How *How* Explains What *What* Computes — How-Provenance for SQL and Query Compilers

Daniel O'Grady Tobias Müller Torsten Grust

[daniel.ogrady,to.mueller,torsten.grust]@uni-tuebingen.de

Universität Tübingen
Germany

Abstract

SQL emphasizes the *What*, the declarative specification of complex computations over a database. *How* exactly the individual parts of an intricate query interact and contribute to the result, often remains in the dark, however. *How*-provenance helps to understand queries and build trust in their results. We propose a new approach that **derives *how*-provenance for SQL at a fine granularity:** (1) every single piece of the result provides information on how exactly it did get there, and (2) the contribution of any query construct to the overall output can be assessed—from entire subqueries down to the subexpression leaf level. The method applies to real-world dialects of SQL and, more generally, to the modern breed of database systems that pursue a compilation-based approach to query processing.

1 *How*-Provenance Explains Queries

The declarative nature of SQL helps to keep focus on the actual subject matter while we author a query. Still, queries that exceed even moderate length and complexity thresholds may prove hard to read, let alone fully understand [6]. The SQL code excerpt of Figure 1 makes this more concrete (disregard the grey and dashed boxes for now). This query implements a parser for a bulk of chess games in algebraic notation, *e.g.*, ♙F2-F3 ♟E7-E5 ♙G2-G4 ♕D8-H4#. When the parser receives such game description input, it splits lines into individual moves and then returns those moves that are given in valid notation (see Figures 2(a) and 2(b) for sample input and output tables). It does take time to study this query and discover which role a query piece—a function or subquery, say—assumes. Even more so since the query deviates from the vanilla `SELECT-FROM-WHERE` pattern but calls on a variety of user-defined and built-in functions as well as recursive common table expressions (CTEs). Certainly, the query is complex enough that non-obvious bugs may lurk.

One way to gain such insight is to observe the query *while it produces* a selected piece of the result: for each cell o of the query's output table, record exactly which parts of the query contributed to the generation of o. Each o will be associated with its data flow that routes through specific syntactic parts of the query. As a result, we will be able to identify threads of computation that produce a given kind of output (*e.g.*, moves involving pawns ♙ or ♟), we can hope to spot dead code (obsolete functions, never-taken branches of conditionals), or we may identify the core query parts that contribute to all output items. The grey boxes of Figure 1 provide this information for the highlighted move ♘C2-D4 in the output table of Figure 2(b). (We take a closer look in Section 2.)

This work proposes one method that can derive such ***how*-provenance** [3, 5] for SQL. *How*-provenance is particularly useful if it

(1) is specific when we trace output items back through the query (we derive *how*-provenance at the level of individual output cells and identify contributing SQL expressions at the syntactic leaf level, *e.g.*, down to individual literals or arithmetic operations), and
(2) can explain complex queries that build on advanced SQL constructs, including rich data types (*e.g.*, arrays), user-defined and built-in functions, grouping and aggregation, window functions, or recursion.

We build on *dynamic backward slicing* [2, 11] and adapt it to work for queries (see Section 3). Once we have sketched its internals, we will see that this approach to *how*-provenance also applies to the modern breed of database systems that pursue query compilation (Section 4).

2 A Query Untangled

To untangle the chess parser query, let us consult the *how*-provenance of move ♘C2-D4 highlighted in the output table of Figure 2(b). The *how*-provenance of this output cell is formed by all SQL expressions (and their subexpressions) of Figure 1 that are enclosed in grey boxes. Nested boxes like $e_1 \otimes e_2$ make this more precise: operator $\otimes$ was evaluated but argument e_1 sufficed to determine the result (e_2 has not been evaluated and thus is not considered part of the cell's *how*-provenance).

We find that function `f12` and a number of further functions not shown in the figure were not involved at all. Indeed,

TaPP 2018, July 11–12, 2018, London, UK

```
1  CREATE OR REPLACE FUNCTION f12(str TEXT, len INT)
2  RETURNS INT AS $$
3    SELECT CASE WHEN left(str, 1) IN ('A','B','C','D'
4                                     ,'E','F','G','H')
5                THEN f13(right(str, -1), len+1)
6              ELSE -1
7         END;
8  $$ LANGUAGE SQL;
9
10 CREATE OR REPLACE FUNCTION f11(str TEXT, len INT)
11 RETURNS INT AS $$
12   SELECT CASE WHEN left(str, 1) = '×'
13               THEN f12(right(str, -1), len+1)
14             WHEN left(str, 1) = '–'
15               THEN f5(right(str, -1), len+1)
16             ELSE -1
17        END;
18 $$ LANGUAGE SQL;
19
20 CREATE OR REPLACE FUNCTION f6(str TEXT, len INT)
21 RETURNS INT AS $$
22   SELECT CASE WHEN left(str, 1) IN ('1','2','3','4'
23                                    ,'5','6','7','8')
24               THEN f7(right(str, -1), len+1)
25             ELSE -1
26        END;
27 $$ LANGUAGE SQL;
28
29 CREATE OR REPLACE FUNCTION f1(str TEXT, len INT)
30 RETURNS INT AS $$
31   SELECT CASE WHEN left(str, 1) IN ('♙','♟')
32               THEN f9(right(str, -1), len+1)
33             WHEN left(str, 1) IN ('♘','♗','♖','♕','♔'
34                                   ,'♞','♝','♜','♛','♚')
35               THEN f2(right(str, -1), len+1)
36             ELSE -1
37        END;
38 $$ LANGUAGE SQL;
39
40 WITH RECURSIVE t(game_id, index, move, rest, valid) AS (
41   SELECT g.id, 1 AS index,
42          '' AS move,
43          g.moves AS rest,
44          false AS valid
45   FROM   games AS g
46     UNION ALL
47   SELECT game_id,
48          1 + t.index + greatest(ix, 0) AS index,
49          left(t.rest, greatest(ix, 0)) AS move,
50          right(t.rest, -abs(ix)) AS rest,
51          ix >= 0 AS valid
52   FROM   t, LATERAL f1(t.rest, 0) AS ix
53   WHERE  t.rest <> ''
54 )
55 SELECT t.move, t.game_id
56 FROM   t
57 WHERE  t.valid
58 ORDER BY t.game_id, t.index;
```

Figure 1. Sample SQL query (excerpt). Boxes highlight evaluated subexpressions, *n* are heat annotations (see Section 4).

each output cell depends on a subset of functions only. (In general, we gain more insight if we successively focus on a variety of output cells—the page budget restrains us to a single cell here.) We focused on move ♘C2-D4 of game 1 but we observe that, generally, the subexpressions involved in processing all *preceding* moves of the same game (here: ♟B2-B3) are boxed as well. Games are therefore obviously parsed from left to right.

In all functions, the conditional CASE expressions never take the ELSE branch. This changes once we feed invalid

games

id	moves
1	♟B2-B3 ♘C2-D4
2	♙E4-E5 ♟D7-D5 ♙E5xD6e.p.

(a) Chess game input.

output

game_id	move
1	♟B2-B3
1	♘C2-D4
2	♙E4-E5
2	♟D7-D5
2	♙E5×D6e.p.

(b) Parsed output.

Figure 2. Input and output tables. Output cell ♘C2-D4 has been selected for *how*-provenance analysis.

move notation into the parser: apparently, a function return value of -1 serves as an error indicator. Function f1 serves as a hub that tells pawns from all other pieces. Both of its conditional's WHEN branches are taken, once for the first and second move in game 1, respectively. When move ♘C2-D4 is processed by the second WHEN branch, we see that only the first piece in the list ('♘','♗',...) actually is evaluated: this is sufficient to find that the move matches and processing proceeds with function f2. *How*-provenance thus reveals the lazy existential semantics of the IN predicate. Function f6 shows similar behavior: a comparison with the first four list elements in ('1','2','3','4',...) is enough to identify the column numbers of the final positions B3 and D4 in the two moves of game 1 (the comparisons with '1' and '2' fail for both positions but still have to be evaluated: comparisons proceed left to right). In f11, the first WHEN branch is entered only to find the predicate `left(str, 1) = '×'` unsatisfied—f12 thus is never called (instead, the second branch is entered to process the dash '-').

The recursive CTE in lines 40 to 58 appears to drive the entire parser as it is always involved. There is more to be learned about the recursion; we turn to this and the *heat annotations* *n* in Section 4 .

An inspection of the *how*-provenance of further output cells reinforces these observations: the SQL query of Figure 1 simulates a finite state machine whose states are represented as functions, each consuming a single character of the input string. Function calls model state transitions.

3 Less Declarative, More Insightful

This notion of *how*-provenance requires an internal representation of the subject query that can (1) capture the rich diversity of constructs in modern SQL dialects, and (2) admits data flow analysis and provenance annotation at a fine level of granularity, all the way down to the query's atomic subexpressions. We argue that, for a variety of reasons, relational algebra (*e.g.*, as used in [5]) does *not* make for such a suitable representation:

- In general, there is no bijection between the user-facing SQL syntax and the operator nodes of an algebraic plan: a single node may implement multiple SQL constructs (*e.g.*, a single *sort* node may support the evaluation of both,

```
Sort
 Output: t.game_id, t.move, t.index
 Sort Key: t.game_id, t.index
CTE t
 -> Recursive Union
   -> Seq Scan on public.games
      Output: games.id, 1, ''::text, games.moves, false
   -> Nested Loop
      Output: t_1.game_id, ((1+t_1.index)+GREATEST(ix.ix,0)),
         "left"(t_1.rest, GREATEST(ix.ix,0)),
         "right"(t_1.rest, (-abs(ix.ix))), (ix.ix>=0)
      -> WorkTable Scan on t t_1
         Output: t_1.game_id, t_1.index,
                 t_1.field, t_1.rest, t_1.valid
         Filter: (t_1.rest <> ''::text)
      -> Function Scan on ix
         Output: ix.ix
         Function Call: CASE WHEN ("left"(t_1.rest,1) ...
 -> CTE Scan on t
    Output: t.game_id, t.move, t.index
    Filter: t.valid
```

Figure 3. PostgreSQL plan for the recursive CTE in Figure 1. Expressions annotate the **bold** relational backbone.

a GROUP BY and an ORDER BY clause) and a single SQL construct may be represented by multiple plan nodes. Tracking data flow through the plan thus will not allow us to track *how*-provenance on the desired level of SQL surface syntax.

- Typically, expressions—*e.g.*, arithmetics, predicates, conditionals, string or array operations—are second class. Plans primarily represent the "relational spine" of a query (scans, selections, joins, or aggregations, say) in which expressions are mere annotations. See Figure 3 for a PostgreSQL plan that exhibits such a two-tiered query representation. This is not adequate whenever essential query logic is embodied by these expressions, recall our discussion in Section 2. (There are few proposals for plan algebras in which expressions are first class [4].)

Imperative programs simulate SQL. In the present work, instead, we map the SQL subject query into an imperative program. This makes the query susceptible to established dynamic program analyses—*backward slicing*, in particular—which we can adapt to derive *how*-provenance at the desired granularity level.

The semantics of rich SQL dialects is expressible in basic imperative languages. We require

- atomic data types that represent SQL's 1NF cell values,
- dictionary and list type constructors (to model row values and tables of rows, respectively),
- variable assignment and reference,
- statement sequences,
- standard control flow (`if · then ·`) and loop (`for · in ·`) constructs, as well as
- invocation of user-defined and provided library functions.

The approach does not hinge on a particular language. We use an abstract *kernel language* here but a subset of C or even LLVM IR code are just as suitable, for example. Section 4 elaborates.

A straightforward compositional translation establishes a bijection between SQL subexpressions and the generated program: each subexpression and SQL clause maps to its dedicated piece of program code. The program for a complex query is assembled from the program pieces of its subqueries and subexpressions. We pursue a *canonical translation* [10] that is only guided by the syntax and semantics of the SQL query and remains independent of any plan—our goal is to comprehend the query logic, not the particular plan the DBMS has chosen. This direct SQL-to-program correspondence is central—any program simplification or optimization is required to maintain the correspondence. To illustrate, the program fragment of Figure 4(a) implements the body of SQL function `f6` in Figure 1. *Code regions* (marked ⓘ) in this program directly correspond with subject query constructs, for example,

- region ⑤ represents the SQL character literal `'1'`, and
- region ② embodies the `IN` predicate. (Note how the imperative code simulates the system's lazy `IN` semantics—the translation can easily be adapted to implement a strict variant instead. This *will* affect the observed *how*-provenance.)

Dynamic backward slicing. To prepare provenance derivation, the program is lifted to operate over pairs $\langle v, R\rangle$ of regular values v and region sets R. *Invariantly, if v is associated with R, then v's value is determined by program code located in the set of regions R.* Lifting is a purely syntactic transformation that can be performed by a single pass over the statements in a program:

- replace expressions ℓ by $\langle \ell, R\rangle$ if they occur enclosed in regions R,
- augment the value $\langle v_e, R_e\rangle$ of a lifted expression e by regions R if e is evaluated in R, giving $\langle v_e, R_e \cup R\rangle$ (abbreviated $e \curvearrowleft R$ in Figure 4),
- recursively lift functions f to yield functions $\widehat{f}$ that accept and return lifted pairs,
- lift operators $\oplus$: $\langle v_1, R_1\rangle \mathbin{\widehat{\oplus}} \langle v_2, R_2\rangle = \langle v_1 \oplus v_2, R_1 \cup R_2\rangle$,
- lift **`if`** p **`then`** s: given $\widehat{p} = \langle v, R\rangle$, $\widehat{\textbf{if}\ p\ \textbf{then}}\ s$ branches on Boolean v and augments expressions in s by R.

Figure 4(b) shows the program of Figure 4(a) after lifting. This program calls on lifted variants of SQL library functions like `left` and `right`. We may choose to fully model such built-ins b in the imperative language or, alternatively, treat them as black boxes: assume e_i returns $\langle v_i, R_i\rangle$, then $\widehat{b}(e_1, \ldots, e_n) = \langle b(v_1, \ldots, v_n), R_1 \cup \cdots \cup R_n\rangle$.

The evaluation of the lifted program will yield a list of dictionaries (*i.e.*, a table of rows) in which each entry (*i.e.*, cell) is paired with the program regions that determined its value. The mentioned SQL-to-program correspondence immediately yields the cell's *how*-provenance with respect to the subject query, at the level of SQL leaf expressions. Cell C2-D4 in the output table of Figure 2(b) is associated with the boxed SQL expressions of Figure 1.

```
1   ⌈c := left(str, 1)
2
3   for x in ['1', ..., '8']
4 ① ⌈ b := (c = x)
5   ② ⌈ if b then
6   ③ ⌈   return f7(right(str, -1),
7   ⌊                 len+1)
8   ⌋ ⌋
9 ④ ⌊return -1
```

(a) Canonical program.

```
1  c := left(str, ⟨1, {①}⟩) ↶ {①}
2
3  for x in [⟨'1',{①, ⑤}⟩, ..., ⟨'8', {①, ⑫}⟩]
4     b := (c = x) ↶ {①, ②}
5     if b then
6        return f7(right(str, ⟨-1, {①, ②, ③}⟩) ↶ {①, ②, ③},
7                  len + ⟨1, {①, ②, ③}⟩) ↶ {①, ②, ③}
8
9  return ⟨-1, {④}⟩
```

(b) Lifted program.

Figure 4. Kernel language programs for the body of SQL function `f6`. Labels ⓘ denote code regions.

This derivation of code region sets is an adaptation of *dynamic backward slicing* [2, 11]. Effectively, the monotonic accumulation of region sets builds a dynamic dependency graph of the program (and thus the query) in which output cells are connected with subexpressions that determined their value. Note that this graph may also be traversed in the opposite direction to connect SQL subqueries and expressions with all output values they influenced.

4 Beyond *How*-Provenance for SQL

This basic approach to *how*-provenane derivation can be tweaked and extended in several dimensions. We sketch a few possible directions here.

Heat (bags of regions). If we alter the lifted program to collect *bags* of regions—instead of sets—and thus keep track of multiplicities, the analysis will additionally reveal how *often* regions have been entered to compute a given value. Such region counts (or *heat*) can, for example, (1) help to identify "hot spots" in a query, *e.g.*, hubs that are central to the query logic (like function `f1` in our sample query), or (2) disclose the behavior of recursive parts of a query. In Figure 1, a heat annotation ***n*** indicates that the associated boxed expression has been evaluated *n* times. We find, for example, that the recursive subquery of the CTE (lines 47-53) has been evaluated twice, once per move in chess game 1.

***How*-Provenance for PL/SQL and transactions.** An imperative representation language facilitates the analysis of functions written in procedural style, *e.g.*, using PL/SQL. Likewise, transactions, *i.e.*, sequences of queries and updates, can readily be embraced by the imperative approach. This is currently under investigation.

***How*-Provenance and query compilers.** Our field currently sees a renaissance of database systems whose query engines compile SQL queries into imperative intermediate (or directly executable) forms. *HyPer* [9], *LegoBase* [7], and *MemSQL* [1] are exemplary members of this family. Recalling our notation of Section 3, these systems emit program code that manipulates values v. Lifting this code to compute over pairs $\langle v, R\rangle$ would call for invasive engine changes that are hard to justify. Instead we propose a lighter "first v, then R" approach:

(1) Execute an instrumented variant of the generated code that logs its control flow decisions (conditional branches taken, loops exited) and locations of data access (array indices or record IDs) but otherwise manipulates and outputs values v as usual.
(2) In a second phase, possibly outside the system's engine, lift the code as described in Section 3 but only execute the slice that is concerned with the accumulation of regions sets R. Since this second execution exclusively manipulates region sets, it reads the logs of (1) to re-enact the original execution's control flow decisions.

Step (1) requires minimal system changes. We have successfully used a variant of this two-phase approach to derive *where*- and *why*-provenance for complex queries [8]. Along these lines, post-mortem *how*-provenance derivation could be added to modern query compilers.

References

[1] 2016. MemSQL MPL. https://docs.memsql.com/concepts/v6.0/code-generation. (2016).
[2] J. Cheney. 2007. Program Slicing and Data Provenance. *IEEE Data Engineering Bulletin* 30, 4 (2007), 22–28.
[3] J. Cheney, L. Chiticariu, and W.-C. Tan. 2007. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2007).
[4] M. Cherniack and S. Zdonik. 1998. Changing the Rules: Transformations for Rule-Based Optimizers. In *Proc. SIGMOD*. Seattle, WA, USA.
[5] T.J. Green, G. Karvounarakis, and V. Tannen. 2007. Provenance Semirings. In *Proc. PODS*. Beijing, China.
[6] T. Grust and J. Rittinger. 2013. Observing SQL Queries in their Natural Habitat. *ACM TODS* 38, 1 (2013).
[7] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. 2014. Building Efficient Query Engines in a High-Level language. In *Proc. VLDB*. Hangzhou, China.
[8] T. Müller and T. Grust. 2015. Provenance for SQL Based on Abstract Interpretation: Value-less, but Worthwhile. In *Proc. VLDB*. Kohala Coast (Hawaii), USA.
[9] T. Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *Proc. VLDB*. Seattle, USA.
[10] T. Neumann and A. Kemper. 2015. Unnesting Arbitrary Queries. In *Proc. BTW*. Hamburg, Germany.
[11] M. Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984).