True Language-Level SQL Debugging

Torsten Grust Fabian Kliebhan Jan Rittinger Tom Schreiber
WSI, Universität Tübingen
Tübingen, Germany
⟨*firstname.lastname*⟩@uni-tuebingen.de

ABSTRACT

We demonstrate HABITAT, a declarative observational debugger for SQL. HABITAT facilitates true language-level (not: plan-level) debugging of, probably flawed, SQL queries that yield unexpected results. Users may mark arbitrary SQL subexpressions—ranging from literals, over fragments of predicates, to entire subquery blocks—to observe whether these evaluate as expected.

From the marked SQL text, HABITAT's algebraic compiler derives a new query whose result represents the values of the desired observations. These observations are generated by the target SQL database host itself. Prior data extraction or extra debugging middleware is not required.

HABITAT merges multiple observations into a single (nested) tabular display, letting a user explore the relationship of various observations. Filter predicates furthermore ease the interpretation of large results.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Query languages; D.2.5 [**Software Engineering**]: Debugging aids

Keywords

SQL, query debugger, observational debugging

1. DEBUGGING FLAWED SQL QUERIES

We built the observational SQL debugger HABITAT [6] that helps users to identify errors, or "bugs", buried in queries. With HABITAT, we pursue language-level debugging of logical flaws—that lead SQL queries to yield unexpected results or even runtime errors—and do *not* consider query engine or performance debugging.

HABITAT enables users to *mark* arbitrary suspect (or interesting) SQL subexpressions of a buggy query. Given such markings, HABITAT crafts new SQL queries that let the target RDBMS compute the value of the suspect subexpressions. Users *observe* and correlate these values, then narrow or widen their markings to hunt down the bug in an iterative, interactive process.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2011, March 22–24, 2011, Uppsala, Sweden.
Copyright 2011 ACM 978-1-4503-0528-0/11/0003 ...$10.00

```
FS  ::= SELECT SC AS ID,..,SC AS ID                 fullselects
          FROM TBL,..,TBL                            table access
        [ WHERE P ]                                  row filter
        [ GROUP BY COL,..,COL                        grouping
         [HAVING P]]                                 group filter
        [ ORDER BY SC,..,SC ]                        ordering

SC  ::= COL | ⟨SQL literal⟩                          scalars
      | SC + SC | SC * SC | ··· | ID(SC)             application
      | CASE WHEN P THEN SC ELSE SC END              conditional
      | COUNT(*) | COUNT(SC) | MAX(SC) | ···         aggregates
      | (FS)                                         scalar subquery

P   ::= P AND P | P OR P | NOT P | (P)               predicates
      | SC CMP SC                                    comparison
      | SC [NOT] IN (FS)                             membership
      | EXISTS(FS)                                   emptiness
      | SC CMP ALL (FS) | SC CMP ANY (FS)            quantification

TBL ::= ID(ID,..,ID) [AS ID]                         tables
      | (FS) AS ID                                   table subquery

CMP ::= < | <= | = | >= | > | <>                     comparison ops.

COL ::= ID[.ID]                                      column references

ID  ::= ⟨SQL identifier⟩                             identifiers
```

Figure 1: SQL fragment considered in this demonstration. Any SQL subexpression derivable from the non-terminals FS, SC, P, or TBL may be observed by the Habitat debugger.

HABITAT uses a context-free grammar for SQL to automatically extend arbitrary user-marked query text fragments to minimal syntactically complete subexpressions. Figure 1 shows the grammar that accepts the SQL dialect considered in this demonstration. Whereas non-terminal FS, defining a SQL fullselect [8, § 7.11], is the start symbol of this grammar, *any subexpression* that is derivable from the non-terminals FS, SC, P, or TBL is considered observable by the debugger. Non-terminal SC, for example, derives any scalar SQL expression, ranging from parenthesized nested fullselects to individual column references or literals.

Observing Queries in their Natural Habitat. HABITAT compiles a marking into a new SQL query based on the compilation rules described in [6] and submits this query to the target database host to collect observations based on the original instance data. The debugger does not depend on prior data extraction, extra middleware or specific software hooks: only an API for SQL query execution—here: JDBC—is required. HABITAT's approach allows to debug expressions in their original (remote) execution environment where the observing queries will find the exact set of built-in and user-defined functions specific to the target database host.

```
SELECT ps_partkey,ps_suppkey
  FROM Partsuppp
 WHERE ps_availqty <=
       (SELECT COUNT(*)
          FROM Lineitem
         WHERE ps_partkey = l_partkey
           AND ps_suppkey = l_suppkey)
```

(a) Original correct variant (Query 1).

```
SELECT ps_partkey,ps_suppkey
  FROM Partsupp,
       (SELECT l_partkey,l_suppkey,
               COUNT(*) AS cnt
          FROM Lineitem
         GROUP BY l_partkey,l_suppkey)
 WHERE ps_partkey = l_partkey AND ps_suppkey = l_suppkey
   AND ps_availqty <= cnt
```

(b) Uncorrelated yet buggy variant (Query 2).

Partsupp

ps_partkey	ps_suppkey	ps_availqty
1	10	0
2	20	1
2	30	4

Lineitem

l_orderkey	l_linenumber	l_partkey	l_suppkey
2	1	2	20
2	2	2	30
3	1	2	20

(c) Excerpt of a sample TPC-H database instance. This subset of tables and columns/rows suffices to illustrate the diverging behavior of Queries 1 and 2.

ps_partkey	ps_suppkey
1	10
2	20

(d) Result of Query 1.

ps_partkey	ps_suppkey
2	20

(e) Result of Query 2.

Figure 2: Two variants of the "out of supplies" SQL query (see (a) and (b)). Results in (d) and (e) differ when evaluated against the TPC-H instance of (c).

True Language-Level SQL Debugging. Contemporary SQL debuggers for RDBMSs implement a stateful paradigm that helps to monitor the execution of SQL stored procedures or scripts: variable updates and procedure call stacks are watched as the script advances line by line [4, 9]. The invocation of a SQL query from within a script, however, makes for a *monolithic action* that cannot be traced or inspected. Instead, HABITAT operates at the level of individual (suspect) SQL query subexpressions, promoting debugging at a considerably finer granularity.

Other debugging approaches typically expose the query engine's internal plan representation [1–3]. They are, however, of limited or no use in fixing logical flaws as the shape of a plan in most cases is largely disconnected from its surface syntax which is mostly due to complex query optimization logic. An *apriori* understanding of how user-facing query constructs emerge in algebraic plans is necessary to interpret such observations. In contrast, HABITAT makes use of the observational debugging paradigm on the level of *user-facing* SQL syntax and semantics: users mark fragments of their own SQL text and observe row variable bindings as well as expression values side by side with their dependencies in tabular form.

2. OBSERVING SQL SUBEXPRESSIONS

Consider a user's query (Query 1 of Figure 2a) which computes those parts for which we are out of supply: the available quantity (column ps_availqty) can not, or only barely, meet the current demand (read off table Lineitem). For the sample TPC-H instance of Figure 2c, the two parts represented by the rows with $\langle \text{ps_partkey}, \text{ps_suppkey} \rangle \in \{\langle 1, 10\rangle, \langle 2, 20\rangle\}$ are identified to be scarce (see Figure 2d).

Query 1 works flawlessly but exhibits disappointing performance for large database instances[1], which we attribute to the correlated aggregation carried out by the subquery. To remedy the issue, we rewrite Query 1, trading correlation for grouping [5], and obtain Query 2 of Figure 2b. While performance improves significantly, we find the rewritten query to not perfectly imitate the original: unexpectedly, part $\langle 1, 10\rangle$ is not considered to be out of supply by Query 2 (see Figure 2e). This is a bug whose cause we try to hunt down using HABITAT.

Debug Session. We start the session with the aim to reinforce our understanding of why the rows $\langle 1, 10\rangle$ and $\langle 2, 20\rangle$ have, correctly, been returned by Query 1. To do so, Marking ① is placed to observe whether the `COUNT(*)` aggregation computes the demand of parts as expected (Figure 3a). We further mark the `<=` predicate that embodies the "out of supply" condition (Marking ②): whenever this observation yields `true`, Query 1 has identified a scarce part.

Observations as Functions of Free Row Variables. HABITAT allows arbitrary query fragments to be marked for observation (see Section 1). In general, markings will contain and depend on *free row variables*—variables whose binding sites (`FROM` clauses) are not contained in the marking itself (*e.g.*, row variable v_0 is free in Marking ① of Figure 3a).

For any marked subexpression e, HABITAT consistently understands e as a *function of its free row variables.* Under this regime, Marking ① defines a function $f_{①}$ with

```
f①(v0) = (SELECT COUNT(*)
            FROM Lineitem AS v1
           WHERE v0.ps_partkey = v1.l_partkey
             AND v0.ps_suppkey = v1.l_suppkey)   ,
```

mapping rows v_0 (of table Partsupp) to tables with associated Lineitem rows. Generally, an observation for subexpression e reflects the set-oriented semantics of SQL and contains

- the values of e evaluated under *the set of all bindings* of its free row variables, and [*output*]
- the row values bound to these free variables (projected onto the columns actually referenced in e), [*input*]

i.e., a tabulation of the function defined by expression e. In programming language jargon, we obtain a tabular representation of the *closures* that capture the free variables and results of all evaluations of e.

As row variable v_0 is free in Marking ①, this defines function $f_{①}(v_0)$, mapping a part v_0 to a scalar of SQL type

[1]Again, this is *not* what we consider a bug in the context of the present discussion.

```
SELECT v0.ps_partkey, v0.ps_suppkey
  FROM Partsuppp AS v0
 WHERE v0.ps_availqty <=                          ②
       (SELECT COUNT(*)                          ①
          FROM Lineitem AS v1
         WHERE v0.ps_partkey = v1.l_partkey
           AND v0.ps_suppkey = v1.l_suppkey)
```

(a) **Markings placed to observe the evaluation of the "out of supply" (<=) predicate in Query 1.**

```
SELECT v0.ps_partkey④, v0.ps_suppkey⑤
  FROM Partsupp AS v0,
       (SELECT v2.l_partkey, v2.l_suppkey,
               COUNT(*) AS cnt
          FROM Lineitem AS v2
      GROUP BY v2.l_partkey, v2.l_suppkey) AS v1
 WHERE v0.ps_partkey = v1.l_partkey AND v0.ps_suppkey = v1.l_suppkey
   AND v0.ps_availqty <= v1.cnt③
```

(b) **A first set of subexpressions marked in Query 2.**

Figure 3: Possible markings that help track down the missing row bug.

v_0.ps_partkey	v_0.ps_suppkey	v_0.ps_availqty	① COUNT(*)	② · <= ·
1	10	0	0	true
2	20	1	2	true
2	30	4	1	false

Figure 4: Observations made for Markings ① and ②. The leading three columns show the projection of free row variable v_0 onto the columns actually referenced in the markings.

v_0.ps_availqty	v_1.cnt	③ · <= ·	④ v_0.ps_partkey	⑤ v_0.ps_suppkey
0	2	true		
0	1	true		
1	2	true	2	20
1	1	true		
4	2	false		
4	1	false		

Figure 5: Observations made for Markings ③ to ⑤.

INTEGER. Similarly, Marking ② defines a Boolean function $f_{②}(v_0)$ on parts v_0. Figure 4 shows the tabulation for both markings: the columns labeled ① and ② indicate the output; the projected columns v_0.ps_partkey, v_0.ps_suppkey, and v_0.ps_availqty mark the input.

Linking Multiple Observations. Markings ① and ② depend on the same free row variable v_0 (equivalently: functions $f_{①}$ and $f_{②}$ share the row parameter v_0) and thus Habitat *merges* the results of the associated observing queries into a single tabular display (Figure 4). More generally, Habitat merges observations whenever their associated sets of free row variables are contained in another (here, we have $\{v_0\} \subseteq \{v_0\}$). Merging related observations in this way greatly helps to understand the interplay of individual subexpressions in a larger query.

We observe the WHERE predicate to yield true two times, coinciding with Query 1's result cardinality of two, and understand that part $\langle 1, 10\rangle$ is considered "out of supply" because its availability (0 in column v_0.ps_availqty) does not exceed the current demand (the COUNT(*) aggregate also yields 0, column ①).

Debug Session (continued). We turn to the rewritten Query 2 and place Marking ③ to check whether its WHERE predicate mirrors the "out of supply" condition as expected (Figure 3b). This marking defines a Boolean function $f_{③}(v_0, v_1)$, capturing two free variables. The additional Markings ④ and ⑤ in the SELECT clause enable us to observe the resulting parts.

First, we see how the "out of supply" condition is evaluated against the combination of all bindings for v_0 and v_1 (Figure 5). This is a consequence of the *nested loop semantics* embodied by SQL FROM clauses that feature two or more row

```
SELECT v0.ps_partkey, v0.ps_suppkey
  FROM Partsupp AS v0,
       (SELECT v2.l_partkey, v2.l_suppkey,           ⑦
               COUNT(*) AS cnt
          FROM Lineitem AS v2
      GROUP BY v2.l_partkey, v2.l_suppkey) AS v1
 WHERE v0.ps_partkey = v1.l_partkey AND v0.ps_suppkey = v1.l_suppkey   ⑥
   AND v0.ps_availqty <= v1.cnt
```

Figure 6: More suspect subexpressions marked in Query 2.

v_0.ps_partkey	v_0.ps_suppkey	v_1.ps_partkey	v_1.ps_suppkey	⑥ · AND ·
1	10	2	20	false
1	10	2	30	false
2	20	2	20	true
2	20	2	30	false
2	30	2	20	false
2	30	2	30	true

Figure 7: Observations made for Marking ⑥, focus on the bindings with $\langle v_0.\mathsf{ps_partkey}, v_0.\mathsf{ps_suppkey}\rangle = \langle 1, 10\rangle$.

variables [8, § 7.5]. Multiple bindings qualify (value true in column ③) but only $\langle 2, 20\rangle$ makes it into the final result. For all other bindings, we observe that the two subexpressions in the SELECT clause are not evaluated at all, indicated by ▨ in columns ④ and ⑤. Those bindings—including the missing bindings with $\langle v_0.\mathsf{ps_partkey}, v_0.\mathsf{ps_suppkey}\rangle = \langle 1, 10\rangle$—must fail to satisfy the foreign key join predicate $v_0.\mathsf{ps_partkey} = v_1.\mathsf{l_partkey} \wedge v_0.\mathsf{ps_suppkey} = v_1.\mathsf{l_suppkey}$ in Query 2.

This join predicate thus is the subject of our next Marking ⑥ (Figure 6). The associated observation $f_{⑥}(v_0, v_1)$ shows the evaluation of the predicate against all combinations of v_0, v_1 bindings, so we let Habitat focus the display on the bindings that we miss (Figure 7). As suspected, the row variable bindings in focus find no join partner (false values in the focus on column ⑥). The grouping subquery appears to not generate bindings with $\langle v_2.\mathsf{l_partkey}, v_2.\mathsf{l_suppkey}\rangle = \langle 1, 10\rangle$ at all. This is exactly what our final Marking ⑦ and the associated observation (Figure 8) indicates.

We have finally uncovered that the rewrite from Query 1 to Query 2 perpetrates the *count bug* [7]. This notorious class of bugs—related to the semantics of grouping and aggregation over empty row sets (GROUP BY yields no row at

⑦ v_2.l_partkey	v_2.l_suppkey	cnt
2	20	2
2	30	1

Figure 8: A closed observation made in Query 2. No row is produced for part $\langle v_2.\mathsf{l_partkey}, v_2.\mathsf{l_suppkey}\rangle = \langle 1, 10\rangle$.

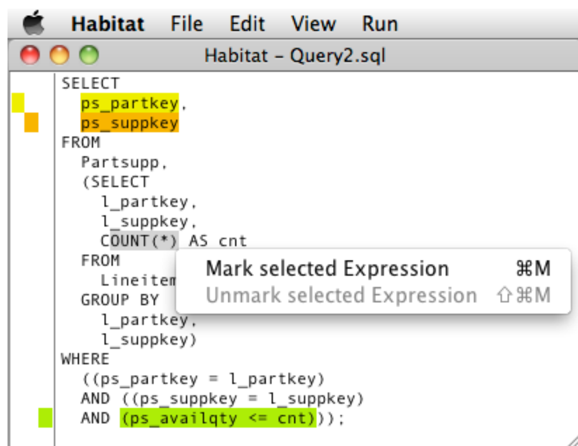

Figure 9: Screenshot of Habitat's *mark* window (compare with Figure 3a). Markings are color-coded and can be added by means of the context menu and keyboard shortcuts.

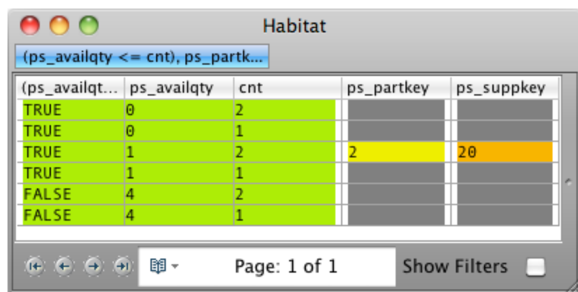

Figure 10: Screenshot of Habitat's *observe* window (compare with Figure 5). Colors link observations and markings. Pagination and filter predicates ensure intelligible results.

all whereas `COUNT(*)` returns `0`)—went unidentified for years before its cause and fix were described [5].

3. MARK AND OBSERVE (DEMO SETUP)

HABITAT's *mark and observe* implements a debugging paradigm via a closure-based SQL compiler, permitting users to think in terms of SQL's surface syntax (*mark*) and simple tabular representations of evaluated expressions (*observe*).

HABITAT's GUI consequently consists of a *mark* window providing a SQL editor panel and an *observe* window in which the function tabulations are rendered. Markings and their associated observations are color-coded (Figures 9 and 10). Arbitrarily placed selections of SQL text are automatically turned into meaningful markings (Section 1).

Debugging a query against original instance data can be vital to hunt down specific data-dependent bugs, but may yield large observations. To address such cases, HABITAT's tabular display uses *pagination* to split up large results and furthermore lets users formulate *filter predicates* on the result columns, thus reducing the size of the observation (see bottom pane in Figure 10).[2]

Demonstration Setup. The live demonstration features a DB2 database with tables of varying size giving an impression of HABITAT's interactivity. A look behind the scenes allows for a peek at the generated observing queries. We furthermore prepared three different use cases for HABITAT:

[2]These filters are a variant of the focus displayed in Figure 7.

- *Debugging of flawed queries.* Various buggy example queries await to be analyzed and fixed with (or without) the support of HABITAT.
- *Understanding complex queries.* Getting a clear idea of what a query does sometimes is not easy at all. With HABITAT we provide a convenient means to understand query fragments, without accidentally modifying the queries' semantics. HABITAT assists to unravel the meaning of various example queries.
- *Teaching SQL.* What is the exact meaning of query correlation? Which columns are available after a `GROUP BY` clause? HABITAT eases the understanding of SQL by supplying hooks to observe the various stages of the query evaluation.

Summary. Authoring bug-free SQL queries can be tricky at times and the language clearly deserves a debugging approach that fits its calculus-style computational model: the iterated evaluation of expressions under varying row variable bindings. HABITAT implements such a debugging paradigm. In summary, HABITAT can observe

- expressions of all scalar SQL data types, including expressions of type `BOOLEAN` (Markings ① to ⑥),
- (possibly empty) table-valued subexpressions (⑦),
- expressions that yield runtime errors (*e.g.*, violations of SQL's scalar subquery constraints) if evaluated in the context of the original query,
- closures (① to ⑥) as well as closed (or constant) expressions (⑦),
- related expressions, merging their observations if the associated sets of free variables are contained in one another (①, ② and ③, ④, ⑤),
- expressions that might not be evaluated for specific variable bindings (④, ⑤), and
- expressions that contain free variables originating in separate, yet nested, subexpressions.

Acknowledgments. This research is supported by the German Research Council (DFG) under grant GR 2036/3-1.

4. REFERENCES

[1] H. Bati, L. Giakoumakis, S. Herbert, and A. Suma. A Genetic Approach for Random Testing of Database Systems. In *Proc. VLDB*, 2007.

[2] C. Binnig and D. Kossmann. Reverse Query Processing. In *Proc. ICDE*, 2007.

[3] N. Bruno, S. Chauduri, and R. Ramamurthy. Interactive Plan Hints for Query Optimization. In *Proc. SIGMOD*, 2009.

[4] *The Embarcadero SQL Debugger.* `http://www.embarcadero.com/products/debugger`.

[5] R.A. Ganski and H.K.T. Wong. Optimization of Nested SQL Queries Revisited. *ACM SIGMOD Record*, 16(3), 1987.

[6] T. Grust and J. Rittinger. Observing SQL Queries in their Natural Habitat. 2010. Submitted.

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

[8] *Database Language SQL—Part 2: Foundation (SQL/Foundation).* ANSI/ISO/IEC 9075, 1999.

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