indicate non-evaluated expressions (▨ in Figure 6).

Figure 21: Screenshots of the interactive browser-based (HTML5, JavaScript) frontend to Habitat.



city	gauge	city	gauge	city=Biggs	gauge
---		'Arvin'	0	'Biggs'	30
---		'Dixon'	40	'Biggs'	60
		'Dixon'	40	'Biggs'	35
---		'Biggs'	60	'Chico'	20

Figure 22: Debugger display with all non-Biggs rows filtered. Interesting rows (🟨) remain in focus.

4. DEMONSTRATION SETUP

The on-site demo is based on a full-featured implementation of the new Habitat SQL debugger, talking to a PostgreSQL (version 9.3) backend. Habitat comes with a browser-based frontend that comprises a syntax-highlighting SQL editor as well as tabular displays that grow or shrink as debugging sessions progress. Markings can be placed and observation displays manipulated with minimal user interaction to encourage an exploratory (or playful) style of debugging: a single mouse drag suffices to place a marking and instantly update the display, for example. Figure 21 provides impressions of the debugger's new frontend. Not shown is Habitat's scenario archive which collects earlier debugging sessions.

Some markings may yield a potentially large number of rows. The debugger thus aims to help users to gain and retain focus during a session. *Column filters* may be used to hide all observations that fail to contain a given value. (Again, to promote rapid interaction, drop-down menus already provide the values in the column's active domain.) In addition, specific observations (*i.e.*, rows or groups of rows in a display) may be highlighted as *interesting*. Filters and highlights persist even if the debugger display changes dynamically—to implement this behavior, the frontend uses the observation contexts discussed in Section 3.

Figure 22 shows a snapshot of our first debugging session in which we have (1) identified the observation of Chico as interesting (*how did we get there?*) and, subsequently, (2) filtered all non-Biggs rows once we found that city was the last hop before we reached Chico.

Debugging scenarios. Based on audience interest and available time, we can offer a variety of scenarios that showcase the debugger and its implementation:

- *Quick demo:* uses a variant of the road network scenario in the present paper—simple queries and compact data set that still allow to explore interesting bugs.
- *Query understanding:* propose your own queries, then observe how (selected clauses of) these queries are evaluated—much like in a possible classroom use of Habitat.
- *SQL bug puzzle:* use Habitat to guide your hunt for particularly tricky bugs in SQL queries—based on canned queries held in Habitat's scenario archive.

The demonstration will be live: ad hoc SQL queries can be formulated and debugged at will. A log of instrumented queries along with their supporting functions $\mathtt{h}_i$ provides a peek under Habitat's hood.

Acknowledgments. Philipp Moers implemented the browser-based frontend for the new Habitat debugger.

5. REFERENCES

[1] ANSI/ISO. *Database Language SQL—Part 2: Foundation (SQL/Foundation)*. IEC 9075.

[2] R. Ganski and H. Wong. Optimization of Nested SQL Queries Revisited. *SIGMOD Record*, 16(3), 1987.

[3] T. Grust, F. Kliebhan, J. Rittinger, and T. Schreiber. True Language-Level SQL Debugging. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT)*, Uppsala, Sweden, 2011.

[4] T. Grust and J. Rittinger. Observing SQL Queries in their Natural Habitat. *ACM TODS*, 38(1), 2013.

[5] W. Kim. On Optimizing an SQL-like Nested Query. *ACM TODS*, 7(3), 1982.

[6] Microsoft Corporation. *Transact-SQL (T-SQL) Debugger in Microsoft SQL Server 2008*. `http://msdn.microsoft.com/en-us/library/cc645997.aspx`.

[7] B. Pope and L. Naish. Practical Aspects of Declarative Debugging in Haskell 98. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, Uppsala, Sweden, 2003.

[8] *The PostgreSQL Relational Database System*. `postgresql.org`.

[9] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA, 1983.