A

Observing SQL Queries in their Natural Habitat

TORSTEN GRUST, Universität Tübingen
JAN RITTINGER, SAP AG

We describe 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 mark SQL subexpressions of arbitrary size and then observe whether these evaluate as expected. HABITAT understands query nesting and free row variables in correlated subqueries, and generally aims to not constrain users while suspect subexpressions are marked for observation.

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 and are derived from the original data: HABITAT does not require prior data extraction or extra debugging middleware. Experiments with TPC-H database instances indicate that observations impose a runtime overhead sufficiently low to allow for interactive debugging sessions.

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

Additional Key Words and Phrases: Declarative debugging, query languages, relational databases, SQL

ACM Reference Format:
Grust, T., Rttinger, J. YYYY. Observing SQL Queries in their Natural Habitat. ACM Trans. Datab. Syst. V, N, Article A (January YYYY), 33 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. OBSERVATIONAL SQL DEBUGGING

We discuss the design and internals of the observational SQL debugger HABITAT that helps users to identify errors, or "bugs", buried in queries. In particular, we pursue debugging of logical flaws that lead SQL queries to yield unexpected results or even runtime errors. We do *not* consider query engine or performance debugging here.

A correct program is built from correct pieces. *Observational debugging* [Silva 2011; Pope and Naish 2003; Marlow et al. 2007] builds on this basic insight and promotes a debugging paradigm in which users observe program pieces to validate the elementary assumption that the pieces evaluate as expected. In this work, we apply the principles of observational debugging to SQL. Here, the *pieces of a SQL query* are its subqueries (if any) and, at a considerably finer granularity, its individual subexpressions. Ob-

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

Authors' addresses: T. Grust (corresponding author), Wilhelm Schickard Institute for Computer Science, Universität Tübingen, Germany; e-mail: torsten.grust@uni-tuebingen.de; J. Rittinger, SAP AG, Germany.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

```
SELECT p_partkey,p_name,
       (SELECT ps_supplycost
          FROM Partsupp
         WHERE ps_partkey = p_partkey)
  FROM Part
```

Fig. 1. A flawed SQL query that raises a runtime error for some TPC-H database instances (Query 1).

servational debugging with its focus on the evaluation of expressions is a particularly good fit for SQL: SQL's non-procedural set-oriented semantics forbid a traditional "gdb-stlye" debugging approach in which the query execution runtime is traced as it changes state.

Observational debugging depends on the query author having an intended interpretation of the query. If this intention is not met by the query result, one option is to work backwards from the erroneous output, identifying and observing those subexpressions that may contribute to the error. The identification part of the process can be automated, *e.g.* in terms of *algorithmic debugging*, see Section 5. Here, we focus on the observation of subexpressions.

The nature of SQL may render it difficult for users to observe whether a given individual query piece functions as anticipated. Observational debugging requires to turn the original query "inside out" such that the piece of interest appears at the query's top-level—only then the piece becomes observable.

To illustrate this challenge, consider the following debug session that, for now, does without HABITAT.

Manual Debug Session 1 (Unexpected Runtime Error). Query 1 of Figure 1 extracts parts and their supply cost from a TPC-H database [Transaction Processing Performance Council]. The query works as expected for some database instances but more often than not bails out with a runtime error—it does fail for the small TPC-H instance of Figure 4, for example. IBM DB2 V9 emits the error message:

```
SQL0811N  The result of a scalar fullselect,
SELECT INTO statement, or VALUES INTO statement
is more than one row.
```

The mention of "`scalar fullselect`" suggests that the nested subquery in the outer `SELECT` clause of Query 1 is the root of the runtime error. We did not expect this subquery to violate SQL's single-row cardinality constraint [ANSI/ISO, § 7.14]. Unfortunately, we cannot simply extract the subquery text between `(···)` and then execute and observe it on its own: the correlated subquery contains a reference to column p_partkey whose originating table Part has just been removed with the extraction.

At this point we need to *modify the original* subquery to obtain any observation at all. Among the many possible edits, we choose to remove the `WHERE` clause and with it the problematic reference to column p_partkey. Instead, we use grouping and resort to a `COUNT` aggregate to check whether the original subquery could possibly yield more than a single row for *any* value of ps_partkey:

```
SELECT ps_partkey,COUNT(ps_supplycost)
  FROM Partsupp                                    (∗)
 GROUP BY ps_partkey .
```

Indeed we find the cause of the runtime bug: for ps_partkey $= 3$, the row count is 2 (see Figure 4, bottom two rows of table Partsupp). □

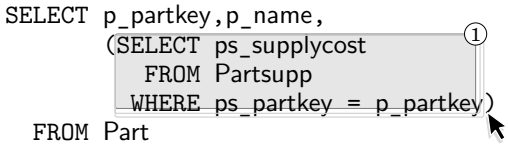


Fig. 2. Placing Marking ① (via mouse dragging) to observe the suspect subquery in Query 1.

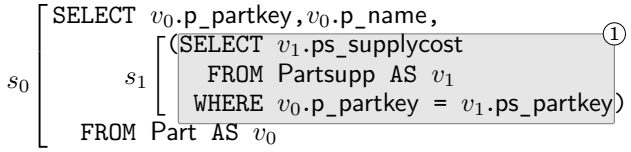


Fig. 3. The row variables v_0 (bound in scope s_0) and v_1 (bound in s_1) are made explicit in an internal variant of Query 1.

Already for this simple example, note that the original query piece (the subquery in Figure 1) and its modification ($*$) clearly deviate. In general, it will be far from obvious whether this manual piece extraction and editing process yields an observation that properly reflects the relevant query semantics (here: the subquery's row count). Additionally, the user is asked to correctly interpret the modified query's output.

HABITAT: *Mark and Observe*. Generally, it is non-trivial to turn a query's inside out to make its pieces observable for debugging. HABITAT's main objective thus is to implement a declarative debugging approach that lets users *mark* arbitrary suspect (or interesting) pieces of a buggy query. A marking may comprise large query fragments and can cut across correlated subqueries, but might also focus on just a predicate or an individual scalar expression. Given such markings, HABITAT crafts relational queries that let the target RDBMS itself compute the value of the suspect pieces based on the original database instance data. Users *observe* and verify these values, and then possibly narrow or widen their markings to hunt the bug in an interactive, iterative fashion.

The debugger enables users to reassure themselves of the workings of arbitrary small or large query pieces. The workings of a single piece are much more easily understood and assessed than those of an entire complex query [Motro 1986]. In particular, the evaluation of a given query piece may succeed (and thus be observable) while its containing query may fail to deliver any result at all.

This second debug session tries to track down the runtime bug of Query 1 with the help of HABITAT.

Debug Session 2 (Unexpected Runtime Error). To investigate the suspect piece, we *mark the subquery text between* $(\cdots)$ to obtain Marking ① (see Figure 2) and then invoke the debugger. HABITAT's internal operation is based on a variant of Query 1 in which SQL row variables are introduced and referenced explicitly (Figure 3). Here, it is immediate that the value of the marked subquery depends on variable v_0 whose binding site `FROM Part AS` v_0 lies outside the subquery, (in scope s_0). The value of a marked piece can only be computed if such dependencies on enclosing scopes are resolved. HABITAT thus constructs an *observer query* and evaluates it on the underlying database system to collect the data required to observe the suspect expression side by side with its dependencies.

HABITAT responds with a tabular display of the observation made for Marking ① (Figure 5). We now see that, for the particular TPC-H instance of Figure 4, the marked piece is evaluated four times under varying values $(1, \ldots, 4)$ bound to v_0.p_partkey. For

Part

p_partkey	p_name	p_mfgr
1	black	Manufacturer#2
2	ivory	Manufacturer#3
3	azure	Manufacturer#1
4	steel	Manufacturer#2

Partsupp

ps_partkey	ps_suppkey	ps_availqty	ps_supplycost
1	30	5	20.00
2	10	0	10.00
3	20	1	30.00
3	30	4	10.00

Lineitem

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

Fig. 4. Excerpt of a sample TPC-H database instance (focus on the subset of tables, columns, and rows that suffices to illustrate the SQL query bugs). Key columns are underlined.

①

p_partkey	SELECT ps_su ···
1	ps_supplycost: 20.00
2	ps_supplycost: 10.00
3	ps_supplycost: 30.00, 10.00
4	ps_supplycost:

Fig. 5. Observation made for Marking ①.

the row v_0 with v_0.p_partkey $=$ 3, we observe the piece ① to unexpectedly return *two* rows in violation of SQL's single-row cardinality constraint. Indeed, table Partsupp lists two distinct suppliers (column ps_suppkey) for this particular part v_0. A possible fix for Query 1 thus extends the subquery's `WHERE` clause, restricting ps_suppkey to refer to one specific supplier. □

In a sense, markings and observations realize an intuitive "`printf`-style" approach to SQL debugging. Acknowledging the relational data model, HABITAT calls on the underlying database system itself to produce and collect the debugging output in a table before it is presented to the user.

In the context of this work, we define *language-level debugging* as a process that supports all debugging activity on the level of the *user-facing* SQL syntax and semantics: users mark fragments of their own SQL text and observe piece values in tabular form, *i.e.* in the form defined by the relational data model itself. Users are not expected to inspect and understand the target RDBMS's query execution process.

Contributions. Our specific contributions are these:

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

ps_partkey	ps_suppkey
2	10
3	20

(a) Original correct variant (Query 2).

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

$\neq$

ps_partkey	ps_suppkey
3	20

(b) Uncorrelated yet buggy variant (Query 3).

Fig. 6. Two variants of the "out of supplies" SQL query and their respective results. Outcomes differ when evaluated against the TPC-H instance of Figure 4.

— We bring observational debugging to SQL for the first time. This enables a debugging approach that fits the declarative nature of SQL whose semantics is based on the evaluation of expressions under variable bindings (Section 2).
— We describe a new breed of SQL compiler with a focus on expression observation rather than performance (Section 3). Users operate (*i.e.* mark and observe) on the level of the syntax and semantics of SQL itself and are shielded from the complex details of query execution.
— Building on the new SQL compiler, we implement expression observation purely in terms of relational queries (Section 3.4). These queries are submitted to the target database host to collect observations based on the original instance data. There is no need for prior data extraction or preparation.
— HABITAT supports the interactive debugging of queries against large data instances (Section 4). Predicates may be applied to filter instance data during debugging. Alternatively, small yet representative sample data can be specifically generated for the purpose of debugging (Section 4.1).
— We built a debugger that does not depend on extra middleware or specific software hooks: no API beyond SQL query execution—in the style of ODBC, say—is required. (Interactive debug sessions may run remotely to minimize impact on a live target database host.) HABITAT's approach allows to debug expressions in their original execution environment where observations will find the exact set of built-in and user-defined functions specific to the target database host.

2. MARKINGS AND OBSERVATIONS

HABITAT aims to not constrain users when query text pieces are marked for observation. In general, then, markings will contain and depend on *free row variables*. The binding site (FROM clause) of a free variable is not contained in the marking itself but resides in an enclosing scope. Row variable v_0 is bound in the enclosing scope s_0 and thus free in Marking ① of Figure 3, for example.

For any marked subexpression e, HABITAT understands e as a *function of its free row variables*. Under this regime, Marking ① defines a function $f_①$ with

$$f_①(v_0) = \texttt{(SELECT } v_1\texttt{.ps\_supplycost FROM Partsupp AS } v_1 \texttt{ WHERE } v_0\texttt{.p\_partkey = } v_1\texttt{.ps\_partkey)} \quad ,$$

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

— the bindings of the free row variables (here: v_0) under which e is evaluated, [*input*] and
— the value of e for each of these bindings. [*output*]

HABITAT uses a straightforward tabular representation to show the input/output behavior of function $f_{①}$:

> *Tabulation.* Together, *input* and *output* constitute a *tabulation* of the function defined by expression e. In these tabulations, we project the input onto the relevant columns actually referenced in e.[1]

Figure 5 shows this tabulation for Marking ①: the column labeled ① indicates the output of function $f_{①}$, column p_partkey shows its input. A debug session tracks down those expressions for which the observed input/output behavior does not match the query author's expectations.

Effectively, HABITAT implements a model of the SQL semantics [Gogolla 1990] in terms of SQL itself. To facilitate observational debugging, the model includes the bindings of row variables as well as the value of any SQL subexpression under these bindings. These two components of the model are represented as tabulations, *i.e.* relational structures that can be built using SQL itself (*meta-interpretation* [Naish 1997]). The relational database host itself can thus be used to realize the debugger.

In the following we describe another debug session to illustrate how this tabulation principle applies to

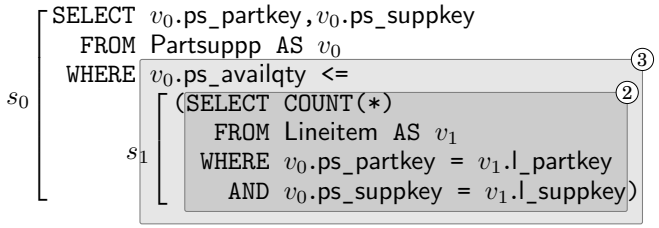
(1) expressions of all scalar SQL data types, including expressions of type `BOOLEAN`,
(2) (possibly empty) table-valued subexpressions,
(3) related expressions, whose tabulations can be merged if their sets of free variables are contained in one another, and
(4) expressions that might not be evaluated at all for specific variable bindings.

Debug Session 3 (Missing Rows after Rewrite). Again referring to the sample TPC-H database of Figure 4, Query 2 of Figure 6(a) 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 (which we read off table Lineitem). For the sample instance, the two parts represented by the rows with $\langle \text{ps\_partkey}, \text{ps\_suppkey} \rangle \in \{\langle 2, 10\rangle, \langle 3, 20\rangle\}$ are identified to be scarce (see the table in Figure 6(a)).

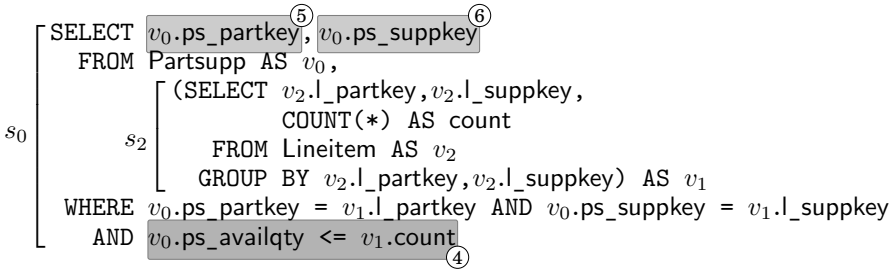
Query 2 works as expected. For large database instances, however, the performance is disappointing.[2] We attribute this to the correlated aggregation carried out by the subquery. To remedy the issue, we rewrite Query 2, trading correlation for grouping [Ganski and Wong 1987], and obtain Query 3 of Figure 6(b). While performance improves significantly, we find the rewritten query to not perfectly imitate the original: unexpectedly, part $\langle 2, 10\rangle$ is not considered to be out of supply by Query 3 (see the table in Figure 6(b)). This is a bug whose cause we try to hunt down using HABITAT.

We start the debug session with the aim to reinforce our understanding of why the rows $\langle 2, 10\rangle$ and $\langle 3, 20\rangle$ have, correctly, been returned by Query 2. To do so, Marking ②

[1] Put differently, we obtain a tabular representation of the *closures* [Landin 1964] that capture the free variables and results of all evaluations of e. (We return to this programming-language-inspired view in Section 5.)
[2] Again, this is *not* what we consider a bug in the context of the present discussion.



(a) Markings placed to observe the evaluation of the "out of supply" ($\cdot$ <= $\cdot$) predicate in Query 2.



(b) A first set of subexpressions marked in Query 3.

Fig. 7. Possible markings that help track down the missing row bug (scopes s_0–s_2 shown here to aid the exposition).

			②	③
v_0.ps_partkey	v_0.ps_suppkey	v_0.ps_availqty	COUNT(*)	$\cdot$ <= $\cdot$
1	30	5	1	false
2	10	0	0	true
3	20	1	2	true
3	30	4	1	false

Fig. 8. Observations made for Markings ② and ③. The leading three columns show the projection of free row variable v_0 onto the columns actually referenced in the markings.

		④	⑤	⑥
v_0.ps_availqty	v_1.count	$\cdot$ <= $\cdot$	v_0.ps_partkey	v_0.ps_suppkey
5	1	false		
5	2	false		
5	1	false		
0	1	true		
0	2	true		
0	1	true		
1	1	true		
1	2	true	3	20
1	1	true		
4	1	false		
4	2	false		
4	1	false		

Fig. 9. Observations made for Markings ④ to ⑥.

is placed to observe whether the `COUNT(*)` aggregation computes the demand of parts as expected (Figure 7(a)). As row variable v_0 is free in the marking, this defines function $f_{②}(v_0)$, mapping a part v_0 to a scalar of SQL type `INTEGER`, the current demand for that part. We further mark the `<=` predicate that embodies the "out of supply" condition (Marking ③). Whenever this observation yields `true`, Query 2 has identified a scarce part v_0: its available quantity v_0.ps_availqty is less than or equal to the current demand. Marking ③ thus defines a Boolean function $f_{③}(v_0)$ on parts v_0.

As Markings ② and ③ depend on the same free row variable v_0 (equivalently: functions $f_{②}$ and $f_{③}$ share the row parameter v_0), HABITAT *merges* the results of the associated observer queries into a single tabulation (Figure 8). More generally, HABITAT merges observations whenever their 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 (see Appendix B).

With Marking ③ we observe the `WHERE` predicate to yield `true` two times, coinciding with Query 2's result cardinality of two, and understand that part $\langle 2, 10\rangle$ 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 ②).

We expect the `WHERE` predicate in the rewritten Query 3 to realize the "out of supply" condition in the same fashion and place Marking ④ to observe its evaluation (Figure 7(b)). This marking defines a Boolean function $f_{④}(v_0, v_1)$, capturing two free variables. The additional Markings ⑤ and ⑥ in the `SELECT` clause, both functions of v_0, will enable us to observe the resulting parts. All three markings are related in terms of their free row variables ($\{v_0\} \subseteq \{v_0\} \subseteq \{v_0, v_1\}$), so HABITAT prepares a merged tabulation.

The resulting observation is telling (Figure 9). First, we see how the "out of supply" condition is evaluated against the combination of all bindings for v_0 and v_1. This is a consequence of the *nested loop semantics* embodied by SQL `FROM` clauses that feature two or more row variables [ANSI/ISO, § 7.5]. Multiple bindings qualify (`true` values in column ④) but only $\langle v_0.\text{ps\_partkey}, v_0.\text{ps\_suppkey}\rangle = \langle 3, 20\rangle$ appears to make 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 ⑤ and ⑥. Those bindings—including the expected but missing bindings with $\langle v_0.\text{ps\_partkey}, v_0.\text{ps\_suppkey}\rangle = \langle 2, 10\rangle$—must fail to satisfy the foreign key join predicate in Query 3.

This join predicate thus is the subject of our next Marking ⑦ (Figure 10). The associated observation $f_{⑦}(v_0, v_1)$ shows the evaluation of the predicate against all combinations of v_0, v_1 bindings, so we let HABITAT *focus* the display on those bindings that we miss (focus predicates are discussed in Section 4.1). As suspected, the row variable bindings in focus find no join partner (Figure 11, `false` values in the highlighted rows in column ⑦). The grouping subquery appears to not generate bindings with $\langle v_2.\text{l\_partkey}, v_2.\text{l\_suppkey}\rangle = \langle 2, 10\rangle$ at all. This is exactly what our final Marking ⑧ and the associated observation (Figure 12) indicates.[3]

Using HABITAT we have finally uncovered that the rewrite from Query 2 to Query 3 disregards the subtle semantics of grouping and aggregation over empty row sets (`GROUP BY` yields no row at all whereas `COUNT(*)` returns `0`). In fact, the rewrite introduced an instance of the *count bug* [Kim 1982], a notorious class of bugs that went unidentified for years before its cause and fix were described [Ganski and Wong 1987]. □

[3]Marking ⑧ is *constant* or *closed*, containing no free variables from enclosing scope s_0.

```
s0 ⌈ SELECT v0.ps_partkey, v0.ps_suppkey
     FROM Partsupp AS v0,
          s2 ⌈ (SELECT v2.l_partkey, v2.l_suppkey,          ⑧
                      COUNT(*) AS count
                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   ⑦
       AND v0.ps_availqty <= v1.count
```

Fig. 10. More suspect expressions marked in Query 3.

v_0.ps_partkey	v_0.ps_suppkey	v_1.ps_partkey	v_1.ps_suppkey	⑦ · AND ·
1	30	1	30	true
1	30	3	20	false
1	30	3	30	false
2	10	1	30	false
2	10	3	20	false
2	10	3	30	false
3	20	1	30	false
3	20	3	20	true
3	20	3	30	false
3	30	1	30	false
3	30	3	20	false
3	30	3	30	true

Fig. 11. Observations made for Marking ⑦. Focus has been set on the bindings that satisfy the filter predicate $\langle v_0.\text{ps\_partkey}, v_0.\text{ps\_suppkey}\rangle = \langle 2, 10\rangle$. Rows outside the focus are grayed out.

⑧		
v_2.l_partkey	v_2.l_suppkey	count
1	30	1
3	20	2
3	30	1

Fig. 12. A closed observation in Query 3. The expected row with $\langle v_2.\text{l\_partkey}, v_2.\text{l\_suppkey}\rangle = \langle 2, 10\rangle$ is missing.

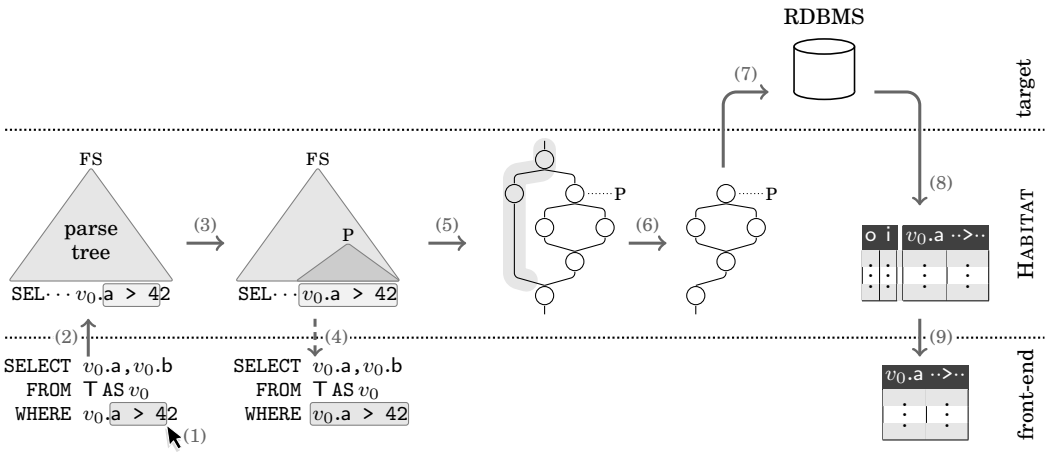


Fig. 13. From marking to observation: sketch of HABITAT's debugging procedure. Users see the front-end layer only and thus operate on the SQL language level. The RDBMS host itself is the debugging target.

3. COMPILING SQL FOR OBSERVATION

We now turn to the internals of the HABITAT debugger and to its special-purpose SQL compiler in particular. Figure 13 provides an overview of the stages and representations involved. The flow from marking to observation could be sketched as follows (numbers in parentheses refer to Figure 13):

(1) Using the debugger's front-end, the user marks an arbitrary query piece for observation. (2) The debugger creates the parse tree for the query, then (3) identifies and (4) indicates the minimal complete SQL subexpression that contains the marking, in this case the predicate P: v_0.a > 42. (5) HABITAT invokes its built-in SQL compiler to obtain an algebraic representation of the input query in which *all* of the query's subexpressions are represented explicitly. (6) The particular plan fragment associated with the marked predicate P is identified. (7) The isolated fragment (*observer query*) is compiled into SQL code and then shipped to the target RDBMS for evaluation. (8) This evaluation yields observation data in tabular form which is then (9) rendered to display the final tabulation.

3.1. Syntactic Completion

An observation originates in a marking, *i.e.* a user-defined continuous sequence or block of characters identifying an arbitrary piece of SQL text. HABITAT consults a context-free SQL grammar to extend this piece to the minimal syntactically complete subexpression that encloses the piece. Figure 14 shows a grammar that accepts the SQL dialect considered in this article. Note that this grammar has been cut down to aid the exposition—any standard grammar that reflects SQL's expression-oriented syntax will do, however.

> *Observable SQL Subexpression.*. While non-terminal FS, defining a SQL fullselect [ANSI/ISO, § 7.11], is the start symbol of this grammar, *any subexpression* that is derivable from a non-terminal in {FS, SC, P, TBL} is considered observable by the debugger.

Non-terminal SC, for example, derives any scalar SQL expression, ranging from parenthesized fullselects to individual column references or literals.

The syntactic completion of a marked query text piece originates in the parse tree leaf nodes that intersect the marking. The leaves' lowest common ancestor node in the

```
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 operators

COL ::= ID[.ID]                                    column references

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

Fig. 14. SQL fragment considered in this article. Any SQL subexpression derivable from the non-terminals FS, SC, P, or TBL may be observed by the HABITAT debugger.

set $\{\text{FS}, \text{SC}, \text{P}, \text{TBL}\}$ defines the subexpression that will be observed by HABITAT. For example, marking the piece "<= v_1.count" in the text of Query 3 (Figure 7(b)) identifies the gray parse tree leaves in Figure 15. A search for the lowest common ancestor yields non-terminal P, representing the syntactically complete predicate "v_0.ps_availqty <= v_1.count" (Marking ④ of Figure 7(b)).

3.2. Observer Queries

HABITAT's core is an original, strictly *syntax-directed* and *compositional* algebraic SQL compiler that has been designed to support subexpression observation. The compiler implements a separate custom translation rule for every SQL subexpression kind (from entire fullselect blocks to syntactic atoms, like literals or column references). A translation rule emits a fragment of algebraic plan code. The translation for a composite expression is built from the translations of the contained subexpressions. Most importantly, the emitted plan fragment for *any observable SQL subexpression may be sensibly understood and evaluated on its own*.

Observer query. As a consequence of this compilation scheme, each observable SQL subexpression e identifies an algebraic operator in the final complete plan code. This operator and its upstream sub-plan define the *observer query* for e.

Intermediate Table Algebra. Observer queries are expressed in a simple *table algebra* dialect (Table I). This dialect has been designed to reflect the query capabilities of off-the-shelf relational database engines. The simple semantics of its algebraic primitives facilitate the generation of executable SQL code (see Section 3.4), which HABITAT sub-

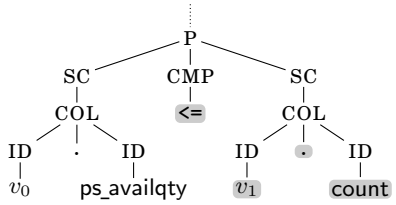


Fig. 15. Snippet of the parse tree for Query 3 (see Figure 7(b)) representing the SQL predicate "v_0.ps_availqty <= v_1.count".

Table I. Primitives of the intermediate algebraic language, with aggregation AGG $\in$ {COUNT, SUM, MAX, MIN, AVG, ANY, . .}, built-in function $f \in \{+, -, =, <, . .\}$.

Primitive	**Semantics**
$\pi_{\mathsf{a}_1:\mathsf{b}_1,..,\mathsf{a}_n:\mathsf{b}_n}$	project onto columns b_i, rename column b_i into a_i
$@_{\mathsf{a}:v}$	attach column a containing constant value v
$\circledast_{\mathsf{a}:f\langle \mathsf{b}_1,..,\mathsf{b}_n\rangle}$	apply n-ary scalar function f, attach result in column a
σ_p	eliminate rows that fail to satisfy predicate p
$\_ \bowtie_{\mathsf{a}} \_$, $\_ \times \_$	equi-join on column a, Cartesian product
$\_ \cup \_$, $\_ \setminus \_$	disjoint union, difference
δ	eliminate duplicate rows
$\mathtt{AGG}_{\mathsf{a}:\langle \mathsf{b}\rangle/\mathsf{c}}$	group rows by c, then attach aggregate of b in column a
$\#_{\mathsf{a}:\langle \mathsf{b}_1,..,\mathsf{b}_n\rangle}$	attach row number in $\mathsf{b}_1, \ldots, \mathsf{b}_n$ order in column a
$\#_{\mathsf{a}}$	attach arbitrary (yet deterministic) row identifier in column a
$\varrho_{\mathsf{a}:\langle \mathsf{b}_1,..,\mathsf{b}_n\rangle}$	attach row rank in $\mathsf{b}_1, \ldots, \mathsf{b}_n$ order in column a
$\circledcirc_t$	access rows in table t

mits to the target RDBMS to collect the required observation data. Notable primitives are the following:

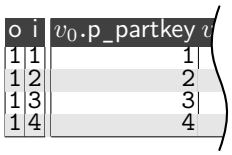
— The binary primitive $t_1 \bowtie_{\mathsf{a}} t_2$ performs the natural join of its input tables $t_{1,2}$ (their common column a is indicated for clarity).
— Unary primitive $\circledast_{\mathsf{a}:f\langle \mathsf{b}_1,..,\mathsf{b}_n\rangle}(t)$ maps the n-ary function f over its input table t, supplying the contents of columns $\mathsf{b}_1, \ldots, \mathsf{b}_n$ as parameters to f (f needs to be a built-in function, *i.e.* the algebra is not higher-order). The function result is attached in the new column a. $@_{\mathsf{a}:v}(t)$ serves as a shorthand if a nullary function (constant) v is applied.
— Unary primitive # denotes SQL:1999's OLAP amendment function ROW_NUMBER(): $\#_{\mathsf{a}:\langle \mathsf{b}_1,..,\mathsf{b}_n\rangle}$ is ROW_NUMBER() OVER (ORDER BY $\mathsf{b}_1, \ldots, \mathsf{b}_n$) AS a. Likewise, operator ϱ denotes SQL:1999's RANK(). In absence of ordering criteria, $\#_{\mathsf{a}}$ simply attaches arbitrary row identifiers in column a (SQL: RID()).
— The elimination of duplicate rows in table t is made explicit in terms of $\delta(t)$.

HABITAT's compilation strategy revolves around the tabular representation of row variable bindings (*inputs*, see Section 2) and the value of subexpressions under these bindings (*output*). We address both in turn.

Row Variable Bindings (Input). Preliminarily, define the single-row single-column table $\mathbb{1} \equiv \boxed{\mathsf{i}}\ \boxed{1}$ and define function

$$\text{LIFT}(t_1, t_2) \equiv \#_{\mathsf{i}}\big(\pi_{\mathsf{o}:\mathsf{i}, cols(t_1)-\{\mathsf{i}\}}(t_1) \times t_2\big) \ ,$$

where t_1, t_2 denote expressions of the table algebra and $cols(t)$ denotes the column set (or schema) of table t.



o	i	v_0.p_partkey v
1	1	1
1	2	2
1	3	3
1	4	4

Fig. 16. Tabular representation of values bound to row variable v_0 in scope s_0 of Query 1 (only column p_partkey shown here). (q_{v_0})

In SQL, a clause FROM e_0 AS v_0 is the only site that introduces row variables: each row of the source expression e_0 (table or subquery) creates one binding for the new row variable v_0. For a row variable v_0 bound in a top-level FROM clause (*i.e.* those located in scope s_0 in Queries 1 to 3), HABITAT's compiler emits the following fragment q_{v_0} of algebra code to represent these bindings:

$$q_{v_0} \equiv \text{LIFT}(\mathbb{1}, e_0) = \#_{\mathsf{i}}\big(\pi_{\mathsf{o:i}}(\mathbb{1}) \times e_0\big) \ .$$

To demonstrate: if e_0 is table Part of Figure 4, then q_{v_0} computes the table of Figure 16. HABITAT uses this table to encode the iterative SQL semantics: subexpressions in the scope of v_0 will be evaluated 4 times; the row $\langle 1, i, x_i, \dots \rangle$, $1 \leqslant i \leqslant 4$, indicates that v_0.p_partkey will be bound to value x_i in the ith iteration.

The introduction of further row variables follows the well-known nested-loop semantics of SQL. Assume an additional binding site FROM e_1 AS v_1 that resides in a subquery block. Subexpressions in the scope of v_1 may refer to v_0 as well as v_1 and are evaluated under *all combinations* of v_0, v_1 bindings. Again, LIFT computes the corresponding table q_{v_1} of row variable bindings:

$$q_{v_1} \equiv \text{LIFT}(q_{v_0}, e_1) = \text{LIFT}(\text{LIFT}(\mathbb{1}, e_0), e_1) \ .$$

Figure 17(a) depicts the resulting table of bindings if we apply this lifting scheme to the row variables v_0 and v_1 introduced by FROM Part AS v_0 and FROM Partsupp AS v_1 in Query 1: as required, subexpressions in scope s_1 (see Figure 3) will be iteratively evaluated under $4 \cdot 4 = 16$ bindings.

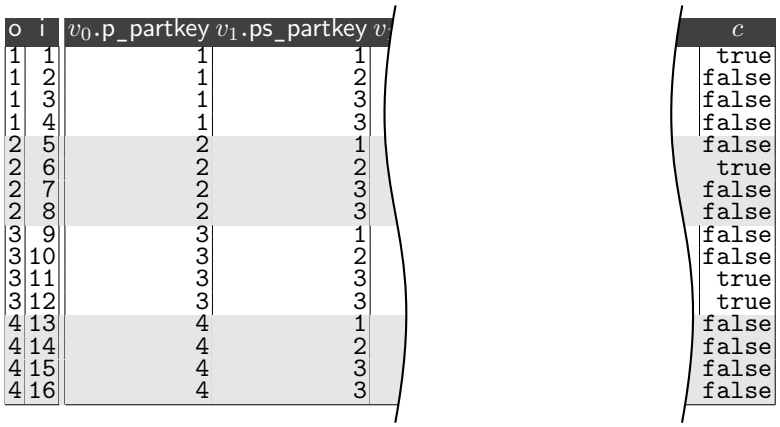
Such binding tables are invariantly wrapped by two leading columns o|i:

> *Invariant* o|i. A pair $\langle o, i \rangle$ in columns o|i indicates that row variables from the enclosing (*outer*) scope assume their oth binding while the subexpressions in the current (*inner*) scope are evaluated for the ith time.

These o|i wrappers help to relate observations made in adjacent scopes of the debugged query. Observations made in non-adjacent scopes may be transitively related in terms of the o|i wrappers of their intermediary scopes. Appendix B includes details on how a sequence of joins on the columns o|i can be used to construct a coherent display of observations that span multiple scopes.

Value of SQL Subexpressions under Row Variable Bindings (Output). Variable bindings make for the *input* half of an observation. An observer query refers to the bindings tables to complete the tabulation and adds the *output* half: for each row of bindings, the observer query evaluates its subexpression e and attaches the computed value to that row.

Figure 17 exemplifies an observation for the predicate subexpression $e \equiv$ v_0.p_partkey = v_1.ps_partkey of Query 1. The output half of Figure 17(b) is a table of 16 Boolean values resulting from the evaluation of the predicate under 16 bindings for v_0 and v_1 provided by the input half of Figure 17(a). The debugger then uses the complete tabulation, made up of both halves, to produce its output.



o	i	v_0.p_partkey	v_1.ps_partkey	v_1.ps_supplycost
1	1	1	1	10
1	2	1	2	20
1	3	1	3	30
1	4	1	3	40
2	5	2	1	50
2	6	2	2	60
2	7	2	3	70
2	8	2	3	80
3	9	3	1	90
3	10	3	2	100
3	11	3	3	110
3	12	3	3	120
4	13	4	1	130
4	14	4	2	140
4	15	4	3	150
4	16	4	3	160

(a) Input: Values bound to row variables v_0,v_1 once scope s_1 has been entered. (q_{v_1})

c
true
false
false
false
false
true
false
false
false
false
true
true
false
false
false
false

(b) Output: $4 \cdot 4 = 16$ evaluations of the subexpression.

Fig. 17. Assembly of an observation: o|i wrapper, variable bindings for v_0, v_1 in Query 1, and the value of the observed subexpression v_0.p_partkey = v_1.ps_partkey under these bindings.

Observer Query Construction Illustrated. Before we comment on details of HABITAT's SQL compiler in Section 3.3, let us gain a better intuition of compilation and observer query construction. To this end, let e denote the flawed Query 1 of Figure 2 in which *all* 10 observable SQL subexpressions have been marked by ① and ⓐ through ⓘ:[4]

```
                ⓐSELECT p_partkeyⓑ, p_nameⓒ,
                       (①SELECT ps_supplycostⓓ
e   ≡                     FROM Partsuppⓔ
                         WHERE ps_partkeyⓕ =ⓖ p_partkeyⓗ)
            FROM Partⓘ
```

(for example, ⓘ marks the reference to base table Part, ⓓ marks the reference to column ps_supplycost, ⓖ marks the equality predicate whose observation we have discussed in Figure 17, whereas ⓐ and ① mark the associated fullselect blocks; Marking ① has been discussed in Debug Session 2).

HABITAT compiles query e and obtains the algebraic plan of Figure 18. The unusual plan shape results from the strictly syntax-directed compilation strategy: *each* of the observable SQL subexpression identifies an algebraic operator in the plan—see the 10 annotations (◯) in Figure 18 that correspond to the just mentioned 10 observable subexpressions.

To obtain debugger output for a given subexpression marked ⓧ,

(1) extract its observer query root (identified by ⓧ in the plan) and its upstream plan fragment,
(2) generate SQL code for the extracted plan fragment (see Section 3.4),
(3) evaluate the generated SQL code on the target host to obtain the tabulation for ⓧ, then
(4) render the final observation display (details in Appendix B).

In the plan of Figure 18, note that all rows (or: bindings) read from the two input tables Part and Partsupp are extended with an (arbitrary) row identifier in column i_0

[4]For clarity, here we omit the boxes that otherwise delineate individual markings.

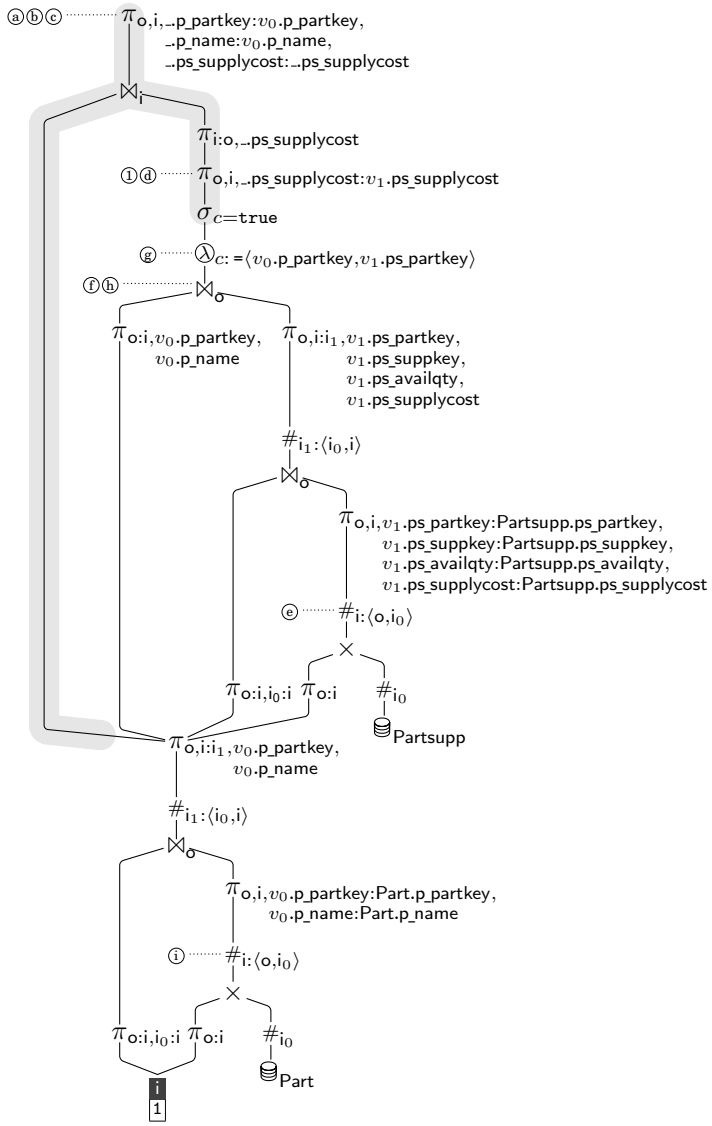


Fig. 18. ① and ⓐ – ⓘ indicate the 10 observer query roots for all observable SQL subexpressions in Query 1. To illustrate: the execution of the plan fragment marked is not required to make the observation at ⓖ.

(via $\#_{i_0}$). The immediate downstream Cartesian products ($\times$) implement the nested-loop semantics of SQL by multiplying the incoming bindings with those bindings contributed by enclosing scopes (cf. the definition of LIFT). Because the clause FROM Part appears at the query top-level, its rows are multiplied (lifted) with table $\mathbb{1}$—effectively, this indicates that there is no enclosing scope. The bindings contributed by tables Part (row variable v_0) and Partsupp (variable v_1), now properly wrapped by o|i columns, are observable at ⓘ and ⓔ, respectively.

At ⓕⓗ, a natural join $\bowtie_o$ is used to relate the bindings contributed by the top-level scope and the subquery's nested scope. From here on, we may observe the variables v_0, v_1 in the context of the subquery and can evaluate subexpressions that refer to both

variables. One such subexpression is the predicate v_0.p_partkey = v_1.ps_partkey. At ⑧, its value is computed by the application $\text{◁}_{c:=\langle\cdots\rangle}$ and attached to the binding table. We exactly obtain the tabulation of Figure 17. Finally, the top-level fullselect block ⓐ and the subexpressions of its SELECT clause ⓑⓒ are observable at the plan root.

To support observation, HABITAT's translation strategy deviates from techniques employed in off-the-shelf SQL compilers. Consider:

— At operator ⑧, the Boolean value of the predicate v_0.p_partkey = v_1.ps_partkey is made available for observation. Actual row filtering only happens later in the downstream plan (via $\sigma_{c=\texttt{true}}$). An off-the-shelf SQL compiler will merge predicate evaluation and filtering.
— A typical SQL compiler will translate query e into a single equi-join between base tables Part and Partsupp. Neither the two input legs nor the output of this join would represent the result of the correlated subquery block. However, with HABITAT, the block's result becomes observable at operator ①.

3.3. HABITAT's SQL Compiler

HABITAT comes with its own SQL compiler that has been designed to support observational debugging for a given input query. Each subexpression of this query is translated into its associated observer query—taken together, these observer queries form an algebraic plan for the entire input query (cf. the plan of Figure 18).

The SQL compiler is specified in terms of inference rules (Figure 19) which collectively define function $\Rrightarrow$. Read $(cs, q), gs \vdash e \Rrightarrow (cs', q')$ as (gs addresses grouping, see below; assume $gs = \varnothing$ for now):

— given algebraic query q that computes a table whose column set cs hold the bindings of in-scope row variables, [*input*]
— construct an algebraic query q' that computes a table whose column set cs' hold the result of the evaluations of SQL subexpression e under the provided row variable bindings. [*output*]

Compilation starts out with the input query's top-level fullselect block expression (no row variable bindings exist): HABITAT invokes

$$(\varnothing, \mathbb{1}), \varnothing \vdash \texttt{SELECT} \cdots \texttt{FROM} \cdots \Rrightarrow (cs', q') \ .$$

The translation of each supported SQL syntactic language construct, from fullselect block to scalar literal, is defined in terms of the construct's custom inference rule. For a composite expression e, the rule's antecedent recursively invokes function $\Rrightarrow$ to compile the subexpressions of e in a compositional, bottom-up fashion. Variant $\mapsto$ (Figure 20) is invoked to compile scalar subexpressions. Auxiliary function $\Rightarrow$ (Figure 21) compiles single-row or scalar fullselects that appear in place of a scalar expression [ANSI/ISO, § 7.14]. The rule's consequent constructs a fragment of algebraic plan code that evaluates to a table wrapped in leading o|i columns. The final plan q' will incorporate the plan fragments for all observable subexpressions of the input query. Appendix A illustrates how these compilation rules cooperate to infer the algebraic plan for SQL Query 1 (of Figure 1 and under discussion in Section 3.2). The resulting plan is that of Figure 18.

Rule ($_5$) of Figure 19 is invoked for each base table reference in a FROM clause. The consequent realizes row variable lifting as introduced above, seeding wrapper column i with physical row identifiers (using $\#_{\mathsf{i}_0}$) such that observations encounter the base table's rows in a deterministic (yet arbitrary) order.

FS-SFWGHO

$$\frac{\begin{array}{lll}\textsc{from} & (cs,q),gs \vdash \texttt{FROM}\, e_{2,1}\texttt{,}e_{2,2}\texttt{,}.. \Mapsto (cs_2,q_2) & \\ & cs_2' \equiv cs_2 + cs \qquad q_2' \equiv q_2 \bowtie_{\mathsf{o}} \pi_{\mathsf{o:i},cs}(q) & \\ \hline \textsc{where} & (cs_2',q_2'),gs \vdash e_3 \mapsto (c_3,q_3) & q_3' \equiv q_2' \quad \cancel{\textsc{where}} \\ & q_3' \equiv \sigma_{c_3=\texttt{true}}(q_3) & \\ \hline \textsc{group by} & g \equiv \mathit{fresh}() \quad cs_4 \equiv \{t_1\texttt{.}c_1, t_2\texttt{.}c_2, ..\} - cs \quad q_4 \equiv \varrho_{g:\langle \mathsf{o},cs_4\rangle}(q_3') & \\ & q_4' \equiv \delta(\pi_{\mathsf{o},\mathsf{i}:g,cs_4}(q_4)) \bowtie_{\mathsf{o}} \pi_{\mathsf{o:i},cs}(q) & cs_4' \equiv cs_2' \quad \cancel{\textsc{group by}} \\ & cs_4' \equiv cs_4 + cs \qquad q_4'' \equiv \pi_{\mathsf{o}:g,\mathsf{i},cs_2}(q_4) \qquad gs' \equiv \{(cs_2, q_4'')\} & \\ \hline \textsc{having} & (cs_4',q_4'),gs' \vdash e_5 \mapsto (c_5,q_5) & q_5' \equiv q_3' \quad gs'' \equiv \varnothing \quad \cancel{\textsc{having}} \\ & q_5' \equiv \sigma_{c_5=\texttt{true}}(q_5) \qquad gs'' \equiv \{(cs_2, q_4'' \bowtie_{\mathsf{o}} \pi_{\mathsf{o:i}}(q_5'))\} & \\ \hline \textsc{select} & (cs_4',q_5'),gs'' \vdash \texttt{SELECT}\, e_{1,1}\,\texttt{AS}\, a_1\texttt{,}e_{1,2}\,\texttt{AS}\, a_2\texttt{,}.. \Mapsto (cs_1,q_1) & \\ \hline \textsc{order by} & (cs_1,q_1),gs \vdash \texttt{ORDER BY}\, e_{6,1}\texttt{,}e_{6,2}\texttt{,}.. \Mapsto (cs_6,q_6) & cs_6 \equiv cs_1 \quad q_6 \equiv q_1 \quad \cancel{\textsc{order by}} \end{array}}{\begin{array}{c}\texttt{SELECT}\ e_{1,1}\,\texttt{AS}\, a_1\texttt{,}e_{1,2}\,\texttt{AS}\, a_2\texttt{,}.. \\ \texttt{FROM}\ e_{2,1}\texttt{,}e_{2,2}\texttt{,}.. \\ (cs,q),gs \vdash \left[\begin{array}{l}\texttt{WHERE}\ e_3\,]\\ [\,\texttt{GROUP BY}\, t_1\texttt{.}c_1\texttt{,}t_2\texttt{.}c_2\texttt{,}..\end{array}\right. \qquad \Mapsto (cs_6, \pi_{\mathsf{o},\mathsf{i},cs_6}(q_6)) \\ [\,\texttt{HAVING}\ e_5\,]\,] \\ [\,\texttt{ORDER BY}\, e_{6,1}\texttt{,}e_{6,2}\texttt{,}..\,]\end{array}}(1)$$

FS-SELECT

$$\frac{\begin{array}{c}cs_0 \equiv \varnothing \quad ps_0 \equiv \varnothing \\ (cs,q_{i-1}),gs \vdash e_i \Mapsto (c_i,q_i) \\ cs_i \equiv cs_{i-1} + \{\_\texttt{.}a_i\} \\ ps_i \equiv ps_{i-1} + \{\_\texttt{.}a_i\texttt{:}c_i\}\end{array}\Big|_{i=1,..,n}}{\begin{array}{c}(cs,q_0),gs \vdash \texttt{SELECT}\, e_1\,\texttt{AS}\, a_1\texttt{,}..\texttt{,}e_n\,\texttt{AS}\, a_n \\ \Mapsto \left(cs_n, \pi_{\mathsf{o},\mathsf{i},ps_n}(q_n)\right)\end{array}}(2)$$

FS-FROM

$$\frac{\begin{array}{c}cs_0' \equiv \varnothing \quad q_0' \equiv \pi_{\mathsf{o:i},\mathsf{i}_0\mathsf{:i}}(q) \\ (cs,q),gs \vdash e_i \Mapsto (cs_i,q_i) \\ cs_i' \equiv cs_{i-1}' + cs_i \\ q_i' \equiv \pi_{\mathsf{o},\mathsf{i}_i,cs_i'}(\#_{\mathsf{i}_i:\langle \mathsf{i}_{i-1},\mathsf{i}\rangle}(q_{i-1}' \bowtie_{\mathsf{o}} q_i))\end{array}\Big|_{i=1,..,n}}{\begin{array}{c}(cs,q),gs \vdash \texttt{FROM}\, e_1\texttt{,}..\texttt{,}e_n \\ \Mapsto (cs_n', \pi_{\mathsf{o},\mathsf{i}:\mathsf{i}_n,cs_n'}(q_n'))\end{array}}(3)$$

FS-ALIAS

$$\frac{\begin{array}{c}cs_0 \equiv \varnothing \qquad ps_0 \equiv \varnothing \\ (cs,q),gs \vdash e_1 \Mapsto (\{v\texttt{.}c_1,..,v\texttt{.}c_n\},q_1) \\ cs_i \equiv cs_{i-1} + \{t\texttt{.}c_i\} \\ ps_i \equiv ps_{i-1} + \{t\texttt{.}c_i\texttt{:}v\texttt{.}c_i\}\end{array}\Big|_{i=1,..,n}}{(cs,q),gs \vdash e_1\,\texttt{AS}\, t \Mapsto \left(cs_n, \pi_{\mathsf{o},\mathsf{i},ps_n}(q_1)\right)}(4)$$

FS-TABLE

$$\frac{q_0' \equiv \#_{\mathsf{i}:\langle \mathsf{o},\mathsf{i}_0\rangle}(\pi_{\mathsf{o:i}}(q) \times \#_{\mathsf{i}_0}(\circledS_t))}{\begin{array}{c}(cs,q),gs \vdash t(c_1\texttt{,}..\texttt{,}c_n) \\ \Mapsto \left(\{t\texttt{.}c_1,..,t\texttt{.}c_n\}, q_0'\right)\end{array}}(5)$$

FS-ORDER

$$\frac{(cs,q_{i-1}),gs \vdash e_i \Mapsto (c_i,q_i)\,\big|_{i=1,..,n}}{\begin{array}{l}(cs,q_0),gs \vdash \texttt{ORDER BY}\, e_1\texttt{,}..\texttt{,}e_n \\ \Mapsto (\{c_1,..,c_n\} \cup cs, \pi_{\mathsf{o},\mathsf{i},c_1,..,c_n,cs}(q_n)))\end{array}}(6)$$

Fig. 19. Definition of $\Mapsto$: compositional compilation of SQL fullselects. Calls on $\Mapsto$ and $\mapsto$. Entry point is Rule (1). The WHERE side of the antecedent is to be used if the input query lacks a WHERE clause (likewise for GROUP BY, HAVING, and ORDER BY). $\mathit{fresh}()$ returns a new, yet arbitrary, column name on each invocation.

SC-VAL

$$\frac{c \equiv \mathit{fresh}()}{(cs, q), gs \vdash \mathit{val} \mapsto (c, @_{c:\mathit{val}}(q))} (7)$$

SC-COL

$$\frac{}{(cs, q), gs \vdash t\,.\,c \mapsto (t\,.\,c, q)} (8)$$

SC-APPLY

$$\frac{(cs, q_{i-1}), gs \vdash e_i \Mapsto (c_i, q_i)\,\big|_{i=1,..,n} \qquad c \equiv \mathit{fresh}() \qquad \textbf{built-in } n\textbf{-ary scalar SQL function/operator } f}{(cs, q_0), gs \vdash f(e_1, .., e_n) \mapsto \left(c, \textcircled{\lambda}_{c:f\langle c_1,..,c_n\rangle}(q_n)\right)} (9)$$

SC-CASE

$$\frac{\begin{array}{c}(cs, q), gs \vdash e_1 \mapsto (c_1, q_1) \qquad c \equiv \mathit{fresh}() \\ (cs, \sigma_{c_1=\texttt{true}}(q_1)), gs \vdash e_2 \Mapsto (c_2, q_2) \\ (cs, \sigma_{c_1=\texttt{false}}(q_1)), gs \vdash e_3 \Mapsto (c_3, q_3)\end{array}}{\begin{array}{l}(cs, q), gs \vdash \texttt{CASE WHEN}\, e_1\, \texttt{THEN}\, e_2\, \texttt{ELSE}\, e_3\, \texttt{END} \\ \mapsto \left(c, \pi_{\mathit{cols}(q),c}(q \bowtie_{\mathsf{i}} (\pi_{\mathsf{i},c:c_2}(q_2) \cup \pi_{\mathsf{i},c:c_3}(q_3)))\right)\end{array}} (10)$$

SC-COUNT

$$\frac{c \equiv \mathit{fresh}() \qquad q_0 \equiv \texttt{COUNT}_{c:\langle\rangle/\mathsf{o}}(q_g)}{\begin{array}{l}(cs, q), \{(cs_g, q_g)\} \vdash \texttt{COUNT(*)} \\ \mapsto (c, q \bowtie_{\mathsf{i}} \pi_{\mathsf{i:o},c}(q_0 \cup (@_{c:0}(\pi_{\mathsf{o:i}}(q) \setminus \pi_{\mathsf{o}}(q_0)))))\end{array}} (11)$$

SC-AGG

$$\frac{\begin{array}{c}\texttt{AGG} \in \{\texttt{COUNT}, \texttt{SUM}, \texttt{MAX}, \texttt{MIN}, \texttt{AVG}\} \\ (cs_g, q_g), \varnothing \vdash e_1 \Mapsto (c_1, q_1) \\ c \equiv \mathit{fresh}() \qquad q_0 \equiv \texttt{AGG}_{c:\langle c_1\rangle/\mathsf{o}}(q_1)\end{array}}{\begin{array}{l}(cs, q), \{(cs_g, q_g)\} \vdash \texttt{AGG}(e_1) \\ \mapsto (c, q \bowtie_{\mathsf{i}} \pi_{\mathsf{i:o},c}(q_0 \cup (@_{c:\texttt{null}}(\pi_{\mathsf{o:i}}(q) \setminus \pi_{\mathsf{o}}(q_0)))))\end{array}} (12)$$

SC-IN

$$\frac{\begin{array}{c}(cs, q), gs \vdash e_1 \Mapsto (c_1, q_1) \qquad (cs, q), gs \vdash e_2 \Mapsto (\{c_2\}, q_2) \\ c \equiv \mathit{fresh}() \qquad q_0 \equiv \texttt{ANY}_{c:\langle c\rangle/\mathsf{o}}(\textcircled{\lambda}_{c:=\langle c_1,c_2\rangle}(\pi_{\mathsf{o:i},c_1}(q_1) \bowtie_{\mathsf{o}} q_2))\end{array}}{\begin{array}{l}(cs, q), gs \vdash e_1\, \texttt{IN}\, e_2 \\ \mapsto (c, q \bowtie_{\mathsf{i}} \pi_{\mathsf{i:o},c}(q_0 \cup @_{c:\texttt{false}}(\pi_{\mathsf{o:i}}(q) \setminus \pi_{\mathsf{o}}(q_0))))\end{array}} (13)$$

SC-EXISTS

$$\frac{\begin{array}{c}(cs, q), gs \vdash e \Mapsto (cs_1, q_1) \\ c \equiv \mathit{fresh}() \qquad q_0 \equiv \delta(\pi_{\mathsf{o}}(q_1))\end{array}}{\begin{array}{l}(cs, q), gs \vdash \texttt{EXISTS}(e_1) \\ \mapsto (c, q \bowtie_{\mathsf{i}} \pi_{\mathsf{i:o},c}(@_{c:\texttt{true}}(q_0) \cup @_{c:\texttt{false}}(\pi_{\mathsf{o:i}}(q) \setminus q_0)))\end{array}} (14)$$

Fig. 20. Definition of $\mapsto$. Compiles scalar SQL expressions (includes operator/function application, aggregation, and predicates). Note: Rules (11) and (12) compile aggregates under the bindings (cs_g, q_g) provided by an associated `GROUP BY` clause.

SC-FS

$$\frac{\mathit{fullselect}(e) \qquad (cs, q), gs \vdash e \Mapsto (\{c\}, q_0)}{(cs, q), gs \vdash (e) \Mapsto (c, q \bowtie_{\mathsf{i}} \pi_{\mathsf{i:o},c}(q_0))} (15)$$

SC-SC

$$\frac{\neg \mathit{fullselect}(e) \qquad (cs, q), gs \vdash e \mapsto (c, q_0)}{(cs, q), gs \vdash e \Mapsto (c, q_0)} (16)$$

Fig. 21. Definition of auxiliary function $\Mapsto$. Enables the use of scalar and row subqueries in place of SQL scalar expressions. $\mathit{fullselect}(e)$ is true iff expression e constitutes a SQL fullselect [ANSI/ISO, § 7.11].

Rule (2) translates a SELECT clause. The clause's subexpressions $e_1, \ldots, e_n$ are recursively translated in reference to query q_0 which provides a binding table for all free variables used by the subexpressions. Besides the pervasive o|i wrapper, the resulting table has n columns (whose names are collected in the set ps_n) containing the values of $e_1, \ldots, e_n$ under the bindings provided by q_0.

Rule (9) embodies the application of an arbitrary n-ary scalar SQL function f (built-in or user-defined). When the observer query is executed, the target database host will call f with the required list of parameters and attach the result in column c.

Finally, grouping affects variable binding: after GROUP BY, row variables assume one binding *per group* formed. Rule (1) makes these new bindings available in table gs'': the rows of a group share a common value o in column o, which identifies such a *set of rows* as the oth binding—see the Invariant o|i above. Table gs'' is passed to $\Rrightarrow$ when the function is invoked to compile aggregates in the SELECT clause (see Rules (11) and (12) in Figure 20).

3.4. Generating SQL Target Code

Observer queries are to be executed on the target database host itself: HABITAT thus translates an observer query's algebraic plan fragment (see Section 3.2) into regular SQL:1999 statements. We note that the plan fragment may be subjected to regular query simplification and optimization before code generation starts. A simple, yet usable, SQL code generator is obtained as follows:

(1) Perform a postorder walk of the plan fragment from the base tables up to the observer query root, applying a straightforward primitive-by-primitive translation to SQL.
(2) Assemble the resulting SQL fragments via SQL's common table expression construct WITH [ANSI/ISO, § 7.12] to form a single executable statement.

A variant of this procedure that performs basic algebraic simplifications (see Section 4) and then greedily collects multiple primitives to form a single SQL statement can generate compact SQL code [Grust et al. 2007a].

Figure 22 lists the SQL code generated for the observer query of Marking ③ (Query 2, Figure 7(a)). A closer look elucidates HABITAT's strategy. The function to tabulate is

$$f_{③}(v_0) = v_0.\mathsf{ps\_availqty} \texttt{ <= (SELECT COUNT(*) FROM Lineitem AS } v_1 \texttt{ WHERE } v_0.\mathsf{ps\_partkey} \texttt{ = } v_1.\mathsf{l\_partkey} \texttt{ AND } v_0.\mathsf{ps\_suppkey} \texttt{ = } v_1.\mathsf{l\_suppkey)} \quad .$$

Function $f_{③}(v_0)$ needs to be evaluated for all bindings of parameter v_0. A consequence of compositional compilation, the values of the subexpression v_0.ps_availqty and the nested fullselect block in $(\cdots)$ are computed separately in tables t0 and t3 (with t1, t2), respectively. The final query block in lines 27 to 35 of Figure 22 ties both intermediate results together to evaluate the comparison (<=) of both subexpressions. The join on column i (line 34) guarantees that intermediate results corresponding to the same binding of v_0 are compared. In effect, Query 2 has been turned "inside out" (cf. Section 1): the comparison subexpression (<=) now appears at the observer query top-level and thus becomes observable.

A few further notes on fragments of the generated SQL code:

t0: The bindings of v_0 are supplied by table t0 in which the rows of table Partsupp have been projected onto the columns (ps_partkey, ps_suppkey, ps_availqty) relevant for observation. Column i attaches an (arbitrary) row identifier to identify the

```
1 WITH
2 t0 (ps_partkey,ps_suppkey,ps_availqty,o,i)
3   AS (SELECT Partsupp.ps_partkey,
4              Partsupp.ps_suppkey,
5              Partsupp.ps_availqty,
6              1 as o,
7              RID(Partsupp) AS i
8         FROM Partsupp),
9
10 t1 (c,o)
11   AS (SELECT COUNT(*) AS c, t0.i AS o
12         FROM t0, Lineitem
13        WHERE t0.ps_partkey = Lineitem.l_partkey
14          AND t0.ps_suppkey = Lineitem.l_suppkey
15        GROUP BY t0.i),
16
17 t2 (o)
18   AS (SELECT t0.i AS o FROM t0
19         EXCEPT ALL
20       SELECT t1.o FROM t1),
21
22 t3 (c,o)
23   AS (SELECT t1.c, t1.o FROM t1
24         UNION ALL
25       SELECT 0 AS c, t2.o FROM t2)
26
27 SELECT t0.ps_partkey,
28        t0.ps_suppkey,
29        t0.ps_availqty,
30        CASE WHEN t0.ps_availqty <= t3.c
31             THEN 'true' ELSE 'false' END AS "· <= ·",
32        t0.o, t0.i
33   FROM t0, t3
34  WHERE t0.i = t3.o
35  ORDER BY o, i;
```

Fig. 22. SQL target code generated for the observer query of Marking ③ (Figure 7(a)). The evaluation of the "out of supply" (· <= ·) predicate now happens at the query top-level (see line 30).

Table II. A quantification of the performance impact of HABITAT's closure-based SQL compilation strategy. Query timings (wall-clock execution time, in seconds).

Query	**SF1 (1 GB)**			**SF10 (10 GB)**		
	# rows	original ⏱(s)	HABITAT ⏱(s)	# rows	original ⏱(s)	HABITAT ⏱(s)
Q1	4	18.20	18.92	4	184.29	195.34
Q2	100	0.04	0.02	100	0.48	0.21
Q3	10	5.43	5.54	10	98.69	101.65
Q4	5	4.43	4.65	5	99.29	101.27
Q5	5	1.08	1.20	5	23.10	26.10
Q6	1	0.71	0.66	1	13.59	13.04
Q7	4	0.48	0.51	4	7.25	7.39
Q8	2	0.29	0.29	2	5.09	5.08
Q9	175	2.86	2.87	175	76.66	78.19
Q10	10	1.07	1.13	10	24.86	26.60
Q11	1,048	0.19	0.37	8,685	2.28	4.14
Q12	2	0.33	0.47	2	5.03	9.01
Q13	42	3.25	3.57	46	45.41	49.49
Q14	1	0.19	0.20	1	2.26	2.28

individual bindings. The value of subexpression v_0.ps_availqty can be directly read off `t0`.

`t1`: Since v_0 is free in the parenthesized nested fullselect block, a Cartesian product with `t0` is performed (line 12) to supply the bindings required for evaluation. Grouping by column `i` evaluates the `COUNT(*)` aggregate for each binding separately.

`t2`/`t3`: Table `t1` will lack entries for those bindings that fail to satisfy the join predicate in lines 13 and 14. To properly capture the semantics of `COUNT(*)` in $f_{③}(v_0)$, table `t2` identifies these bindings. The definition of table `t3` then uses `t2` to add the expected aggregate result of `0` (line 25). [See Rule (11) in Figure 20.]

4. INTERACTIVE DEBUGGING

Observational debugging is an interactive process. A user's first guess of which piece will provide insight into the behavior of a buggy query is sufficient to start a debug session. Nonetheless, for more complex queries or subtle bugs, a typical session will repeatedly zoom into suspect query pieces and then out again. HABITAT's non-standard SQL compiler has *not* been primarily designed for performance. The explicit support for expression observation requires the construction of algebraic plans that evaluate expressions *and* produce the associated bindings of free row variables. Because it is to be expected that this overhead affects query execution performance, in the following we try to quantify HABITAT's actual runtime impact.

The price of compositionality. We translated the queries *Q1* to *Q14* of the TPC-H benchmark [Transaction Processing Performance Council] using HABITAT's compiler (function ⇛, Section 3). The resulting algebraic plans thus explicitly compute the value of *every* subexpression as well as the associated free variable bindings (cf. Figure 18). In the experiment, we thus chose to observe the top-level fullselect block, *i.e.* observer query root and plan root coincide. (We discuss the observation of non-top-level expressions below.) Under this setup, the experiment assesses how well the system can cope with the overhead of HABITAT's unusual explicit compositional mode of compilation: in comparison with a classical SQL compiler, plan size is larger and plan shape is tall: join operations do not cluster in a join tree but appear distributed all over the plan (recall Figure 18).

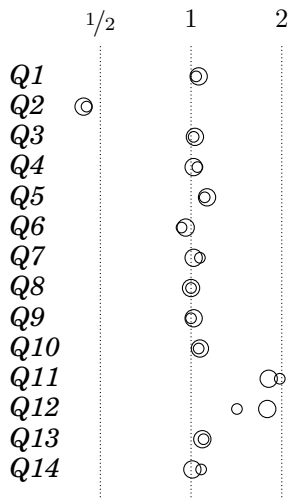


Fig. 23. Relative execution times of debug-able queries (○ ≙ SF1, ◯ ≙ SF10).

Experimental setup. All plans were then fed into a greedy SQL code generator that follows the principles described in [Grust et al. 2007a]. We executed the generated SQL statements against two vanilla TPC-H database instances of scale factor SF $\in \{1, 10\}$. Both instances were hosted by an IBM DB2 V9.7 database system, running on a contemporary Linux 2.6 host equipped with a 3.2 GHz Intel Xeon™ CPU addressing 8 GB of primary and SCSI-based secondary disk memory. The wall-clock execution times of the original TPC-H queries, executed as-is as specified by the benchmark, provide the performance baseline for the comparison of Table II. All timings were measured under the index configuration suggested by IBM DB2's index advisor `db2advis` once it had been exposed to the original TPC-H workload. No HABITAT-specific adaptations were made. Queries were executed with warm caches, modeling the state of the system during an ongoing debug session: we performed 11 runs for each query and report the average timings of the last 10 runs.

Discussion. Figure 23 reports on HABITAT's relative performance impact in this scenario. A mark ○ on the line labeled "2" indicates that the use of HABITAT's SQL compiler leads to doubled query execution times, for example. We generally saw only modest increases in query execution time. HABITAT's low overhead is primarily due to the fact that already very basic plan simplifications built into the SQL code generator succeeded in reducing plan size and complexity. One such simplification implements the removal of unused columns. To illustrate, consider the operator pair $\circledast_{c:\,=\langle v_0.\mathsf{p\_partkey}, v_1.\mathsf{ps\_partkey}\rangle} - \bowtie_{\mathsf{o}}$ at ⓖ in the plan of Figure 18. The role of the join $\bowtie_{\mathsf{o}}$ is to perform the required multiplication of bindings for row variables v_0 and v_1 in Query 1. The effective size of the join result would thus be that of the Cartesian product of tables Part and Partsupp. Atop this join, compositional compilation has placed operator $\circledast_{c:\,=\langle\cdots\rangle}$ to attach column c whose contents permit the observation of the equality predicate $v_0.\mathsf{p\_partkey} = v_1.\mathsf{ps\_partkey}$ (Rule (9) of Figure 20). In the current experiment, the predicate is not under observation, however, and beyond the subsequent selection $\sigma_{c=\mathtt{true}}$, column c is not referenced in the downstream plan. Code generation thus collapsed the three operators into the single θ-join $\bowtie_{\mathsf{o} \wedge v_0.\mathsf{p\_partkey} = v_1.\mathsf{ps\_partkey}}$. This predicate pushdown effectively saved the query processor from evaluating the costly multiplication of bindings just mentioned. The resulting θ-join is close to what an off-the-shelf SQL compiler would have planned for this query.

The presence of `ROW_NUMBER() OVER (ORDER BY ...)` in the generated SQL code (see Section 3) negatively impacts query execution time. We saw this for queries *Q10* and *Q11* that clearly suffer from the sorting overhead that comes with this ordered variant of row numbering. What, then, explains HABITAT's surprising performance advantage for TPC-H query *Q2*? The original formulation of this TPC-H query, featuring a correlated subquery, led IBM DB2 to plan for a nested-loop join. HABITAT's use of `ROW_NUMBER()` in the observer query led the system to materialize the outer leg of this join, which ultimately enabled to plan for an efficient hash join instead. This particular plan variant appears not to be included in IBM DB2's usual optimizer search space. Despite this artifact of IBM DB2's approach to query optimization, as expected, HABITAT-compiled plans pay a price for compositionality. The price appears to be sufficiently low, however, and does not preclude interactive debugging.

Non-Top-Level Observations. In a typical debugging scenario, a user will concentrate on one (or few) subexpressions at a time. Execution cost will incur for the relevant plan fragment—the observer query—only. Yet, the size of the observation can exceed the result size of the query that contains the piece.

To see this, reconsider Query 1 of Section 1. Below, we have annotated its query pieces with the *number of rows* that an observation will return (let $p = |\mathsf{Part}|$, $ps = |\mathsf{Partsupp}|$, and let $0 \leqslant sel \leqslant 1$ denote the selectivity of the predicate $v_0\,.\,\mathsf{p\_partkey} = v_1\,.\,\mathsf{ps\_partkey}$):

$$
\begin{array}{l}
\texttt{SELECT}\ v_0\,.\,\mathsf{p\_partkey}^{p}, v_0\,.\,\mathsf{p\_name}^{p}, \\
\quad\ \ (\texttt{SELECT}\ v_1\,.\,\mathsf{ps\_supplycost}^{p\cdot ps\cdot sel} \\
\quad\quad\ \ \texttt{FROM}\ \mathsf{Partsupp}^{p\cdot ps}\ \texttt{AS}\ v_1 \\
\quad\quad\ \ \texttt{WHERE}\ v_0\,.\,\mathsf{p\_partkey}^{p\cdot ps} =^{p\cdot ps} v_1\,.\,\mathsf{ps\_partkey}^{p\cdot ps}) \\
\quad \texttt{FROM}\ \mathsf{Part}^{p}\ \texttt{AS}\ v_0\ .
\end{array}
$$

Note that, while the overall query result only has p rows, the observation of table Partsupp yields $p \cdot ps$ (not ps) rows. Analogous remarks apply to all subexpressions in the scope of the nested subquery. While query optimizers try to avoid the generation of such large intermediate results, proper query observation needs to preserve this nested-loop semantics of SQL subqueries. If we consider Marking ⓖ again and thus mark the equality predicate $v_0\,.\,\mathsf{p\_partkey} = v_1\,.\,\mathsf{ps\_partkey}$ for observation, we effectively make the operator pair $\lambda_{c:\,=\langle v_0.\mathsf{p\_partkey}, v_1.\mathsf{ps\_partkey}\rangle} - \bowtie_{\circ}$ the root of the observer query (see ⓖ and its upstream plan fragment in Figure 18). This time, column c holds the object of the (Boolean) observation and, since it defines the plan output, cannot be optimized away: the database back-end will compute a tabulation of $p \cdot ps$ rows.

The placement of markings, thus, may inhibit operator fusion which a relational query optimizer would otherwise exploit to keep the size of intermediate results in check. In the worst case, markings may conjure a Cartesian product (*e.g.* when multiple tables or subqueries of a common `FROM` clause are to be observed) where the original query could employ a selective join.

4.1. Coping with Large Observations

Debugging a query against original instance data can be vital to hunt down specific data-dependent bugs (see Debug Session 2), but may yield large observations as we have just mentioned. HABITAT does not impose an unduly performance overhead in such scenarios—nevertheless, detecting the cause of a bug in a sea of rows can be hard.

The target database host itself can help to avoid the generation of such seas of rows in the first place. Recall that markings are compiled into observer queries and then translated into regular SQL code. Shipping this SQL code to the target RDBMS's *ex-*

```
1 WITH
2 t0 (p_partkey,p_name,i)
3   AS (SELECT Part.p_partkey,
4              Part.p_name,
5              1 AS o,
6              RID(Part) AS i
7         FROM Part)
8
9 SELECT t0.p_partkey,
10       Partsupp.ps_partkey,
11       CASE WHEN t0.p_partkey = Partsupp.ps_partkey
12            THEN 'true' ELSE 'false' END AS "·=·",
13       t0.i AS o,
14       RID(Partsupp) AS i
15  FROM t0, Partsupp -- ⊗
16 ORDER BY o, i;
```

Fig. 24. Generated SQL code for the observer query of Marking ⓖ. RDBMS explain facilities predict the Cartesian product at ⊗—and thus the overall observation—to yield $p \cdot ps$ rows ($p = |\mathsf{Part}|$, $ps = |\mathsf{Partsupp}|$).

plain plan facility[5] provides an estimate of the size of an observation before the observer query code is actually executed. Figure 24 shows the SQL code generated for the observer query of Marking ⓖ discussed above. The explain plan facilities of all mentioned RDBMSs correctly predicted the exact tabulation size of $p \cdot ps$ rows. Note, though, that already a rough estimate of the expected row count is useful—such an estimate suffices to tell observations of size p from those of size $p \cdot ps$, for example. If a size threshold is exceeded, HABITAT warns the user and prompts for action, *e.g.* the application of *focus filters* or the *debugging with small example data instances*.

Focus Filters. HABITAT's use of tabulations suggests to let users formulate *filter predicates* against these tables. A filter predicate defines a subset of interesting observations that should be in focus from now on. In Figure 11, such a focus was defined using the predicate

$$v_0.\mathsf{ps\_partkey} = 2 \wedge v_0.\mathsf{ps\_suppkey} = 10$$

while searching for a missing row carrying these two column values. Once a focus has been set, Invariant o|i of Section 3 helps the debugger to highlight related rows in other observation displays defined for the same query. Filters can be

(1) implemented by a visual debugger front-end, or
(2) added to the algebraic observer query to let the target database host only compute the row set that is in focus.

The preferable option (2) saves the debugger client from receiving out-of-focus data at all.

Since HABITAT's SQL compositional compilation strategy considers subexpressions independently and guarantees that the bindings for all in-scope variables are available, users may formulate focus filters in terms of arbitrary SQL predicate syntax. More precisely, a focus filter is acceptable if it is derviable from non-terminal P (Boolean subexpressions) of the grammar in Figure 14 and does refer to in-scope row variables only. In the case of Query 1 of Figure 2, for example, a debugger shell may

[5] *Explain plan* facilities and client-accessible row count estimates are available in all major RDBMS implementations, including IBM DB2, MS SQL Server, MySQL, Oracle, and PostgreSQL.

allow users to define the focus predicate

```
COUNT(①) > 1
```

that restricts the tabulation to only include those evaluations of Marking ①, which indeed violate the single-row cardinality constraint (*e.g.* the problematic third row with p_partkey $=$ 3 in Figure 5). The algebraic code that evaluates the focus filter predicate is attached on top the observer query root operator before SQL code generation is invoked.

Debugging with Small Example Data. Alternatively, we can perceive an algebraic observer query as a specific *dataflow program*. It is likely that we are able to spot bugs buried in this program if we make sure that all its primitives are indeed "exercised". Sets of input rows exercise an algebraic primitive [Chopra et al. 2009] if the primitive's key semantics all come into effect during program evaluation (selection σ_p both rejects and accepts at least one row in the set, $\_ \cup \_$ produces duplicate as well as unique rows, *etc.*).

To this end, HABITAT could adapt an algorithm that has originally been designed to generate instructive example data for Pig Latin programs [Chopra et al. 2009]. This algorithm performs a multi-pass traversal of the algebraic program to ensure that the resulting input row sets are exercising as well as concise, so that users would not be required to observe equivalent expression behavior multiple times. At the plan leaves (*i.e.* LOAD $t(\cdots)$ in Pig Latin or ⛁_t in the case of HABITAT), the data generator aims to draw rows from actual instance data in an attempt to display observations that contain realistic example rows. These compact example data would then be used in place of the target's database instance—other than that, HABITAT operates as before.

Nonetheless, the effective use of any debugger involves a learning phase in which users develop an understanding of which markings yield tractable and insightful observations [Katz and Anderson 1987]. The same is true for HABITAT.

5. RELATED WORK

HABITAT's *language-level debugging* approach marks a deviation from *plan-level debugging* that exposes the query engine's *internal* plan representation [Bati et al. 2007; Bruno et al. 2009]. Laying plans bare facilitates performance analysis but appears to be of limited use in fixing logical flaws:

(1) With today's capable and complex query optimizers, the shape of a plan is largely disconnected from its surface query. Views may have been unfolded, subqueries unnested and turned into joins, tables removed, joins reordered, predicates eliminated or introduced, *etc.* [Liang 2009].
(2) Even minimal surface query edits may very well lead to disruptive plan changes [Haritsa and Reddy 2005], thus impeding interactive debug sessions in which users gradually introduce query fixes.
(3) Finally, a (physical) plan is not the appropriate spot to address a *logical* flaw: recall that there has been no issue with the plan for Query 1 per se, rather its data-dependent bug only occurred at query runtime.

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. The invocation of a SQL query from within a script, however, makes for a *monolithic opaque action* that cannot be traced or inspected [Microsoft Corporation; Embarcadero Technologies]. Instead, HABITAT operates at the level of individual (suspect) SQL query subexpressions, promoting debugging at a considerably finer granularity.

Quite different from programming languages, support for language-level query debugging is rarely found in the database domain. Various notions of provenance [Cheney et al. 2009] that explain the presence or absence of expected rows in a query result are close to HABITAT's objective of *declarative* query debugging [Tran and Chan 2010; Chapman and Jagadish 2009; Herschel and Hernández 2010]. However, explanations are either

(1) *plan-based*, at the level of algebraic primitives, and thus disconnected from the user-facing syntax [Chapman and Jagadish 2009; Tran and Chan 2010], or
(2) *instance-based* and thus fail to address bugs in the query itself—the missing row $\langle 2, 10 \rangle$ of Debug Session 3 is a so-called *never answer* that cannot be explained by these approaches [Herschel and Hernández 2010].

Further, as of today, these techniques cover a limited subset of SQL (conjunctive queries with `UNION` plus restricted forms of grouping and aggregation). None of the debugged SQL queries that we discussed in this article would fall into this class.

In fact, plan-based approaches to query explanation and performance debugging abound [Binnig and Kossmann 2007; Bruno et al. 2009]. Their usage requires an *apriori* understanding of how user-facing query constructs emerge in algebraic plans. Advances in query compilation, optimization, and execution largely disassociate surface syntax from plan internals (see Section 1), rendering these approaches and tools—*e.g.* visualizers for plan differences with respect to index usage, join method or order changes [Liang 2009]—useful only if in the hands of database engine experts. PerfXplain [Khoussainova et al. 2012] offers explanations for the unexpected disappointing performance of MapReduce jobs. Explanations take the form of conjunctive predicates that refer to physical parameters (*e.g.* the block size in use) of the underlying compute cluster, aiming to provide output that may be interpreted by MapReduce application developers (as opposed to the cluster's administrator). Finally, the database engine itself is the debugged subject in recent developments that promote the systematic synthetization of SQL text to exercise database kernel features and to assess their correct operation [Bruno 2010; Bati et al. 2007].

Rover [Grust et al. 2007b] realizes observational debugging for XQuery. Unlike HABITAT, Rover requires changes to the underlying database kernel (a new trace operator $\triangledown$ is injected into MonetDB/XQuery) to support the collection of observation data. HABITAT's closure-based compiler follows a different and much less invasive approach that (1) is applicable to any off-the-shelf SQL RDBMS and (2) comes with a significantly lighter runtime impact.

Inspector Gadget [Olston and Reed 2011], a debugging framework for the Pig Latin dataflow language, shares with HABITAT a non-invasive approach to debugger construction: debugging affects the query layer (as opposed to the system layer) only. While HABITAT's operation revolves around the concept of observable SQL subexpressions (Section 3.1), users of Inspector Gadget instrument existing dataflows with *monitor agents* that collect observations about passing data items but otherwise act like the identity. The placement and coordination (via a simple messaging API) of these agents is a coding activity but allows for the implementation of a variety of debugging behaviors, including performance debugging. In this respect, HABITAT's *mark and observe* model is less flexible but stays true to its principle to exclusively operate on the surface language level (arguably a better fit for SQL's declarative nature).

Understanding a SQL subexpression as a closure that captures variable bindings as well as expression value, makes HABITAT a relative of declarative debuggers for functional programming languages [Pope and Naish 2003; Marlow et al. 2007]. Much

like HABITAT, these debuggers transform the debugged program—through lambda lifting (β-abstraction, essentially) [Johnsson 1985]—such that subexpressions with free variables may be observed in isolation.

We close by noting that HABITAT lays the complete groundwork for the application of *algorithmic debugging* to SQL. Originally devised in the context of logic programming languages, in this particular debugging style a debugger automatically generates a minimal set of simple *yes/no?* questions about the observed and expected behavior of a program [Silva 2011; Shapiro 1983]. Starting from the program's errorneous output, the user's answers guide a backwards traversal of a detailed computation trace—or computation tree—to identify the suspect subcomputation(s) that led to the error. The computation tree is obtained through meta-interpretation [Naish 1997]: the buggy subject program is instrumented (or transformed) such that its subcomputations return computation subtrees along with the normal result. HABITAT's compilation scheme enables the construction of such computation trees for SQL: subexpressions are compiled such that their evaluation returns all bindings of free row variables along with the normal result. This captures everything that is required to enable the user to judge correctness (responding with *yes/no?*) when she observes subexpression evaluation.

Work reported in [Caballero et al. 2012] indeed suggests that algorithmic debugging is effective for SQL. This recent approach transforms SQL views into a logic program that is then subjected to algorithmic debugging. The debugger operates at the granularity of table or view definitions, however. The subcomputations of a view are the views and base tables it refers to. Consequently, view definitions as a whole are being identified as correct or erroneous: subqueries are not considered as computations of their own and the debugger would not be able to spot the bugs present in Queries 1 and 3. In comparison, HABITAT operates at a considerable finer granularity. Computation trees based on the SQL compiler of Section 3.3 can guide users to faulty query pieces at the level of individual subexpressions.

6. CONCLUSION

Authoring bug-free SQL queries can be tricky at times and the language clearly deserves a debugging approach that fits its set-oriented computational model: the iterated evaluation of expressions under varying row variable bindings. HABITAT's *mark and observe* implements such a debugging paradigm via *a new type of SQL compiler* based on the notion of closures. Users are permitted to think in terms of SQL's surface syntax (*mark*) and simple tabular representations of evaluated expressions (*observe*). HABITAT innovates the unimpeded observation of expressions, including those that yield runtime errors if evaluated in the context of the original query. Observation data is collected on and by the target database host itself, *i.e.* in the most authentic execution environment.

Usability aspects of a complete HABITAT prototype (see Figure 25) and the effectiveness of the *mark and observe* paradigm for SQL debugging have already been demonstrated [Grust et al. 2011]. HABITAT constitutes a tool that supports the dissection and understanding of SQL queries also for those users that have not been exposed to query engine internals yet [Murphy et al. 2008]. Bug hunting aside, we have successfully used HABITAT in classes to teach the intricacies of the SQL language and the semantics of correlated subqueries, in particular. We noticed that HABITAT enables students to acquaint themselves with complex query authoring more quickly.

Further work will shed light on how more advanced debugging methodologies that rest on HABITAT's groundwork can benefit query authors. Among others, the following options appear interesting to pursue:

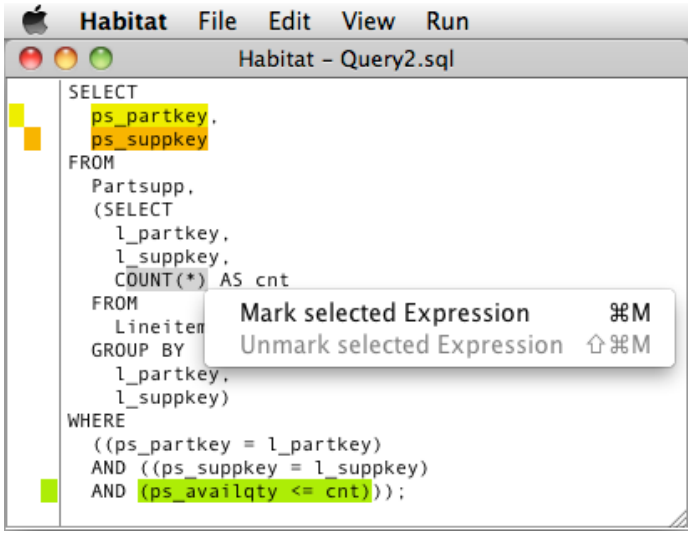


Fig. 25. Placing the markings of Figure 7(b) using the prototypical HABITAT debugger.

(1) Complementary to *mark and observe*, a granular *algorithmic debugging* front-end for HABITAT (Section 5) now becomes a possibility. Offering the use of Pig Latin-style generated example data (Section 4.1) will help users to respond to the *yes*/*no?* questions being asked.

(2) Users are not only able to observe the result of arbitrary subexpressions. Observations could also be *edited* and query evaluation would then resume based on the modified intermediary result—enabling live debugging in terms of *what if?* scenarios. Such a *stop-and-go* discipline of query execution is supported by compositional compilation and Invariant o|i (Section 3.2): on resumption, the residual observer query can read its input off the user-edited tabulation. The immediate connection of query authors with their creation would be strengthened, a crucial principle of (database) usability [Victor 2012; Jagadish et al. 2007].

APPENDIX

A. COMPILING QUERY 1

HABITAT's SQL compiler is defined in terms of three functions that map SQL constructs to algebraic plan fragments:
— $\Rrightarrow$ (translates the syntactic components of a SQL fullselect expression),
— $\Rightarrow$ (admits the use of single-row, single-column subqueries in place of scalars), and
— $\mapsto$ (compiles scalar subexpressions, including predicates and aggregates).
Figures 19 to 21 of Section 3.3 use inference rules to specify these functions on a per-construct basis. In a rule

$$\frac{a_1 \quad \cdots \quad a_n}{c} \ ,$$

the consequent c defines how to infer an algebraic plan fragment for a given SQL language construct e. Rules recursively invoke their antecedents $a_1, \ldots, a_n$—again specificied in terms of inference rules—to infer the plan fragments for the subconstructs contained in e (if any). It is a feature of HABITAT's approach that the consequents of all rules infer a plan fragment that may be sensibly evaluated on its own—this is key to enable the observation of arbitrary SQL constructs.

Figure 26 shows the resulting stack of inference rules, or *proof tree*, for SQL Query 1 (discussed in Section 3.2):

```
SELECT v0.p_partkey AS p_partkey,
       v0.p_name AS p_name,
       (SELECT v1.ps_supplycost AS ps_supplycost
          FROM Partsupp AS v1
         WHERE v0.p_partkey = v1.ps_partkey) AS ps_supplycost
  FROM Part AS v0   .
```

Due to its size we have split the proof tree and display it in two separate figures: Figure 27 replaces the dashed rectangle in Figure 26. (Note that both proof trees refer to each other: the proof subtree of Figure 27 refers to the (column set, query) pair (cs_2, q_2) defined in Figure 26. The latter refers to q_8 to receive the algebraic plan fragment for the nested subquery in Query 1). To remove clutter, we have omitted the component $gs = \varnothing$ in these figures; gs addresses grouping and is immaterial for this example, see Section 3.2.

The output of this compilation process, inferred by the proof tree's root (see the consequent of the bottom rule in Figure 26), is the table algebra expression

$$\pi_{\mathsf{o,i},cs_3}(\pi_{\mathsf{o,i},ps_3}(q_2 \bowtie_{\mathsf{i}} \pi_{\mathsf{i:o,\_.ps\_supplycost}}(q_8)))$$

where column set $cs_3 = \{\mathsf{\_.p\_partkey}, \mathsf{\_.p\_name}, \mathsf{\_.ps\_supplycost}\}$. A plan diagram of this algebraic expression is displayed in Figure 18 of Section 3.2. In the diagram, adjacent projection operations have been collapsed to aid the rendering.

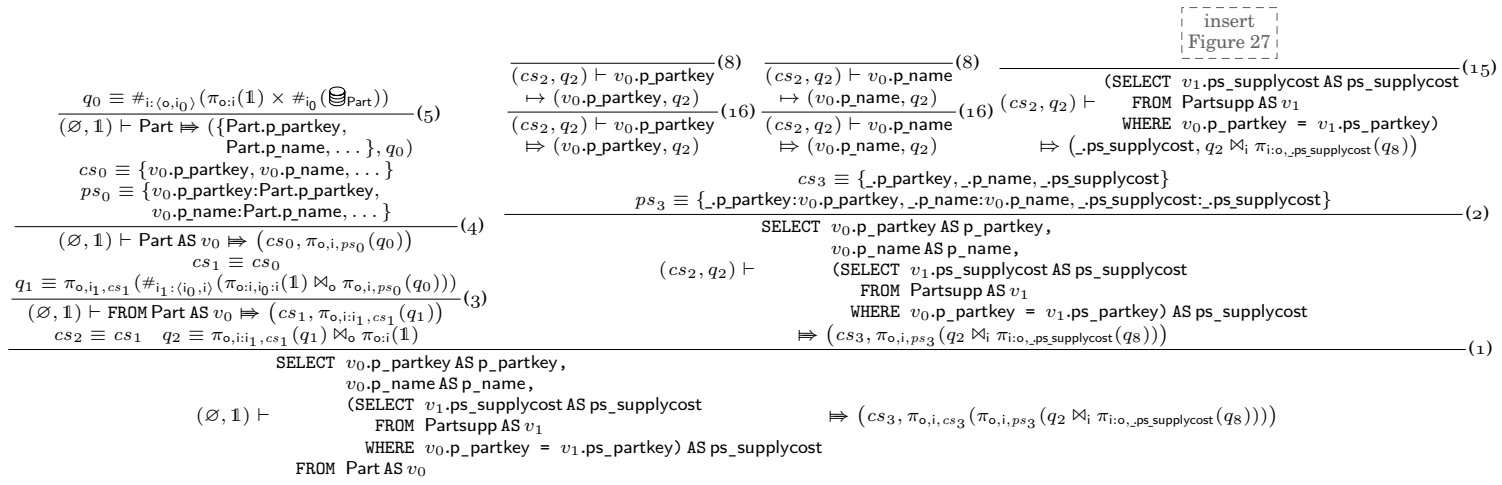


Fig. 26. Proof tree that infers the algebraic plan for Query 1 (continued in Figure 27). Inference rules are numbered according to Figures 19 to 21.

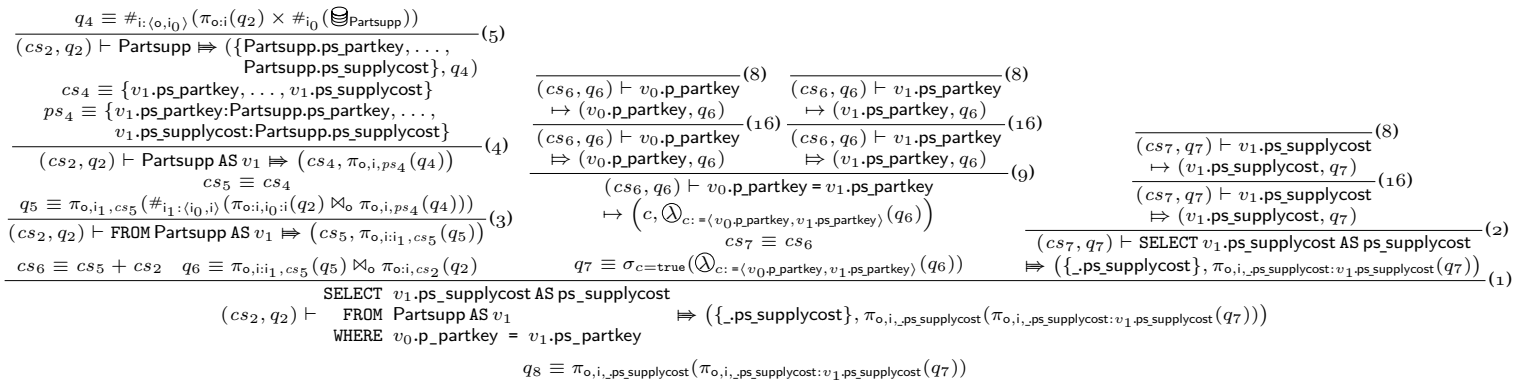


Fig. 27. Continuation of the proof tree for Query 1 (replaces dashed rectangle in Figure 26).

B. RENDERING TABULATIONS

A HABITAT observation display for a Marking ⓧ is a tabular representation of a closure that captures the value of the marked SQL subexpression, together with the free in-scope row variables, say $v_0, v_1, \ldots, v_{n-1}$. (Equivalently: the display will tabulate the n-ary function $f_{\text{ⓧ}}(v_0, v_1, \ldots, v_{n-1})$.) To create the display, HABITAT merges

(1) the n binding tables $q_{v_0}, q_{v_1}, \ldots, q_{v_{n-1}}$, [*input*]
and
(2) table $q_{\text{ⓧ}}$ that holds the results of the individual evaluations of the marked subexpression (Section 3). [*output*]

As the row variables' binding sites as well as Marking ⓧ itself may reside in separate (yet nested) scopes, HABITAT consults the $n + 1$ participating tables' leading o|i columns to perform the merge (recall Invariant o|i of Section 3). Let s_i denote the parent scope enclosing scope s_{i+1}, $i \geqslant 0$. Define $s(t) \equiv i$ if o|i-wrapped table t represents a subexpression evaluated (or a row variable bound) in scope s_i. Wlog, assume $s(q_{v_0}) \leqslant s(q_{v_1}) \leqslant \cdots \leqslant s(q_{v_{n-1}}) \leqslant s(q_{\text{ⓧ}})$.[6]

HABITAT constructs the display according to

$$\text{M}(q_{v_0}, \text{M}(q_{v_1}, \text{M}(\ldots, \text{M}(q_{v_{n-1}}, q_{\text{ⓧ}}) \cdots)))$$

where

$$\text{M}(q, q') \equiv \begin{cases} q \ni q' & \text{if } s(q) < s(q') \\ q \parallel q' & \text{if } s(q) = s(q') \end{cases}$$

and $s(\text{M}(q, q')) \equiv s(q')$. The two cases are as follows:

(1) $q \ni q'$: We have $\pi_{\text{i}}(q) \supseteq \pi_{\text{o}}(q')$ as a consequence of the lifting scheme (Section 3). Conceptually, perform the right outerjoin $q \bowtie_{\text{i=o}} q'$ to merge the two tables. Then render the outerjoin result as an observation display. If q' represents a table-valued SQL subexpression, draw a *nested* display and render `null` as an empty nested table (see Figure 28 from which the display in Figure 5 is derived.).
(2) $q \parallel q'$: We have $\pi_{\text{i}}(q) \supseteq \pi_{\text{i}}(q')$ or $\pi_{\text{i}}(q') \supseteq \pi_{\text{i}}(q)$, due to unsatisfied `WHERE` or `HAVING` predicates that may inhibit expression evaluation under particular bindings. Align tables q, q' with respect to their i-columns; if required for alignment, insert ▨ rows—*i.e.* conceptually perform the full outerjoin $q \bowtie_{\text{i}} q'$ and render `null` as ▨ (see Figure 9).

A further Marking ⓨ may be incrementally merged into an existing display D, giving $\text{M}(D, q_{\text{ⓨ}})$, if D includes the free variables of ⓨ (see Debug Session 3). To exemplify, the observation displays of Figures 8 and 9 have been rendered according to $q_{v_0} \ni (q_{②} \parallel q_{③})$ and $q_{v_0} \parallel q_{v_1} \parallel q_{④} \parallel q_{⑤} \parallel q_{⑥}$, respectively.

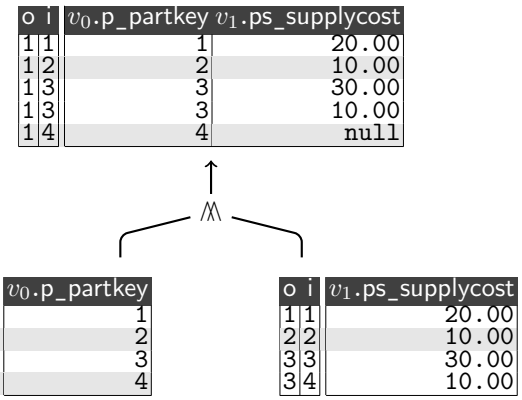
REFERENCES

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

BATI, H., GIAKOUMAKIS, L., HERBERT, S., AND SUMA, A. 2007. A Genetic Approach for Random Testing of Database Systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, Vienna, Austria, 1243–1251.

BINNIG, C. AND KOSSMANN, D. 2007. Reverse Query Processing. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*. IEEE, Istanbul, Turkey, 506–515.

[6]This is a consequence of SQL's scoping rules: the variables $v_0, v_1, \ldots, v_{n-1}$ are accessible in Marking ⓧ and thus must be defined in the scopes enclosing ⓧ.



(a) Binding table q_{v_0} in scope s_0.

(b) Table $q_{①}$ of results for Marking ① in scope s_1.

Fig. 28. HABITAT merges the tables q_{v_0}, $q_{①}$ to prepare the rendering of the nested observation display of Figure 5 ($q_{①}$ represents a table-valued observation). Since v_0 is bound in scope s_0 enclosing scope s_1, we have $\lambda\!\!\!\lambda(q_{v_0}, q_{①}) = q_{v_0} \ni q_{①} \approx q_{v_0} \bowtie\!\!\!\!\!\!\perp_{\mathsf{i=o}} q_{①}$ (modulo column renaming).

BRUNO, N. 2010. Minimizing Database Repros Using Language Grammars. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*. ACM, Lausanne, Switzerland, 382–393.

BRUNO, N., CHAUDURI, S., AND RAMAMURTHY, R. 2009. Interactive Plan Hints for Query Optimization. In *Proceedings of the 35th ACM SIGMOD International Conference on Management of Data*. ACM, Providence, USA, 1043–1046.

CABALLERO, R., GARCIA-RUIZ, Y., AND SÁENZ-PÉREZ, F. 2012. Declarative Debugging of Wrong and Missing Answers for SQL Views. In *Proceedings of the 11th International Symposium on Functional and Logic Programming (FLOPS)*. Springer, Kobe, Japan.

CHAPMAN, A. AND JAGADISH, H. 2009. Why Not? In *Proceedings of the 35th ACM SIGMOD International Conference on Management of Data*. ACM, Providence, USA, 523–534.

CHENEY, J., CHITICARIU, L., AND TAN, W. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases 1,* 4, 379–474.

CHOPRA, S., OLSTON, C., AND SRIVASTAVA, U. 2009. Generating Example Data for Dataflow Programs. In *Proceedings of the 35th ACM SIGMOD International Conference on Management of Data*. ACM, Providence, USA, 245–256.

Embarcadero Technologies. *The Embarcadero SQL Debugger*. Embarcadero Technologies. `http://www.embarcadero.com/products/debugger`.

GANSKI, R. AND WONG, H. 1987. Optimization of Nested SQL Queries Revisited. *SIGMOD Record 16,* 3, 23–33.

GOGOLLA, M. 1990. A Note on the Translation of SQL to the Tuple Calculus. *SIGMOD Record 19,* 1, 18–22.

GRUST, T., KLIEBHAN, F., RITTINGER, J., AND SCHREIBER, T. 2011. True Language-Level SQL Debugging. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT)*. ACM, Uppsala, Sweden, 562–565.

GRUST, T., MAYR, M., RITTINGER, J., SAKR, S., AND TEUBNER, J. 2007a. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In *Proceedings of the 33rd ACM SIGMOD International Conference on Management of Data*. ACM, Beijing, China, 1162–1164.

GRUST, T., RITTINGER, J., AND TEUBNER, J. 2007b. Data-Intensive XQuery Debugging with Instant Replay. In *Proceedings of the 4th International Workshop on XQuery Implementation, Experience, and Perspective (XIME-P)*. ACM, Beijing, China, 4:1–4:6.

HARITSA, J. AND REDDY, N. 2005. Analyzing Plan Diagrams of Database Query Optimizers. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, Trondheim, Norway, 1228–1239.

HERSCHEL, M. AND HERNÁNDEZ, M. 2010. Explaining Missing Answers to SPJUA Queries. *PVLDB 3,* 1–2, 185–196.

JAGADISH, H., CHAPMAN, A., ELKISS, A., JAYAPANDIAN, M., LI, Y., NANDI, A., AND YU, C. 2007. Making Database Systems Usable. In *Proceedings of the 33rd ACM SIGMOD International Conference on Management of Data*. ACM, Beijing, China, 13–24.

JOHNSSON, T. 1985. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings Functional Programming Languages and Computer Architecture (FPCA)*. Springer, Nancy, France, 190–203.

KATZ, I. AND ANDERSON, J. 1987. Debugging: An Analysis of Bug-Location Strategies. *Human-Computer Interaction (HCI) 3,* 4, 351–399.

KHOUSSAINOVA, N., BALAZINSKA, M., AND SUCIU, D. 2012. PerfXplain: Debugging MapReduce Job Performance. *PVLDB 5,* 7, 598–609.

KIM, W. 1982. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems (TODS) 7,* 3, 443–469.

LANDIN, P. 1964. The Mechanical Evaluation of Expressions. *The Computer Journal 6,* 4, 308–320.

LIANG, R. 2009. What's Changed between my New Query Plan and the Old One? Plan Comparison (built-in function `diff_plan_outline()`) in Oracle 11gR2. *Insights into the working of the Oracle Optimizer* (blog), `https://blogs.oracle.com/optimizer/entry/whats_changed_between_my_new_query_plan_and_the_old_one`.

MARLOW, S., IBORRA, J., POPE, B., AND GILL, A. 2007. A Lightweight Interactive Debugger for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM, Freiburg, Germany, 13–24.

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

MOTRO, A. 1986. Query Generalization: A Method for Interpreting Null Answers. In *Proceedings from the First International Workshop on Expert Database Systems (EDS)*. Benjamin/Cummings, Kiawah Island, USA, 597–616.

MURPHY, L., LEWANDOWSKI, G., MCCAULEY, R., SIMON, B., THOMAS, L., AND ZANDER, C. 2008. Debugging: The Good, the Bad, and the Quirky—A Quantitative Analysis of Novices' Strategies. *SIGCSE Bulletin 40,* 1, 163–167.

NAISH, L. 1997. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming 1997,* 3.

OLSTON, C. AND REED, B. 2011. Inspector Gadget: A Framework for Custom Monitoring and Debugging of Distributed Dataflows. *PVLDB 4,* 12, 1237–1248.

POPE, B. AND NAISH, L. 2003. 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)*. ACM, Uppsala, Sweden, 230–244.

SHAPIRO, E. 1983. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA.

SILVA, J. 2011. A Survey on Algorithmic Debugging Strategies. *Advances in Engineering Software 42,* 11, 976–991.

TRAN, Q. AND CHAN, C.-Y. 2010. How to ConQueR Why-Not Questions. In *Proceedings of the 36th ACM SIGMOD International Conference on Management of Data*. ACM, Indianapolis, USA, 15–26.

Transaction Processing Performance Council. *TPC-H, a Decision-Support Benchmark*. Transaction Processing Performance Council. `http://tpc.org/tpch/`.

VICTOR, B. 2012. Inventing on Princple. In *Proceedings of the Canadian University Software Engineering Conference (CUSEC)*. Montreal, Canada.

Received Month Year; revised Month Year; accepted Month Year