You Say ‘What’, I Hear ‘Where’ and ‘Why’ — (Mis-)Interpreting SQL to Derive Fine-Grained Provenance

Tobias Müller Benjamin Dietrich Torsten Grust

Universität Tübingen, Germany

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

ABSTRACT

SQL declaratively specifies *what* the desired output of a query is. This work shows that a non-standard interpretation of the SQL semantics can, instead, disclose *where* a piece of the output originated in the input and *why* that piece found its way into the result. We derive such data provenance for very rich SQL dialects—including recursion, windowed aggregates, and user-defined functions—at the fine-grained level of individual table cells. The approach is non-invasive and implemented as a compositional source-level SQL rewrite: an input SQL query is transformed into its own interpreter that wields data dependencies instead of regular values. We deliberately design this transformation to preserve the shape of both data and query, which allows provenance derivation to scale to complex queries without overwhelming the underlying database system.

PVLDB Reference Format:
Tobias Müller, Benjamin Dietrich, Torsten Grust. You Say ‘What’, I Hear ‘Where’ and ‘Why’—(Mis-)Interpreting SQL to Derive Fine-Grained Provenance. *PVLDB*, 11(11): 1536–1549, 2018.
DOI: https://doi.org/10.14778/3236187.3236204

1. DATA PROVENANCE EXPLAINS COMPLEX SQL QUERIES

A complex SQL query. In a hilly landscape, which marks are visible from your current location? That will depend on your position’s altitude and the height of the terrain around you: valleys are obscured by nearby ridges, while peaks, even if remote, may still be in view. The two-dimensional sketch of Figure 1 suggests one answer to the question: first, compute the running maximum (or: *max scan*) of view angles between our location ⊠ and the ever farther hill tops before us. Second, a mark is visible iff its angle is at least as large as the maximum angle α_i we have measured so far. We thus can spot the tree (its view angle α_3 exceeds the current maximum of α_2) while marks p_1 and p_2 are obscured.

The *max scan* technique does apply in three dimensions, but things get a bit more complicated. Figure 2(a) depicts the height map of a sample terrain in which shades of grey indicate altitude and ⊠ at $(x, y) = (17, 10)$ marks our location again. If we encode this terrain in a table `map`, see Figure 2(b), we can use the SQL query of Figure 3 to compute the visible spots (▨). The query uses a common table expression (CTE, `WITH`...) to structure the computation. We can spot the *max scan* in Lines 37 to 44, but the interplay of local table definitions, user-defined and builtin functions, and complex query logic (*e.g.*, the use of window functions) weaves a tangled web that is hard to see through. How does this query work and how does it adapt the two-dimensional *max scan* idea?

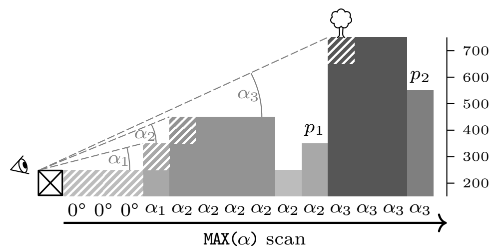
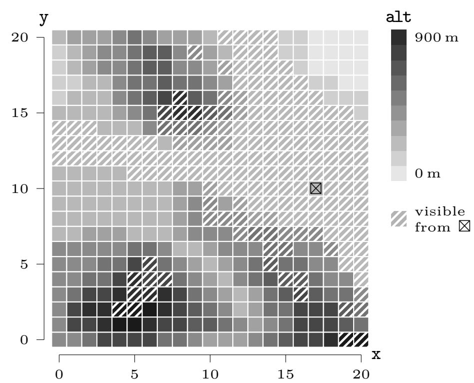


Figure 1: Visibility in a two-dimensional hilly landscape: spots marked ▨ are visible from ⊠. The max scan encounters the view angles $0° < \alpha_1 < \alpha_2 < \alpha_3$ from left to right.



(a) Height map with our location ⊠.

`map`

x	y	alt	
0	0	400.0	
0	1	400.0	
0	2	400.0	
⋮	⋮	⋮	
17	10	200.0	⊠
⋮	⋮	⋮	
20	18	0.0	
20	19	0.0	
20	20	0.0	

(b) Table `map`.

Figure 2: Height map of three-dimensional terrain and its tabular encoding. Again, spots marked ▨ are visible from ⊠.

An extended version of this paper with a comprehensive appendix is available at `https://arxiv.org/abs/1805.11517`.

Proceedings of the VLDB Endowment, Vol. 11, No. 11

DOI: https://doi.org/10.14778/3236187.3236204

```sql
-- Distance between points (x1,y1) and (x2,y2)
CREATE FUNCTION
dist(x1 int, y1 int, x2 int, y2 int) RETURNS float AS
$$
 SELECT sqrt((x2 - x1)^2 + (y2 - y1)^2)
$$ LANGUAGE SQL;

-- Number of steps on the line (x1,y1)-(x2,y2)
CREATE FUNCTION
steps(x1 int, y1 int, x2 int, y2 int) RETURNS int AS
$$
 SELECT greatest(abs(x2 - x1), abs(y2 - y1))
$$ LANGUAGE SQL;

-- Points (x,y) on the line (x1,y1)-(x2,y2)
CREATE FUNCTION
line(x1 int, y1 int, x2 int, y2 int) RETURNS TABLE(x int, y int) AS
$$
 SELECT x1 + round(i * ((x2 - x1) / steps(x1, y1, x2, y2))) AS x,
        y1 + round(i * ((y2 - y1) / steps(x1, y1, x2, y2))) AS y
 FROM   generate_series(0, steps(x1, y1, x2, y2)) AS i
$$ LANGUAGE SQL;

WITH
-- (1) Ray from ⊠ to (x1,y1) has points (rx,ry)
rays(x1, y1, rx, ry) AS (
 SELECT m.x AS x1, m.y AS y1, l.x AS rx, l.y AS ry
 FROM   map AS m,
        LATERAL line(17, 10, m.x, m.y) AS l(x,y)
 WHERE  m.x IN (0,20) OR m.y IN (0,20)   -- points on the border
),
-- (2) Angle between point (x,y) and ⊠
angles(x, y, angle) AS (
 SELECT m.x, m.y,
        degrees(atan((m.alt - 200) /   -- ⊠ is at altitude 200m
                (dist(m.x, m.y, 17, 10)))) AS angle
 FROM   map AS m
 WHERE  ROW(m.x, m.y) <> ROW(17, 10)
),
-- (3) Line of sight along each ray (uses a max scan)
max_scan(x, y, angle, max_angle) AS (
 SELECT r.rx AS x, r.ry AS y, a.angle, MAX(a.angle) OVER (
          PARTITION BY r.x1, r.y1
          ORDER BY dist(17, 10, r.rx, r.ry)) AS max_angle
 FROM   rays AS r, angles AS a
 WHERE  ROW(r.rx, r.ry) = ROW(a.x, a.y)
),
-- (4) Assemble visibility map from all lines of sight
visible(x, y, "visible?") AS (
 SELECT s.x, s.y, bool_or(s.angle >= s.max_angle) AS "visible?"
 FROM   max_scan AS s
 GROUP BY s.x, s.y
)
SELECT v.x, v.y, v."visible?"
FROM   visible AS v;
```

Figure 3: SQL query to compute visibility in three-dimensional terrain encoded in table `map`. A row $(x, y, \texttt{true})$ in result table `visible` indicates that spot (x, y) is visible from ⊠.

Data provenance offers answers to these and further questions [3, 12, 29, 44]. Provenance relates a query's individual input and output data items (table cells, say), sheds light on query internals and bugs, and helps to build trust in query results—a critical service to data-dependent science and society [19]. In our present case, we may hope that provenance helps to understand how the visibility *max scan* has been tweaked to function in three-dimensional terrain.

Clearly, the benefits of data provenance grow with the complexity of the query logic it is able to explain. As modern query languages continue to gain expressive constructs [43] and algorithms of increasing intricacy are cast into relational queries (*e.g.*, graph processing and machine learning tasks [1, 17, 27]), the gap between queries found in practice and existing approaches for provenance derivation widens considerably, however [12, 14, 25, 29]. The principal languages of study have been the (positive) relational algebra and its SQL equivalent. Grouping and aggregation can be handled by some approaches [15, 21] but are already considered challenging. In this light, the derivation of database provenance for complex queries found "outside the lab" appears elusive.

We set out to bridge this gap and enable the derivation of **fine-grained data provenance for a significantly richer family of SQL queries**. The admissable query dialect includes

- common table expressions including the recursive kind (`WITH RECURSIVE`...),
- window functions with arbitrary frame specifications as well as grouping and aggregation,
- scalar or table-valued builtin and user-defined functions,
- complex types (*e.g.*, row values and arrays), and
- subqueries without or with dependencies (through `LATERAL` or correlation) to their enclosing query.

We aim for compositionality, *i.e.*, these and further constructs may be nested arbitrarily as long as SQL's scoping and typing rules are obeyed.

The approach is based on a **non-standard interpretation of the SQL semantics**. This new interpretation focuses on the dependencies between input and output data items—the items' values play a secondary role only. The required interpreter is systematically derived from the original value-based query and formulated in SQL itself. As long as we can perform this derivation for a SQL construct or idiom, the approach is ready to embrace it. While we work with PostgreSQL in what follows, the method may be implemented on top of any SQL-based RDBMS. *No engine internals need to be altered.*

Goal (*Where-* and *Why-*Provenance). *Given a SQL subject query q and its output table t, for each cell o of t compute*

- *which input table cells were* copied or transformed *to determine o's value, and* [*where*-provenance]
- *which input table cells were* inspected to decide *that o is present in the output at all.* [*why*-provenance]

This understanding of *where-* and *why*-provenance largely coincides with that of earlier work [6, 14, 21]—Section 5 notes where we deviate. Together, both types of provenance characterize the exact set of input table cells that were sourced by query q, providing invaluable information for query explanation and debugging [26]. Such complete cell-level provenance provides the most detailed insight into query behavior but comes at a size and speed cost. We thus also outline how coarser granularity may be traded for performance.

Data provenance explains queries. Once we perform provenance derivation for the SQL query of Figure 3, we can understand how the data in input table `map` (Figure 2(b)) is used to compute visibility. Figure 4 highlights those points in the input terrain that determine the visibility of spots ❶ and ❷. Since we compute the provenance of the entire query output, we could have selected any spot and investigated the provenance of its visibility. Provenance analysis reveals that the query "shoots" rays from ⊠ to the points at the border of the map (see the ←-- in Figure 4), effectively leaving us with a two-dimensional problem that can be tackled via the *max scan* technique of Figure 1. We see that the visibility of point (x, y) only depends on the points on the ray between ⊠ and (x, y), *i.e.*, those points visited by the *max scan* so far. These and similar findings help to untangle the query and build trust in its result.

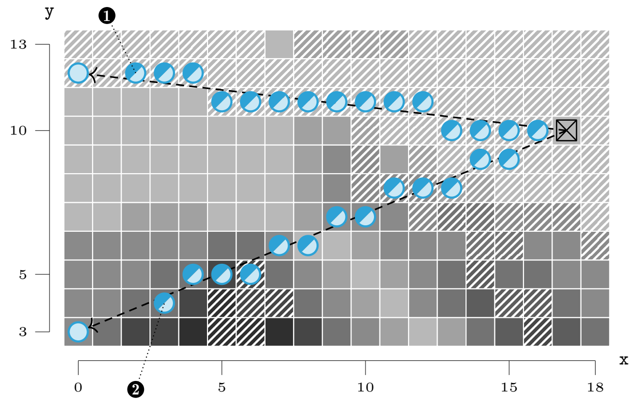


Figure 4: Excerpt of terrain map after provenance derivation. We find that the (non-)visibility of spots ❶ and ❷ is *where*- as well as *why*-dependent on the points marked and only *why*-dependent on the two border points marked .

2. FROM VALUES TO DEPENDENCY SETS

Regular query evaluation computes the *value* of an output cell o through the inspection and transformation of input *values*. In this work, instead, we focus on o's *dependency set*:

Definition 1 (Dependency Set). *Given an output cell o, the* dependency set *of o is the (possibly empty) set $\{i_1, i_2, \ldots, i_n\}$ of input table cells that were copied, transformed, or inspected to compute the value of o. Values are secondary: o and $i_1, \ldots, i_n$ identify the cells themselves, not their values. We use $\mathbb{P}$ to denote the type of dependency sets.*

It is our main hypothesis that a *non-standard interpretation* of queries provides a solid foundation to reason about this shift of focus from values to dependency sets [10]. We pursue a purely SQL-based implementation of this shift: from the original value-based SQL query, we generate its dependency-deriving variant—or *interpreter*, for short—through query transformation. Since this variant manipulates dependency sets and is oblivious to values, we supply just enough runtime information to guide the interpreter whenever the original query made a value-based decision.

Overview. These considerations shape a two-phase approach. Let q denote the original SQL query:

Phase $\mathbb{1}$: Instrument q to obtain query $q^{\mathbb{1}}$ that performs the same value-based computation as q and outputs the same result. Whenever $q^{\mathbb{1}}$ makes a value-based decision (*e.g.*, let a row pass a predicate or locate a row inside a window frame), those values relevant to this decision are appended to logs as a side effect of evaluation.

Phase $\mathbb{2}$: Evaluate interpreter $q^{\mathbb{2}}$ that performs dependency derivation. Query $q^{\mathbb{2}}$ reads, manipulates, and outputs tables of dependency sets. To properly replay the decisions made by $q^{\mathbb{1}}$, $q^{\mathbb{2}}$ additionally consults the logs written in Phase $\mathbb{1}$.

We shed light on Phases $\mathbb{1}$ and $\mathbb{2}$ and their interaction in the upcoming Sections 2.2 and 2.3. The construction of the instrumented query $q^{\mathbb{1}}$ as well as the interpreter $q^{\mathbb{2}}$—both can be built in tandem—are the subject of Section 3. Since the evaluation of $q^{\mathbb{1}}$ incurs logging effort and $q^{\mathbb{2}}$ needs to manipulate sets instead of first normal form (1NF) values, Section 4 discusses the sizes of both logs and result tables, quantifies the impact on query evaluation time, and discusses SQL interpretation at the coarser row granularity. Sections 5 and 6 review related efforts and wrap up.

2.1 Changing Types, Preserving Shape

Consider q, a general template for a single-table `SELECT-FROM-WHERE` block:

$$q(e, p, t) = \texttt{SELECT}\ e(x)\ \texttt{FROM}\ t\ \texttt{AS}\ x\ \texttt{WHERE}\ p(x) \ .$$

The type of q, namely $\forall a, b\colon (a \to b) \times (a \to \texttt{bool}) \times \{a\} \to \{b\}$, is *parametric* [48] in the row types a and b of the input and output tables.[1] Any instantiation of type variables a and b yields a workable *filter-project* query. If t is table `map` of Figure 2(b) and e projects on its third column `alt` of type `real`, then $a \equiv \texttt{int} \times \texttt{int} \times \texttt{real}$, $b \equiv \texttt{real}$, and q has type

$$(\texttt{int} \times \texttt{int} \times \texttt{real} \to \texttt{real}) \times (\texttt{int} \times \texttt{int} \times \texttt{real} \to \texttt{bool}) \times \{\texttt{int} \times \texttt{int} \times \texttt{real}\} \to \{\texttt{real}\} \ .$$

With the shift from values (Phase $\mathbb{1}$) to dependency sets (Phase $\mathbb{2}$) we are interested in the particular row type instantiation in which *all column types are replaced by* $\mathbb{P}$, the type of dependency sets. If we perform this shift for the former example, we get $a \equiv \mathbb{P} \times \mathbb{P} \times \mathbb{P}$, $b \equiv \mathbb{P}$, yielding query $q^{\mathbb{2}}$ of type

$$(\mathbb{P} \times \mathbb{P} \times \mathbb{P} \to \mathbb{P}) \times (\mathbb{P} \times \mathbb{P} \times \mathbb{P} \to \texttt{bool}) \times \{\mathbb{P} \times \mathbb{P} \times \mathbb{P}\} \to \{\mathbb{P}\} \ ,$$

over tables of dependency sets. Most importantly, $q^{\mathbb{2}}$ is indifferent to the choice of row types [48]: it continues to implement the *filter-project* semantics.

This parametricity of queries is central to the approach:

- The shift to $\mathbb{P}$ in Phase $\mathbb{2}$ not only preserves the shape of the query type but, largely, also the *syntactic shape* of the SQL query. We can thus derive an interpreter for a given query via a transformation that is compositional (will not break in the face of complex queries) and extensible (can embrace new constructs as the SQL language grows). The query execution plans of the transformed queries resemble those of the originals which reduces the risk of overwhelming the query processor, an adverse effect that has been observed by earlier work on data provenance for SQL [39].
- The value-based and dependency-based queries read and output tables of the same width and row count: we also preserve the *shape of the data* (albeit not its type). A one-to-one correspondence between the cells in value-based and dependency-carrying tables admits a straightforward association of individual data items with their provenance.

In Phase $\mathbb{2}$, note that predicate p (of type $\mathbb{P} \times \mathbb{P} \times \mathbb{P} \to \texttt{bool}$) exclusively receives dependency sets as input. These dependency sets reveal what influenced the predicates's outcome but do not let us compute the Boolean value of the original p. We address this in Phase $\mathbb{1}$ in which we instrument the original query such that the outcome of relevant value-based computation is logged. The interpreter of Phase $\mathbb{2}$ then uses the log to look up p's Boolean value and to re-enact the original query's behavior.

2.2 Phase $\mathbb{1}$: Instrumentation

Definition 2 (Instrumented Query, Phase $\mathbb{1}$). *Given a subject query q, its* instrumented variant *$q^{\mathbb{1}}$ computes the same*

[1] We use $a \times b$ to denote pair (or record) types and write $\{a\}$ for the type of tables with rows of type a.

```
max_scan(x, y, angle, max_angle) AS (
 SELECT t.r_rx AS x, t.r_ry AS y, t.a_angle AS angle,
        MAX(t.a_angle) OVER (w) AS max_angle
 FROM   (SELECT r.x1 AS r_x1, r.y1 AS r_y1,
                r.rx AS r_rx, r.ry AS r_ry,
                a.angle AS a_angle
         FROM   rays AS r, angles AS a
         WHERE  r.rx = a.x AND r.ry = a.y) AS t
 WINDOW w AS
   (PARTITION BY t.r_x1, t.r_y1
    ORDER BY sqrt((t.r_rx - 17)^2 + (t.r_ry - 10)^2))
)
```

Figure 5: Common table expression max_scan of the visibility query (Figure 3) after normalization. UDF dist has been inlined into the ORDER BY clause.

```
1  max_scan𝟙(ρ, x, y, angle, max_angle) AS (
2   SELECT  write_WIN(④, t.ρ, FIRST_VALUE(t.ρ) OVER (w),
3                          RANK() OVER (w)) AS ρ,
4           t.r_rx AS x, t.r_ry AS y, t.a_angle AS angle,       added
5           MAX(t.a_angle) OVER (w) AS max_angle
6   FROM    (SELECT write_JOIN2(③, r.ρ, a.ρ) AS ρ,
7                   r.x1 AS r_x1, r.y1 AS r_y1,
8                   r.rx AS r_rx, r.ry AS r_ry,
9                   a.angle AS a_angle
10           FROM   rays𝟙 AS r, angles𝟙 AS a
11           WHERE  r.rx = a.x AND r.ry = a.y) AS t        -- p𝟙(r,a)
12  WINDOW w AS
13    (PARTITION BY t.r_x1, t.r_y1                         -- f𝟙(t)
14     ORDER BY sqrt((t.r_rx - 17)^2 + (t.r_ry - 10)^2))   -- g𝟙(t)
15 )
```

Figure 6: Instrumented variant of CTE max_scan in Phase $\mathbb{1}$.

output table as q. Whenever q evaluates an expression of non-parametric type to make a relevant value-based decision, $q^{\mathbb{1}}$ logs the outcome of that decision as a side-effect of query evaluation.

The instrumentation of q will be *compositional*: q's overall instrumentation is assembled from the instrumentation of q's subqueries— the latter transformations do not interfere and may be performed in isolation. Here, we exploit this to save page space and focus on CTE fragment max_scan of the SQL query in Figure 3. Input to instrumentation is a normalized form of the original query in which individual operations (*e.g.*, joins, window functions, ordering) are placed in separate subqueries. The normalized CTE max_scan is shown in Figure 5. Normalization, discussed in Section 3, helps to devise compact sets of query transformation rules.

Figure 6 shows max_scan$^{\mathbb{1}}$, the instrumented form of max_scan. (For a query, expression, CTE, or table named n, we use n,

	map𝟙 x	map𝟙 y	map𝟙 alt	ρ	map𝟚 x	map𝟚 y	map𝟚 alt	
	0	0	400.0	$m_{(0,0)}$	$\{x_{(0,0)}\}$	$\{y_{(0,0)}\}$	$\{a_{(0,0)}\}$	
	0	1	400.0	$m_{(0,1)}$	$\{x_{(0,1)}\}$	$\{y_{(0,1)}\}$	$\{a_{(0,1)}\}$	
	0	2	400.0	$m_{(0,2)}$	$\{x_{(0,2)}\}$	$\{y_{(0,2)}\}$	$\{a_{(0,2)}\}$	
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
⊠	17	10	200.0	$m_{(17,10)}$	$\{x_{(17,10)}\}$	$\{y_{(17,10)}\}$	$\{a_{(17,10)}\}$	⊠
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
	20	18	0.0	$m_{(20,18)}$	$\{x_{(20,18)}\}$	$\{y_{(20,18)}\}$	$\{a_{(20,18)}\}$	
	20	19	0.0	$m_{(20,19)}$	$\{x_{(20,19)}\}$	$\{y_{(20,19)}\}$	$\{a_{(20,19)}\}$	
	20	20	0.0	$m_{(20,20)}$	$\{x_{(20,20)}\}$	$\{y_{(20,20)}\}$	$\{a_{(20,20)}\}$	

Figure 7: Table map in Phases $\mathbb{1}$ and $\mathbb{2}$. A row with key (x, y) = (x, y) is identified by row ID $\rho = m_{(x,y)}$.

```
max_scan𝟚(ρ, x, y, angle, max_angle) AS (
 SELECT  t.ρ AS ρ,
         t.r_rx AS rx, t.r_ry AS ry, t.a_angle AS angle,
         ⋃ t.a_angle OVER (w) ∪ ⋃ Y_win OVER (w) AS max_angle
 FROM
   (SELECT join.ρ,
           r.x1 ∪ Y_join AS r_x1, r.y1 ∪ Y_join AS r_y1,
           r.rx ∪ Y_join AS r_rx, r.ry ∪ Y_join AS r_ry,
           a.angle ∪ Y_join AS a_angle
    FROM   rays𝟚 AS r, angles𝟚 AS a,
           LATERAL read_JOIN2(③, r.ρ, a.ρ) AS join(ρ),
           LATERAL Y(r.rx ∪ a.x ∪ r.ry ∪ a.y) AS Y_join -- Y(p𝟚(r,a))
   ) AS t,
   LATERAL read_WIN(④, t.ρ) AS win(ρ,part,rank),
   LATERAL Y(t.r_x1 ∪ t.r_y1) ∪                          -- Y(f𝟚(t))
           Y(dist𝟚(∅, ∅, t.r_rx, t.r_ry)) AS Y_win        -- Y(g𝟚(t))
 WINDOW w AS (PARTITION BY win.part ORDER BY win.rank)
)
```

Figure 8: Interpreter for CTE max_scan in Phase $\mathbb{2}$.

$n^{\mathbb{1}}$, and $n^{\mathbb{2}}$ to refer to the original and its Phase $\mathbb{1}$/$\mathbb{2}$ variants.) Where the original query reads from table r, the instrumented version reads from $r^{\mathbb{1}}$ in which column ρ carries row identifiers—otherwise, r and $r^{\mathbb{1}}$ are identical. Indeed, r and $r^{\mathbb{1}}$ may denote the very same table if the underlying RDBMS externalizes row identity in some form (*e.g.*, through virtual column ctid in PostgreSQL or rowid in IBM Db2 and Oracle). Table map$^{\mathbb{1}}$ is depicted in Figure 7 on the left.

When we log the outcome v of a computation over a row r, we write the pair $(\mathtt{r}.\rho, v)$ to identify the row once we read the log back. It is the primary aim of instrumentation to insert calls to side-effecting functions $\mathit{write}_\square(Ⓒ, \mathtt{r}.\rho, v)$ that perform the required log writing. Parameter Ⓒ distinguishes the calls' locations in the instrumented SQL text such that one log may hold entries written by multiple call sites. Phase $\mathbb{2}$ (see below) then uses $\mathit{read}_\square(Ⓒ, \mathtt{r}.\rho)$ to obtain v again. The approach is indifferent to the actual realization of $\mathit{write}_\square$ and $\mathit{read}_\square$. Section 3 shows pseudo code and the appendix proposes a possible SQL-internal implementation of logging.

In the subquery in Lines 6 to 11 of Figure 6, the result of the join depends on the evaluations of predicate $p^{\mathbb{1}}(\mathtt{r}, \mathtt{a})$ = r.rx = a.x AND r.ry = a.y. We make the outcomes of $p^{\mathbb{1}}$ available to Phase $\mathbb{2}$ via calls to $\mathit{write}_{\mathtt{JOIN2}}(③, \mathtt{r}.\rho, \mathtt{a}.\rho)$ in Line 6. Note that we chose to not log $p^{\mathbb{1}}$'s actual Boolean value but, equivalently, the fact that rows r and a are join partners—this refinement saves us from logging the false outcomes of $p^{\mathbb{1}}$ and also simplifies Phase $\mathbb{2}$. The invocation of $\mathit{write}_{\mathtt{JOIN2}}$ performs log writing and then returns a newly generated row identifier that represents the joined row t.

In the window-based query enclosing the join, evaluation depends on the partitioning and ordering criteria that determine the placement of row t inside window w (Lines 13 and 14 in Figure 6). Both criteria are functions of t, namely $f^{\mathbb{1}}(t)$ = (t.r_x1, t.r_y1) and $g^{\mathbb{1}}(t)$ = sqrt((t.r_rx - 17)^2 + (t.r_ry - 10)^2). Phase $\mathbb{2}$ will not be able to evaluate either function once computation has shifted from column values to dependency sets. The invocation of $\mathit{write}_{\mathtt{WIN}}$ in Lines 2 to 3 thus writes the required log entries. Here, again, we do not log the values of $f^{\mathbb{1}}(t)$ and $g^{\mathbb{1}}(t)$ as is, but equivalently record FIRST_VALUE(t.ρ) OVER (w) and RANK() OVER (w): the former represents t's partition in terms of the identifier of that partition's first row, the latter gives t's position inside that partition. Once both criteria are logged, the $\mathit{write}_{\mathtt{WIN}}(④, t.\rho, \ldots)$ call returns $t.\rho$.

2.3 Phase $\mathbb{2}$: Interpretation

Definition 3 (Interpreter, Phase $\mathbb{2}$). Interpreter $q^{\mathbb{2}}$ *for instrumented query* $q^{\mathbb{1}}$ *exclusively manipulates dependency sets: if the evaluation of a subexpression* $e^{\mathbb{1}}$ *of* $q^{\mathbb{1}}$ *depended on the input table cells* $i_1, i_2, \ldots, i_n$*, its interpreted counterpart* $e^{\mathbb{2}}$ *in* $q^{\mathbb{2}}$ *evaluates to the dependency set* $\{i_1, i_2, \ldots, i_n\}$*.*

The definition implies that interpreter $q^{\mathbb{2}}$ reads and outputs tables of the same shape (cardinality and width) as instrumented query $q^{\mathbb{1}}$: where Phase $\mathbb{1}$ reads table $r^{\mathbb{1}}$, the interpreter reads $r^{\mathbb{2}}$ whose cells hold dependency sets (see table $\texttt{map}^{\mathbb{2}}$ in Figure 7 on the right). Note that corresponding rows in $r^{\mathbb{1}}$ and $r^{\mathbb{2}}$ share their identifiers ρ to establish a one-to-one correspondence between the cells of both tables.

Singleton dependency sets in source table cells indicate that each of these cells only depends on itself. In table $\texttt{map}^{\mathbb{2}}$, unique identifier $\texttt{x}_{(x,y)}$ represents the cell in column `x` of the row with $\rho = m_{(x,y)}$; likewise, $\texttt{y}_{(x,y)}$ and $\texttt{a}_{(x,y)}$ represent cells in columns `y` and `alt`, respectively. These cell identifiers are entirely abstract and never computed with (cf. with the *colors* of [5]).

The interpreter for CTE `max_scan` is shown in Figure 8. CTE $\texttt{max\_scan}^{\mathbb{2}}$ preserves the syntactic shape of $\texttt{max\_scan}^{\mathbb{1}}$ in Figure 6: a window-based aggregation consumes the result of the join between tables $\texttt{rays}^{\mathbb{2}}$ and $\texttt{angles}^{\mathbb{2}}$. Computation, however, is over dependency sets instead of values. Rather than committing early to one of many viable relational set representations [7, 28, 42], $\texttt{max\_scan}^{\mathbb{2}}$ uses the usual operators $\cup/\bigcup$ where these sets are combined/aggregated.

Following the above definition, the non-standard interpretation of functions $p^{\mathbb{1}}$, $f^{\mathbb{1}}$, $g^{\mathbb{1}}$ yields variants $\_^{\mathbb{2}}$ collecting the dependencies for those columns that influence the functions' evaluation (cf. with Section 2.2):

$$
\begin{aligned}
p^{\mathbb{2}}(\texttt{r},\texttt{a}) &= \texttt{r.rx} \cup \texttt{a.x} \cup \texttt{r.ry} \cup \texttt{a.y} \\
f^{\mathbb{2}}(t) &= t\,\texttt{.r\_x1} \cup t\,\texttt{.r\_y1} \\
g^{\mathbb{2}}(t) &= t\,\texttt{.r\_rx} \cup \varnothing \cup t\,\texttt{.r\_ry} \cup \varnothing \ .
\end{aligned}
$$

As described in Section 2.1, these functions exclusively manipulate dependency sets of type $\mathbb{P}$. The literals `17` and `10` map to $\varnothing$ in $g^{\mathbb{2}}$ since both do not depend on any input data whatsoever. Set aggregate $\bigcup t\texttt{.a\_angle OVER (}w\texttt{)}$ in Line 4 interprets $\texttt{MAX(}t\texttt{.a\_angle) OVER (}w\texttt{)}$ in $\texttt{max\_scan}^{\mathbb{1}}$: according to the SQL semantics, all t`.a_angle` values inside current window w are aggregated to evaluate the `MAX` window function [43, § 4.16.3] and thus influence the function's result.

The interpreter uses $Y(D)$ to indicate that dependency set D contains cells describing *why*-provenance instead of the default *where*-provenance. We construct the *why*-dependency set $Y(p^{\mathbb{2}}(\texttt{r},\texttt{a}))$ in Line 12 to reflect that predicate p inspects exactly these cells to decide whether rows `r` and `a` are join partners. (We use `LATERAL` to bind this set to Y_{join} as it is referenced multiple times later on.) Likewise, we form Y_{win} in Lines 15 and 16 to collect the cells $Y(f^{\mathbb{2}}(t)) \cup Y(g^{\mathbb{2}}(t))$ that are inspected to decide how window frames are formed. Line 4 then adds these *why*-dependencies to the provenance of the `MAX` window aggregate.

$\texttt{max\_scan}^{\mathbb{2}}$ reads the logs written in Phase $\mathbb{1}$ to (1) reenact $p^{\mathbb{1}}$'s filtering decisions and (2) to reconstruct the window frames formed by $f^{\mathbb{1}}$ and $g^{\mathbb{1}}$. Iff $\mathit{read}_{\texttt{JOIN2}}(③,\texttt{r}.\rho,\texttt{a}.\rho)$ returns a join row identifier, rows `r` and `a` have been found to partner in Phase $\mathbb{1}$. $\mathit{read}_{\texttt{WIN}}(④,t.\rho)$ retrieves partition representative win`.part` and in-partition position win`.rank` to enable the `WINDOW` clause to place row t inside its proper frame.

output

	x	y	visible?
	⋮	⋮	⋮
❶	$\{\texttt{x}_{(2,12)}\}$	$\{\texttt{y}_{(2,12)}\}$	$\left\{\begin{array}{c}\texttt{x}_{(2,12)},\ \texttt{y}_{(2,12)},\ \texttt{a}_{(2,12)},\\ \texttt{x}_{(3,12)},\ \texttt{y}_{(3,12)},\ \texttt{a}_{(3,12)},\\ \cdots\\ \texttt{x}_{(15,10)},\texttt{y}_{(15,10)},\texttt{a}_{(15,10)},\\ \texttt{x}_{(16,10)},\texttt{y}_{(16,10)},\ \texttt{a}_{(16,10)}\end{array}\right\}$
	⋮	⋮	⋮
❷	$\{\texttt{x}_{(3,4)}\}$	$\{\texttt{y}_{(3,4)}\}$	$\left\{\begin{array}{c}\texttt{x}_{(3,4)},\ \texttt{y}_{(3,4)},\ \texttt{a}_{(3,4)},\\ \texttt{x}_{(4,5)},\ \texttt{y}_{(4,5)},\ \texttt{a}_{(4,5)},\\ \cdots\\ \texttt{x}_{(15,9)},\ \texttt{y}_{(15,9)},\texttt{a}_{(15,9)},\\ \texttt{x}_{(16,10)},\texttt{y}_{(16,10)},\texttt{a}_{(16,10)}\end{array}\right\}$
	⋮	⋮	⋮

Figure 9: *Where*-provenance of the visibility of spots ❶ and ❷ (see Figure 4) as derived by interpretation in Phase $\mathbb{2}$.

Output. Interpretation for the visibility query of Figure 3 yields the dependency set table of Figure 9. For a spot in the terrain located at (x, y), we learn that its coordinates have been copied over from input table `map` (the cells in column `x` solely depend on $\texttt{x}_{(x,y)}$; likewise for column `y`). Spot visibility, however, depends on the terrain's altitude along the ray from (x, y) to ⊠. Indeed, Figure 4 simply is a visualization of the dependency sets found in column `visible?` of table `output` in Figure 9.

3. INTERPRETING SQL IN SQL

The query instrumentation of Phase $\mathbb{1}$ and the construction of the interpreter of Phase $\mathbb{2}$ are based on a pair of rule-based SQL source transformations. We first *normalize* the input query to facilitate transformation rules that do not face large monolithic `SELECT` blocks but may focus on a single SQL clause at a time.

Definition 4 (Normalized Query). *All* `SELECT` *blocks in the* normalized query *for subject query* q *adhere to the syntactic form shown in Figure 10. Normalization preserves the semantics of* q*.*

Normalization of the input query rests on the following two cornerstones:

Explicitness. Expand the column list implicit in `SELECT *`. In `SELECT` clauses, name expressions e explicitly (e `AS` c). In `FROM` clauses, introduce explicit row aliases for tables or subqueries q (q `AS` t). In expressions, use qualified column references ($t.c$) only. Expand `DISTINCT` into `DISTINCT ON`. Trade inline window specifications for explicit `WINDOW` clauses. Inline the bodies of non-recursive UDFs (like `dist` of Figure 3). Remove syntactic sugar to reduce query diversity, *e.g.*, supply empty `GROUP BY` criteria $g \equiv$ `()`, or make defaults like `OFFSET 0` and `LIMIT ALL` explicit, should any of these be missing.

Clause isolation. Traverse the query syntax tree bottom up. Inside a `SELECT` block, isolate its SQL clauses by placing each clause inside a separate subquery. This leads to "onion-style" uncorrelated nesting in the `FROM` clause, cf. the sketch of the resulting normal form in Figure 10. On completion, transformation rules like WINDOW or GROUP (see Figure 12, discussed below) may assume that they encounter single-table `FROM` q `AS` t clauses only.

```
SELECT ···
FROM  (SELECT DISTINCT ON(···) ···
       FROM  (SELECT ··· AGG(···) OVER(w ϕ) ···
              FROM  (SELECT ··· AGG(···) ···
                     FROM  (SELECT ···
                            FROM  q,...,q
                            WHERE p) AS t
                     GROUP BY g
                     HAVING p) AS t
              WINDOW w AS (···)) AS t
       ORDER BY o) AS t
ORDER BY o
OFFSET n
LIMIT n
```

Figure 10: Syntactic shape of a normalized SELECT block after SQL clause isolation. Any but the innermost layer of the "onion" may be missing.

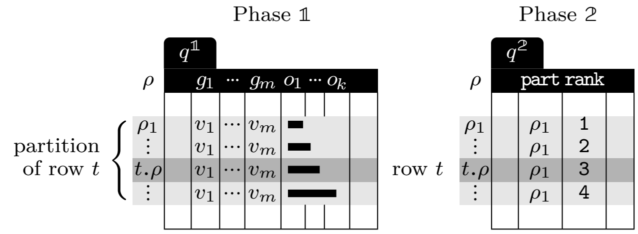


Figure 11: Placement of row t in a windowed table with clause WINDOW w AS (PARTITION BY $g_1,\ldots,g_m$ ORDER BY $o_1,\ldots,o_k$). All rows in t's partition agree on FIRST_VALUE($t.\rho$) OVER (w) $= \rho_1$. In its partition, t ranks 3rd (bars ▬ picture the ordering criteria). Pair $(\rho_1, 3)$ thus exactly pinpoints t's placement in Phase 𝟚.

As an example, see Figure 5 where the WINDOW clause has been isolated from the join of rays and angles. Normalization preserves query semantics as well as data provenance. This holds, in particular, for clause isolation: from inner to outer, the onion's layers adhere to the evaluation order defined for SQL clauses in a SELECT block [43, § 7.5ff].

Definition 5 (Syntactic Transformation $\Rrightarrow$). *Given a normalized subject query q, the syntax-directed mapping*

$$q \mapsto \langle q^{𝟙}, q^{𝟚} \rangle$$

derives both q's instrumented variant $q^{𝟙}$ and interpreter $q^{𝟚}$. Mapping $\mapsto$ is collectively defined by the inferences rules of Figure 12 and the appendix.

The synchronized derivation allows $q^{𝟙}$ and $q^{𝟚}$ to readily share information about call sites Ⓒ when we place a call to *write*$_\square$(Ⓒ,...) in $q^{𝟙}$ and its associated *read*$_\square$(Ⓒ,...) call in $q^{𝟚}$. (The inference rules invoke Ⓒ $=$ *site*() to obtain arbitrary yet fresh call site identifiers Ⓒ.)

Figure 12 displays a representative subset of the complete rule set. Taken jointly with the additions of the appendix, the rules cover the rich SQL dialect characterized in the introduction and can translate the visibility query of Figure 3 as well as the 22 queries of the TPC-H benchmark (see Section 4 below). In the rules' antecedents, we use $|q_i \mapsto \langle\cdot,\cdot\rangle|_{i=1,\ldots,n}$ to indicate that all (sub)queries $q_1, \ldots, q_n$ are to be transformed.

Mapping $\mapsto$ proceeds bottom-up and first establishes trivial interpreters for SQL's syntactic leaf constructs. No logging is required in these cases. Rule Lit: A literal l represents itself: its interpreter thus returns the empty set $\varnothing$ of input data dependencies. Rule Col: In Phase 𝟚, a column reference $t.c$ holds a set of cell identifiers that represents $t.c$'s data dependencies (see Definition 3). The rule thus simply returns this set. Rule Table ensures that Phase 𝟙 operates over regular base data held in the cells of $table^{𝟙}$ while Phase 𝟚 reads (singleton) dependency sets from $table^{𝟚}$ that represent these cells (cf. Figure 7).

Non-leaf rules first invoke $\mapsto$ on constituent queries and assemble the results to form composite instrumentations and interpreters. Rule Builtin manifests that the evaluation of a built-in SQL operator $\oplus$ (returning a single scalar, row, or array value) depends on all of its n arguments e_i. The interpreter thus unions the arguments' dependency sets $e_i^{𝟚}$. Rule With invokes $\mapsto$ recursively on the common table expression q_i but otherwise preserves the syntactic shape of the input query (Section 2.1). The rule does, however, extend the schemata of all CTEs to expose new column ρ whose row identifiers help to relate the results of Phases 𝟙 and 𝟚 (again, see Figure 7). Shape preservation in Rule With, specifically, presents the opportunity to use SQL's WITH to assign a name, say t, to any intermediate query result of interest. After interpretation, table $t^{𝟚}$ will hold the *where*- and *why*-provenance of the intermediate result. The ability to inspect such intermediate provenance (as computed by common table expression max_scan^{𝟚}, for example, see Figure 8) can be instrumental in the analysis and debugging of very complex queries.

Rule Join infers the instrumentation and interpreter for m-fold joins. Such joins (or its simpler variants, see the appendix) form the innermost layer of the onion. All other SQL clauses of the current SELECT block are placed in enclosing layers.

As discussed in Section 2.2, instrumented query $i^{𝟙}$ invokes *write*$_{\text{JOIN}\langle m\rangle}$ to record which combinations of rows satisfied join predicate p and to obtain a new row identifier ρ that represents the joined row—otherwise, the input query and $i^{𝟙}$ perform the same computation. Interpreter $i^{𝟚}$ re-enacts the join based on the log and *read*$_{\text{JOIN}\langle m\rangle}$ as described in Section 2.3. Since, in the input query, the evaluation of p determined the inclusion of a joined row with its columns $c_1, \ldots, c_n$, we collect $e_i^{𝟚} \cup Y(p^{𝟚})$ to form the full *where*- and *why*-provenance for column c_i.

Rule Group instruments GROUP BY queries to collect the row identifiers of the current group (via the set aggregate $\bigcup\{t.\rho\}$). *write*$_{\text{GRP}}$ logs the resulting row identifier set along with a unique group identifier ρ. When Phase 𝟚 processes row t, it invokes *read*$_{\text{GRP}}$(Ⓒ,$t.\rho$) to retrieve the identifier *group*.ρ of t's group as a stand-in grouping criterion. The interpreter thus faithfully re-enacts the grouping performed in Phase 𝟙. In Phase 𝟚, Rule Agg turns a value-based aggregate $AGG(e^{𝟙})$ into a set aggregate $\bigcup e^{𝟚}$ that collects the dependencies of all evaluations of its argument e (this models SQL's aggregate semantics [43, § 4.16.4]). To this *where*-provenance, Rule Group adds the *why*-provenance $\bigcup Y_{group}$ to reflect (1) that the criteria g_i jointly determined which group a row belongs to and (2) that HAVING predicate p decided the group's inclusion in the result.

The rows of a windowed table are partitioned and then ordered before a window—or: frame—of rows is formed around each input row t [43, § 4.15.14]. Rule Window thus injects a call to *write*$_{\text{WIN}}$ that logs the identifier of t's partition as well as the row's intra-partition position (Figure 11 illustrates). Later, the interpreter reads the pair back (cf.

$$\frac{}{l \Mapsto \langle l, \varnothing \rangle}\ (\textsc{Lit}) \qquad \frac{}{t.c \Mapsto \langle t.c, t.c \rangle}\ (\textsc{Col}) \qquad \frac{}{\mathit{table} \Mapsto \langle \mathit{table}^{\mathbb{1}}, \mathit{table}^{2} \rangle}\ (\textsc{Table})$$

$$\frac{\begin{array}{c}\oplus \in \{\cdot\texttt{+}\cdot, \cdot\texttt{<=}\cdot, \texttt{ROW}(\cdot,\dots,\cdot), \cdot\ \texttt{IN}\ (\cdot,\dots,\cdot), \dots\} \\ \big| e_i \Mapsto \langle e_i^{\mathbb{1}}, e_i^{2} \rangle \big|_{i=1\dots n}\end{array}}{\oplus(e_1,\dots,e_n) \Mapsto \langle \oplus(e_1^{\mathbb{1}},\dots,e_n^{\mathbb{1}}), e_1^{2} \cup \cdots \cup e_n^{2} \rangle}\ (\textsc{Builtin})$$

$$\frac{\big| q_i \Mapsto \langle q_i^{\mathbb{1}}, q_i^{2} \rangle \big|_{i=0\dots n} \quad i^{\mathbb{1}} = \begin{array}{l}\texttt{WITH [RECURSIVE]}\ t_1^{\mathbb{1}}(\rho, c_{11},\dots,c_{1k_1})\ \texttt{AS}\ (q_1^{\mathbb{1}}), \dots, \\ \qquad t_n^{\mathbb{1}}(\rho, c_{n1},\dots,c_{nk_n})\ \texttt{AS}\ (q_n^{\mathbb{1}}) \\ \qquad q_0^{\mathbb{1}}\end{array} \quad i^{2} = \begin{array}{l}\texttt{WITH [RECURSIVE]}\ t_1^{2}(\rho, c_{11},\dots,c_{1k_1})\ \texttt{AS}\ (q_1^{2}), \dots, \\ \qquad t_n^{2}(\rho, c_{n1},\dots,c_{nk_n})\ \texttt{AS}\ (q_n^{2}) \\ \qquad q_0^{2}\end{array}}{\begin{array}{l}\texttt{WITH [RECURSIVE]}\ t_1(c_{11},\dots,c_{1k_1})\ \texttt{AS}\ (q_1), \dots, \\ \qquad t_n(c_{n1},\dots,c_{nk_n})\ \texttt{AS}\ (q_n) \\ \qquad q_0\end{array} \Mapsto \langle i^{\mathbb{1}}, i^{2} \rangle}\ (\textsc{With})$$

$$\frac{\begin{array}{c} m \geqslant 1 \quad \big| e_i \Mapsto \langle e_i^{\mathbb{1}}, e_i^{2} \rangle \big|_{i=1\dots n} \quad \big| q_i \Mapsto \langle q_i^{\mathbb{1}}, q_i^{2} \rangle \big|_{i=1\dots m} \quad p \Mapsto \langle p^{\mathbb{1}}, p^{2} \rangle \quad © = \mathit{site}() \\[1ex] i^{\mathbb{1}} = \begin{array}{ll}\texttt{SELECT} & \mathit{write}_{\texttt{JOIN}\langle m\rangle}(©, t_1.\rho,\dots,t_m.\rho)\ \texttt{AS}\ \rho, \\ & e_1^{\mathbb{1}}\ \texttt{AS}\ c_1,\dots,e_n^{\mathbb{1}}\ \texttt{AS}\ c_n \\ \texttt{FROM} & q_1^{\mathbb{1}}\ \texttt{AS}\ t_1,\dots,\texttt{[LATERAL]}\ q_m^{\mathbb{1}}\ \texttt{AS}\ t_m \\ \texttt{WHERE} & p^{\mathbb{1}}\end{array} \quad i^{2} = \begin{array}{ll}\texttt{SELECT} & \mathit{join}.\rho\ \texttt{AS}\ \rho,\ e_1^{2} \cup Y_{\mathit{join}}\ \texttt{AS}\ c_1,\dots,e_n^{2} \cup Y_{\mathit{join}}\ \texttt{AS}\ c_n \\ \texttt{FROM} & q_1^{2}\ \texttt{AS}\ t_1,\dots,\texttt{[LATERAL]}\ q_m^{2}\ \texttt{AS}\ t_m, \\ & \texttt{LATERAL}\ \mathit{read}_{\texttt{JOIN}\langle m\rangle}(©, \langle t_1.\rho,\dots,t_m.\rho\rangle)\ \texttt{AS}\ \mathit{join}(\rho), \\ & \texttt{LATERAL}\ Y(p^{2})\ \texttt{AS}\ Y_{\mathit{join}}\end{array}\end{array}}{\begin{array}{ll}\texttt{SELECT} & e_1\ \texttt{AS}\ c_1,\dots,e_n\ \texttt{AS}\ c_n \\ \texttt{FROM} & q_1\ \texttt{AS}\ t_1,\dots,\texttt{[LATERAL]}\ q_m\ \texttt{AS}\ t_m \\ \texttt{WHERE} & p\end{array} \Mapsto \langle i^{\mathbb{1}}, i^{2} \rangle}\ (\textsc{Join})$$

$$\frac{\begin{array}{c} q \Mapsto \langle q^{\mathbb{1}}, q^{2} \rangle \quad p \Mapsto \langle p^{\mathbb{1}}, p^{2} \rangle \quad \big| e_i \Mapsto \langle e_i^{\mathbb{1}}, e_i^{2} \rangle \big|_{i=1\dots n} \quad \big| g_i \Mapsto \langle g_i^{\mathbb{1}}, g_i^{2} \rangle \big|_{i=1\dots m} \quad © = \mathit{site}() \\[1ex] i^{\mathbb{1}} = \begin{array}{ll}\texttt{SELECT} & \mathit{write}_{\texttt{GRP}}(©, \bigcup\{t.\rho\})\ \texttt{AS}\ \rho, \\ & e_1^{\mathbb{1}}\ \texttt{AS}\ c_1,\dots,e_n^{\mathbb{1}}\ \texttt{AS}\ c_n \\ \texttt{FROM} & q^{\mathbb{1}}\ \texttt{AS}\ t \\ \texttt{GROUP BY} & g_1^{\mathbb{1}},\dots,g_m^{\mathbb{1}} \\ \texttt{HAVING} & p^{\mathbb{1}}\end{array} \quad i^{2} = \begin{array}{ll}\texttt{SELECT} & \mathit{group}.\rho\ \texttt{AS}\ \rho,\ e_1^{2} \cup \bigcup Y_{\mathit{group}}\ \texttt{AS}\ c_1,\dots,e_n^{2} \cup \bigcup Y_{\mathit{group}}\ \texttt{AS}\ c_n \\ \texttt{FROM} & q^{2}\ \texttt{AS}\ t, \\ & \texttt{LATERAL}\ \mathit{read}_{\texttt{GRP}}(©, t.\rho)\ \texttt{AS}\ \mathit{group}(\rho), \\ & \texttt{LATERAL}\ Y(g_1^{2} \cup \cdots \cup g_m^{2} \cup p^{2})\ \texttt{AS}\ Y_{\mathit{group}} \\ \texttt{GROUP BY} & \mathit{group}.\rho\end{array}\end{array}}{\begin{array}{ll}\texttt{SELECT} & e_1\ \texttt{AS}\ c_1,\dots,e_n\ \texttt{AS}\ c_n \\ \texttt{FROM} & q\ \texttt{AS}\ t \\ \texttt{GROUP BY} & g_1,\dots,g_m \\ \texttt{HAVING} & p\end{array} \Mapsto \langle i^{\mathbb{1}}, i^{2} \rangle}\ (\textsc{Group})$$

$$\frac{e \Mapsto \langle e^{\mathbb{1}}, e^{2} \rangle}{\mathit{AGG}(e) \Mapsto \langle \mathit{AGG}(e^{\mathbb{1}}), \bigcup e^{2} \rangle}\ (\textsc{Agg}) \qquad \frac{\mathit{AGG}(e) \Mapsto \langle a^{\mathbb{1}}, a^{2} \rangle \quad Y = \bigcup Y_{\mathit{win}}\ \texttt{OVER}\ (w\ \phi)}{\mathit{AGG}(e)\ \texttt{OVER}\ (w\ \phi) \Mapsto \langle a^{\mathbb{1}}\ \texttt{OVER}\ (w\ \phi), a^{2}\ \texttt{OVER}\ (w\ \phi) \cup Y \rangle}\ (\textsc{AggWin})$$

$$\frac{\begin{array}{c} q \Mapsto \langle q^{\mathbb{1}}, q^{2} \rangle \quad \big| e_i \Mapsto \langle e_i^{\mathbb{1}}, e_i^{2} \rangle \big|_{i=1\dots n} \quad \big| g_i \Mapsto \langle g_i^{\mathbb{1}}, g_i^{2} \rangle \big|_{i=1\dots m} \quad \big| o_i \Mapsto \langle o_i^{\mathbb{1}}, o_i^{2} \rangle \big|_{i=1\dots k} \quad © = \mathit{site}() \\[1ex] i^{\mathbb{1}} = \begin{array}{l}\texttt{SELECT}\ \mathit{write}_{\texttt{WIN}}(©, t.\rho, \texttt{FIRST\_VALUE}(t.\rho)\ \texttt{OVER}\ (w), \\ \qquad\qquad \texttt{RANK()}\ \texttt{OVER}\ (w))\ \texttt{AS}\ \rho, \\ \qquad e_1^{\mathbb{1}}\ \texttt{AS}\ c_1,\dots,e_n^{\mathbb{1}}\ \texttt{AS}\ c_n \\ \texttt{FROM}\ q^{\mathbb{1}}\ \texttt{AS}\ t \\ \texttt{WINDOW}\ w\ \texttt{AS (PARTITION BY}\ g_1^{\mathbb{1}},\dots,g_m^{\mathbb{1}} \\ \qquad\qquad \texttt{ORDER BY}\ o_1^{\mathbb{1}}\ \mathit{dir}_1,\dots,o_k^{\mathbb{1}}\ \mathit{dir}_k)\end{array} \quad i^{2} = \begin{array}{l}\texttt{SELECT}\ t.\rho\ \texttt{AS}\ \rho,\ e_1^{2}\ \texttt{AS}\ c_1,\dots,e_n^{2}\ \texttt{AS}\ c_n \\ \texttt{FROM}\ q^{2}\ \texttt{AS}\ t, \\ \qquad \texttt{LATERAL}\ \mathit{read}_{\texttt{WIN}}(©, t.\rho)\ \texttt{AS}\ \mathit{win}(\texttt{part},\texttt{rank}), \\ \qquad \texttt{LATERAL}\ Y(g_1^{2} \cup \cdots \cup g_m^{2} \cup o_1^{2} \cup \cdots \cup o_k^{2})\ \texttt{AS}\ Y_{\mathit{win}} \\ \texttt{WINDOW}\ w\ \texttt{AS (PARTITION BY}\ \mathit{win}.\texttt{part} \\ \qquad\qquad \texttt{ORDER BY}\ \mathit{win}.\texttt{rank})\end{array}\end{array}}{\begin{array}{l}\texttt{SELECT}\ e_1\ \texttt{AS}\ c_1,\dots,e_n\ \texttt{AS}\ c_n \\ \texttt{FROM}\ q\ \texttt{AS}\ t \\ \texttt{WINDOW}\ w\ \texttt{AS (PARTITION BY}\ g_1,\dots,g_m \\ \qquad\qquad \texttt{ORDER BY}\ o_1\ \mathit{dir}_1,\dots,o_k\ \mathit{dir}_k)\end{array} \Mapsto \langle i^{\mathbb{1}}, i^{2} \rangle}\ (\textsc{Window})$$

Figure 12: Excerpt of inference rules $q \Mapsto \langle i^{\mathbb{1}}, i^{2} \rangle$ that derive instrumented query $i^{\mathbb{1}}$ and interpeter i^{2} from input query q.

win.`part` and *win*.`rank` in i^{2}) to correctly place t among its peers. Since the interpretation of windowed aggregates preserves their original frame clause ϕ (see Rule AggWin), Phase 2 builds *where*-provenance from exactly those dependency sets found in the frame around row t.[2] Again, this coincides with the SQL window aggregate semantics [43, § 4.16.3]. Much like in the `GROUP BY` case, the rules add *why*-provenance based on the partitioning and ordering criteria g_i and o_j, respectively (these are collected in Y_{win} and then added in Rule AggWin).

[2]The `max_scan` CTE of Figure 3 omits the default frame clause $\phi \equiv$ `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. Rules AggWin and Window work for arbitrary frames.

Rule characteristics. The mapping rules for $\Mapsto$ discussed here exhibit general properties that are characteristic for the full rule set:

- *Why*-provenance may be optionally derived in addition to *where*-provenance. If we omit the lighter subexpressions in the definitions of the i^{2}, interpretation will compute *where*-provenance only. Since the *why*-provenance of an output cell can be substantial (*e.g.*, in Rules Agg and AggWin, the rows of an entire group or window frame contribute their dependency sets), we can expect significant time and space savings if we skip the derivation of *why*-dependencies.
- During interpretation, provenance sets grow monotonically (once found, dependencies are never thrown away). This

```
write_JOIN⟨m⟩(ⓒ,ρ1,...,ρm):                          read_JOIN⟨m⟩(ⓒ,⟨ρ1,...,ρm⟩):
  {ρ} ← get_JOIN(ⓒ, ⟨ρ1,..., ρm⟩)                      return get_JOIN(ⓒ, ⟨ρ1,..., ρm⟩)
  if {ρ} = ∅ then
  ⌊ ρ ← put_JOIN(ⓒ, ⟨ρ1,..., ρm⟩, row())
  return ρ

write_GRP(ⓒ,{ρ1,..., ρn}):                           read_GRP(ⓒ,ρ):
  {ρ} ← get_GRP(ⓒ, {ρ1,..., ρn})                       return get_GRP(ⓒ, {..., ρ,...})
  if {ρ} = ∅ then                                                    ︸
  ⌊ ρ ← put_GRP(ⓒ, {ρ1,..., ρn}, row())                        match sets P
  return ρ                                                      with ρ ∈ P

write_WIN(ⓒ,ρ,ρ_part,rank):                           read_WIN(ⓒ,ρ):
  if get_WIN(ⓒ, ρ) = ∅ then                            return get_WIN(ⓒ, ρ)
  ⌊ put_WIN(ⓒ, ρ, ⟨ρ_part, rank⟩)
  return ρ
```

Figure 13: Pseudo code: *write*$_\square$/*read*$_\square$ function pairs for log writing and reading, $\square \in \{\texttt{JOIN}\langle m\rangle, \texttt{GRP}, \texttt{WIN}\}$.

helps to devise a simple and efficient internal representation of provenance sets.

3.1 Log Writing and Reading

In the absence of concrete values, the interpreters consult logs (via *read*$_\square$ calls) to re-enact relevant value-based computations performed in Phase $\mathbb{1}$. Pseudo code for three *read*$_\square$ functions and their *write*$_\square$ pendants is shown in Figure 13. These functions invoke lower-level routines *put*$_\square$ and *get*$_\square$ that write to and read from log file *log*$_\square$. The log files may be realized in various forms, *e.g.*, in terms of operating system files or indexed relational tables. Below, we discuss the details of logging but abstract from any particular implementation. The appendix shows concrete log contents for a set of sample tables and queries and also elaborates on a purely relational, indexed encoding of log files. The SQL-based implementation described there has been used in the upcoming Section 4.

Lower-level logging routines are:

- *put*$_\square$(ⓒ$, k, e)$: add record $\langle$ⓒ$, k, e\rangle$ to file *log*$_\square$, then return entry e.
- *get*$_\square$(ⓒ$, k)$: from *log*$_\square$, return the set of e found in records $\langle$ⓒ$, k, e\rangle$. Return $\varnothing$ if there are no matching records.
- *row*(): generate and return a new unique row identifier.

A record $\langle$ⓒ$, \langle\rho_1, \dots, \rho_m\rangle, \rho\rangle$ in file *log*$_{\texttt{JOIN}}$ indicates that a `FROM` clause at site ⓒ joined m rows $\rho_1, \dots, \rho_m$ to yield a new row ρ. Function *write*$_{\texttt{JOIN}\langle m\rangle}$ ensures that this fact is recorded *once* in the log: only if a join of $\rho_1, \dots, \rho_m$ at ⓒ has not been encountered before (*i.e.*, *get*$_{\texttt{JOIN}}$(ⓒ$, \langle\rho_1, \dots, \rho_m\rangle) = \varnothing$), a log entry with a fresh ρ is made. Phase $\mathbb{1}$ may attempt such repeated identical writes to *log*$_{\texttt{JOIN}}$ if site ⓒ is located inside a subquery which the query optimizer decided to evaluate more than once (this may happen in the TPC-H benchmark, for example, see Section 4). In such scenarios, *write*$_{\texttt{JOIN}\langle m\rangle}$ makes sure that its side-effect on *log*$_{\texttt{JOIN}}$ is not carried out repeatedly. This *write once* safeguard also ensures that *read*$_{\texttt{JOIN}\langle m\rangle}$(ⓒ$, \langle\rho_1, \dots, \rho_m\rangle)$ will either yield a set of 0 or 1 row identifiers—recall that the interpreter $i^{\mathbb{2}}$ of Rule JOIN uses this behavior to properly re-enact the join semantics. Analogous remarks apply to *write*$_{\texttt{GRP}}$/*read*$_{\texttt{GRP}}$ and *write*$_{\texttt{WIN}}$/*read*$_{\texttt{WIN}}$.

4. THE PROVENANCE TAX

Provenance derivation processes substantially more data than the value-based computation it explains. First, we trade the value-based query q for the pair $\langle q^{\mathbb{1}}, q^{\mathbb{2}}\rangle$: effectively, the subject query is executed twice. Second, we expect that

$q^{\mathbb{1}}$ **is costly:** The two queries communicate via log files. Log file writes in $q^{\mathbb{1}}$ lead to additional data movement and incur side effects that may constrain the query optimizer.

$q^{\mathbb{2}}$ **is costly:** Where q outputs 1NF cell values, $q^{\mathbb{2}}$ returns entire sets of dependencies. These dependency sets may be large, *e.g.*, if q invokes aggregate functions.

This section aims to quantify how high this "provenance tax" indeed is and how it correlates with general query characteristics. On the way, we demonstrate how variations of the provenance granularity, dependency set representation, and an awareness of the properties of set operations can lead to significant runtime improvements.

The experiments below derive the full *where*- and *why*-provenance for all 22 queries of the TPC-H benchmark [46]. Here, we set the benchmark's scale factor to 1, *i.e.*, table `lineitem` holds about 6 000 000 rows. A repetition of the experiments at TPC-H scale factor 10 shows how the approach scales with growing database instances: see the appendix which reports slowdowns and speed-ups nearly identical to those observed in the discussion below.

All queries execute on a PostgreSQL 9.5 engine hosted on a Linux (kernel 4.4) machine with two 4-core Intel Xeon 5570 CPUs, 70 GB of RAM, and harddisk-based secondary storage. We report the average performance of five runs with best and worst execution times ignored. Instead of absolute wall clock times we focus on the slowdown—or speed-up—we observe once we switch from value to dependency set computation. In all plots below, a slowdown of $\times 1$ represents the evaluation time of the original TPC-H queries (no provenance derived). Queries *Q1* to *Q22* are displayed across the horizontal axes; the plots are thus best read column by column.

The (non-)impact of normalization. Figure 14 summarizes the impact of the individual phases of provenance derivation. The "onion-style" query normalization (Figure 10) does not alter the semantics and—on its own—also appears to preserve query performance (see the points ▫ cluster around $\times 1$). We have found RDBMSs to successfully remove the simple uncorrelated nesting in the `FROM` clause and generate plans identical to those for the original TPC-H queries. For *Q9*, the explicit onion nesting leads PostgreSQL to aggregate first and sort later which even beats the system's usual planning in which these operations are swapped. (Out of curiosity, we also fed the 22 original and normalized queries into HyPer [35] with its advanced query unnesting procedure [37] and found no plan differences at all.)

An analysis of the experiments reveals four major subject query characteristics that influence the overhead of provenance derivation in Phases $\mathbb{1}$ and $\mathbb{2}$. Figure 15 shows these query categories and how the 22 TPC-H queries fit in. We discuss this categorization below, phase by phase.

4.1 Phase $\mathbb{1}$: Impact of Logging

Relative to the original TPC-H queries, we observe a geometric mean slowdown of 3.4 in Phase $\mathbb{1}$ (see ▪ in Figure 14). The gaps ▫▪ are a measure of the logging effort that the instrumented queries invest. The log sizes and call site counts in ◯ at the bottom of Figure 14 show that, on average, a TPC-H query contains 2.5 *write*$_\square$ call sites that log just below 24 MB of data if we use the tabular representation of logs described in the appendix.

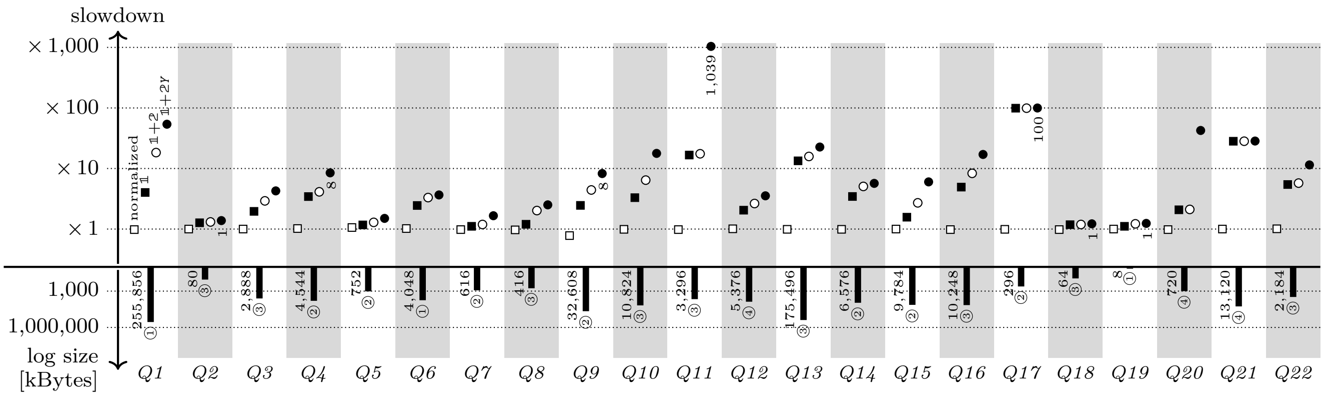


Figure 14: Normalization (□), Phase 𝟙 (■), Phases 𝟙+𝟚 (○ without/● with *why*-provenance) relative to value-based TPC-H.

	Query Characteristic	Yes (High Overhead)	No (Low Overhead)
𝟙	non-selective? (high # of log writes)	*Q1*, *Q13*	*Q2*, *Q5*, *Q7*, *Q17*, *Q18*, *Q19*
𝟙	correlation? (repeated side effects)	*Q4*, *Q11*, *Q17*, *Q21*, *Q22*	*Q1*, *Q5*, *Q6*, *Q7*, *Q12*, *Q18*
𝟚	high dependency set cardinality?	*Q1*, *Q9*, *Q11*, *Q13*, *Q15*, *Q16*, *Q20*, *Q22*	*Q2*, *Q3*, *Q10*, *Q17*, *Q19*
𝟚	expensive *why*-provenance?	*Q1*, *Q11*, *Q20*	*Q5*, *Q6*, *Q8*, *Q14*, *Q18*, *Q19*, *Q21*

Figure 15: Query characteristics that influence provenance overhead in Phases 𝟙 and 𝟚 (all TPC-H queries categorized).

Selectivity and logging overhead. Selective filters and joins reduce data *as well as* log volume (recall the placement of $write_{\text{JOIN}\langle m\rangle}$ in Rule Join). Queries *Q2*, *Q18*, *Q19* show this most clearly and only induce negligible overhead in Phase 𝟙. The opposite holds for *Q1* (whose non-selective predicates let almost all 6 000 000 rows of `lineitem` pass) and *Q13* (in which a left outer join requires the logging of the identifiers of both qualifying and non-qualifying pairs of rows). Both queries log substantial data volumes and exhibit large Phase 𝟙 overhead.

Correlation and side-effecting log writes. $write_{\square}$ call sites located in a subquery that the RDBMS fails to decorrelate and unnest (a problem that already occurs with the original TPC-H query [4]) trigger the functions' guard against writing identical log entries repeatedly, see Section 3.1. In our implementation, this increases the cost of $write_{\square}$ to about 0.14 ms per call. Queries *Q4*, *Q11*, *Q17*, *Q21*, and *Q22* contain such correlated subqueries in their `WHERE` and `HAVING` clauses and show the Phase 𝟙 cost of avoiding these unwanted side effects.

Logging without side effects? *Tupling* [30] suggests a functionally pure implementation alternative in which a row is extended by *an extra column* that holds its associated log entry: instead of issuing a side-effecting log write, $write_{\text{JOIN}\langle 2\rangle}$ constructs and returns a pair $(\rho, \langle\rho_1, \rho_2\rangle)$ to indicate that rows ρ_1 and ρ_2 were joined to form a new row ρ, for example. For *Q17* and *Q21*, this shows a promising runtime improvement of factor 76 and 28 in Phase 𝟙, respectively. However, tupling complicates the treatment of constructs like scalar or `IN` subqueries which, effectively, now need to be executed twice (once yielding the original, once the extended rows). Queries like *Q18* thus are penalized. Tupling bears a promising performance advantage in Phase 𝟙 but we would

(1) lose fully compositional query transformation (the use of tupling would be conditioned on the absence of the mentioned query constructs) and
(2) sacrifice query shape preservation (Section 2.1) and ultimately face the same problems as *Perm* and *GProM* (Section 4.4).

We consider the conditional use of tupling an interesting item of future work.

4.2 Phase 𝟚: Computing with Dependency Sets

We derive provenance through the composition of Phases 𝟙 and 𝟚. Measurement ○ in Figure 14 thus reflects the *overall slowdown if both phases are executed in sequence.* We find a mean slowdown of factor 4.6 (visualized by the □○ gaps) compared to value-based query evaluation.

Dependency set cardinality. Where a value-based query manipulates an 1NF cell value v, its interpreter will construct v's—possibly large—dependency set: if we consider the entire TPC-H benchmark and form the mean, we find that each output data cell depends on about 10 000 input cells. When a single cell holds an aggregate of a group (or window) of rows, its dependency set cardinality directly reflects the group (or window) size, see Rule Agg. Foremost, this affects *Q1* and its eight aggregates, one of which (column `sum_charge`) yields a *where*-dependency set of about 4 500 000 elements per output cell. As an aggregation-heavy OLAP benchmark, TPC-H generally constitutes a challenging workload in this respect (see Figure 15).

Expensive *why*-provenance. Recall that we can selectively enable the derivation of *why*-provenance in Phase 𝟚. If we do, we experience larger overall overheads as marked by the points ● in Figure 14, with a mean overall slowdown of factor 9.0. While the logs encode the outcome of a predicate p, this does not suffice to derive *why*-provenance: we now also need to interpret p (*i.e.*, evaluate $p^{𝟚}$) to learn which input items influenced p's value. For *Q11*, in particular, this requires the interpretation of a complex `HAVING` p clause where p contains a three-way join and aggregation. *Q1* now additionally derives how aggregates depend on grouping critera (see subexpressions g_i in Rule Group), doubling `sum_charge`'s dependency cardinality to 9 000 000 input cells per output cell.

Dependency set representation. Given these substantial dependency cardinalities, it is expected that Phase 𝟚 can bene-

$\mathtt{r}^{\mathbb{1}}$ a	b	c	ρ	$\mathtt{r}^{\mathbb{2}}$ a	b	c
1	10	a	ρ_1	$\{a_1\}$	$\{b_1\}$	$\{c_1\}$
1	20	b	ρ_2	$\{a_2\}$	$\{b_2\}$	$\{c_2\}$
1	30	c	ρ_3	$\{a_3\}$	$\{b_3\}$	$\{c_3\}$
2	40	d	ρ_4	$\{a_4\}$	$\{b_4\}$	$\{c_4\}$
2	50	e	ρ_5	$\{a_5\}$	$\{b_5\}$	$\{c_5\}$

(a) Table r in Phases $\mathbb{1}$+$\mathbb{2}$.

```
SELECT r.a,
       SUM(r.b) AS sum
FROM   r
GROUP BY r.a
```

$\mathtt{output}^{\mathbb{1}}$

ρ	a	sum
ρ_6	1	60
ρ_7	2	90

(b) Query and result of Phase $\mathbb{1}$.

$\mathtt{output}^{\mathbb{2}}$

ρ	a	sum
ρ_6	$\{a_1, a_1, a_2, a_3\}$	$\{b_1, b_2, b_3, a_1, a_2, a_3\}$
ρ_7	$\{a_4, a_4, a_5\}$	$\{b_4, b_5, a_4, a_5\}$

(c) Cell-level provenance.

$\mathtt{r}^{\mathbb{2}}$

ρ	prov
ρ_1	$\{\rho_1\}$
ρ_2	$\{\rho_2\}$
ρ_3	$\{\rho_3\}$
ρ_4	$\{\rho_4\}$
ρ_5	$\{\rho_5\}$

$\mathtt{output}^{\mathbb{2}}$

ρ	prov
ρ_6	$\{\rho_1, \rho_2, \rho_3\}$
ρ_7	$\{\rho_4, \rho_5\}$

(d) Row-level provenance.

Figure 16: Provenance derivation at cell and row granularities for a simple `GROUP BY` query.

fit from efficient set representations. The appendix indeed makes this observation if we replace the PostgreSQL-native set encoding based on type `int[]` by bit sets [7].

Beneficial effects of logging. Logging incurs overhead in Phase $\mathbb{1}$, but Phase $\mathbb{2}$ can benefit from the effort. To exemplify, in the original *Q19*, a join between `lineitem` and `part` accounts for 98% of the execution time. In the interpreted *Q19*, table log_{JOIN2} acts much like a join index or access support relation [34, 47] from which the row identifiers of the join partners are read off directly. As a result, interpretation is about 10× faster than value-based evaluation. The situation is similar for *Q21*, where log_{JOIN4} assumes the role of a join index for an expensive four-way join. Additionally, the interpreter saves the evaluation effort for two complex `[NOT] EXISTS(···)` subqueries: the identifier of the row that constitutes the existential quantifier's *why*-provenance is simply read off the log tables. Access support relations that materialize provenance relationships between rows have shown very similar beneficial effects in [33].

4.3 Switching From Cell to Row Granularity

The present approach derives provenance at the granularity of *individual table cells*: each output cell is assigned the set of input cells that influenced its value. We obtain highly detailed insight into input-output data dependencies but surely pay a price in terms of interpreter overhead and size of the resulting provenance. It turns out that this level of granularity is not firmly baked into the method. We can straightforwardly adapt it to operate at the less detailed *row* level which suffices for many uses and also is the granularity provided by the majority of existing work [11, 13, 15, 20, 25]. Below, we contrast both granularity levels, sketch how row-level interpretation can be realized, and assess the resulting performance advantage.

For the cell granularity case, consider 5-row input table `r` whose Phase $\mathbb{1}$ and $\mathbb{2}$ variants are shown in Figure 16(a). In $\mathtt{r}^{\mathbb{2}}$, each cell is assigned a singleton dependency set (cf. Figure 7). If we use the `GROUP BY` query of Figure 16(b) as the subject query, Phase $\mathbb{1}$ yields the $\mathtt{output}^{\mathbb{1}}$ table shown in the same figure. Phase $\mathbb{2}$ preserves the shape of the output but returns a table whose cells hold dependency sets (Figure 16(c)). Cell identifier shades indicate the provenance kind (*where*, *why*): to arrive at the aggregate value `90` of row ρ_7, the query had to sum the input cells b_4, b_5 (holding `40`, `50`) and decide group membership based on cells a_4, a_5 (both holding `2`).

If we switch to row granularity, Phase $\mathbb{1}$ remains unchanged. Phase $\mathbb{2}$ entirely abstracts from the input's columns and thus assigns one singleton identifier set per row, see the modified two-column version of $\mathtt{r}^{\mathbb{2}}$ in Figure 16(d). A simplified interpreter (discussed below) tracks row dependencies and finally emits the $\mathtt{output}^{\mathbb{2}}$ table in Figure 16(d). We learn that

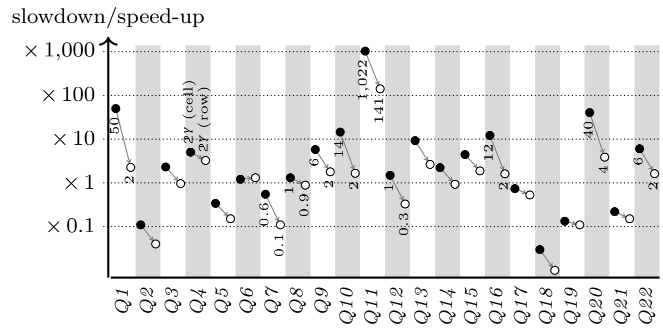


Figure 17: Deriving cell-level (•) vs. row-level provenance (∘).

aggregate value `60` of output row ρ_6 depends on input rows ρ_1, ρ_2, ρ_3, *i.e.*, exactly those rows of table `r` that constitute the group in which column `a = 1`.

The rules of Figure 12 adapt to row granularity in a systematic fashion. As mentioned, the definitions of the instrumented queries $i^{\mathbb{1}}$ remain as is. Where the cell-level interpreters $i^{\mathbb{2}}$ track dependencies column by column, the new row-level interpreters collect dependencies held in the single `prov` column. In Rule JOIN, $i^{\mathbb{2}}$ is now defined as

$$i^{\mathbb{2}} = \begin{array}{ll} \texttt{SELECT} & join.\rho, t_1.\texttt{prov} \cup \cdots \cup t_m.\texttt{prov} \;\texttt{AS prov} \\ \texttt{FROM} & q_1^{\mathbb{2}} \;\texttt{AS}\; t_1, \ldots, [\texttt{LATERAL}]\; q_m^{\mathbb{2}} \;\texttt{AS}\; t_m, \\ & \texttt{LATERAL}\; read_{\text{JOIN}\langle m\rangle}(©, \langle t_1.\rho, \ldots, t_m.\rho\rangle) \;\texttt{AS}\; join(\rho) \end{array}.$$

At row granularity level, we process narrow two-column tables (columns ρ, `prov`) regardless of the width of the input and output tables. Also, compared to the cell-level variant, the interpreter evaluates fewer $\cup/\bigcup$ operations that build smaller dependency sets: in TPC-H, one output row has about 2 500 dependencies on input rows (mean across the benchmark). Figure 17 documents how the interpretation overhead drops by an order of magnitude once we switch from cell- (•) to row-level (∘) dependencies.

4.4 A Comparison with Perm and GProM

The computation of row-level dependencies also paves the way for a direct comparison with *Perm* [20–22, 24], a long-running research effort that makes a genuine attempt at provenance derivation for SQL. In *Perm*, input queries are translated into a multiset algebra, rewritten and augmented for provenance computation, and then translated back into SQL for execution on PostgreSQL. Unlike the present work, *Perm* opts for an invasive approach and adds code that sits between the query rewriter and planner of PostgreSQL 8.3. To any output row o, *Perm* attaches all columns of those input rows that influence o's computation (influence contribution semantics [21])—if o has n

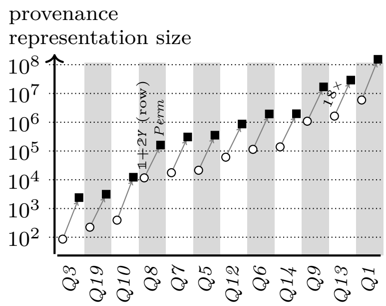


Figure 19: Size of provenance representation: dependency sets (○) vs. *Perm* (■).

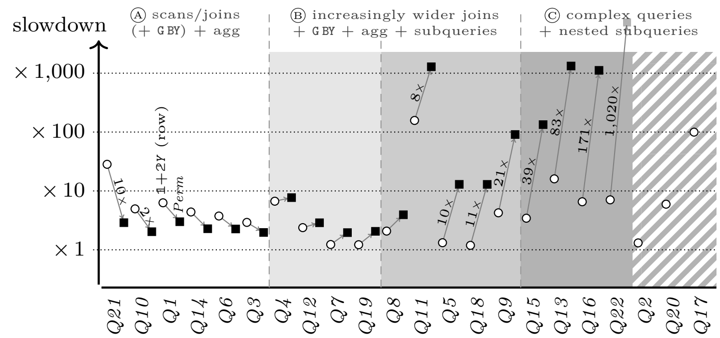


Figure 20: Head-to-head: interpretation at row granularity (○) and *Perm* (■).

influencing rows, o is repeated n times in the result. For table `r` and the `GROUP BY` query of Figures 16(a) and (b), *Perm* thus emits the table of Figure 18. Row $(\mathtt{a}, \mathtt{sum}) = (1, 60)$, for example, is contained three times as it depends on all input rows with $\mathtt{a} = 1$. Recall that row-level SQL interpretation represents the same provenance information in the $\mathtt{output}^{2}$ table of Figure 16(d). In practice, the resulting redundancy can be significant, as Figure 19 illustrates: across the TPC-H benchmark queries, *Perm*'s normalized representation of provenance consistently requires more space than dependency sets (we measured a mean factor of 19).

output

	a	sum	r.a	r.b	r.c
result	1	60	1	10	a
	2	90	2	40	d
provenance	1	60	1	20	b
	1	60	1	30	c
	2	90	2	50	e

provenance

Figure 18: *Perm*'s fully normalized representation of result and provenance ( ) for the `GROUP BY` query of Figure 16(b).

Row-level interpretation vs. *Perm*. These space considerations and our earlier observations about query characteristics (Sections 4.1 and 4.2) are also reflected in Figure 20. In this head-to-head slowdown comparison of row-level SQL interpretation (○, including Phases 𝟙 and 𝟚) and *Perm* (■), a trend ○↗■ indicates that interpretation showed less slowdown than *Perm*. Over all executable queries, row-level interpretation levies a provenance tax of factor 5.1 while *Perm* imposes a factor of 18.9 (geometric means).

Figure 20 shows that the advantage of interpretation over *Perm* increases with query complexity. *Perm*'s log-free approach pays off in category Ⓐ of scans or simple joins and (grouped) aggregation. The price of writing a large log and correlation in Phase 𝟙 (see Section 4.1) makes *Q21* the only complex TPC-H query for which *Perm* outperforms interpretation. As discussed above, tupling for *Q21* could tip the scales in favor of interpretation, though. The queries in Ⓑ are characterized by increasing predicate complexity and join width, with the latter contributing to the discussed space overhead: *Perm* generates queries that emit wide rows (of 62 instead of the original 2 columns for TPC-H query *Q8*, for example). Some queries in Ⓑ and *all* queries in Ⓒ feature nested subqueries, which amplify *Perm*'s provenance representation size problem:[3] a row of the outer query is replicated n times if the subquery emits n rows—even if the subquery is existentially quantified [23]. Three queries in category Ⓒ that *Perm* failed to process within 4 hours are marked ▨ in Figure 20.

GProM-style provenance-aware query optimization. After rewriting for provenance, *Perm* has been found to generate query shapes that significantly deviate from the original subject query. Plans take on a form that challenges existing query processors or may lead to the duplication of work (*e.g.*, see *Perm*'s `GROUP BY` translation rule **R5** in [21]). These observations led to follow-up work on successor project *GProM* that identifies specific algebraic optimizations tuned to cope with challenging query structure [2, 39, 40]. These provenance-aware optimizations primarily target grouping and aggregation and, for some queries, can offer a speed-up of up to factor 3 (personal communication with the author and [40]). With these—partially heuristic, partially cost-based—algebraic rewrites, *GProM* reaches even deeper into the underlying RDBMS than *Perm*.

However, provenance-specific optimizations also apply to the interpretation of SQL. The principle can be adapted to
- match our provenance model (dependency sets),
- be non-invasive, *i.e.*, not reach inside the RDBMS kernel,
- be easily expressible on the SQL language level, *i.e.*, in terms of a shape-preserving source-level transformation.

One particular transformation relates to the occurrence of a closed (non-correlated) subquery q_1 under an aggregrate. In Phase 𝟚 we have:

$$\bigcup_{r\in\mathbf{r}^{2}} \big(q_1 \cup q_2(r)\big) \quad\equiv\quad q_1 \cup \bigcup_{r\in\mathbf{r}^{2}} q_2(r) \quad , r \text{ not free in } q_1.$$

Note that this rewrite is specific for set aggregation and would be incorrect in a subject query that uses `SUM`/`+`, for example. In TPC-H, such constellations arise for *Q11*, *Q16*, *Q18*, *Q20*, and *Q22*, where the transformation reduces interpretation time in Phase 𝟚 by factors between 2 and 160 (the latter for *Q22*). The experiments of this section have been performed with the transformation enabled.

5. MORE RELATED WORK

The traced evaluation of subject queries is a defining feature of the present work. Phase 𝟙 identifies rows that *actually participated* in query evaluation; Phase 𝟚 adds cell-level dependencies and aggregates the Phase 𝟙 findings as needed. This places the approach in the landscape of established provenance notions.

To form *where*-provenance, we collect those input cells that were *used to compute the value* of an output cell—this includes those input cells that were copied verbatim and

[3]We thus omitted these queries from Figure 19—for *Q22* and its two scalar and existentially quantified subqueries, *Perm* incurs a representation size overhead of factor 25 000.

thus generalizes the notion of *where*-provenance as defined by Buneman in [6]. *Perm* [21] argues for and implements the same generalization. If we combine the *where*- and *why*-provenance derived by interpretation, for each output cell o we obtain *one* set of input cells that witness o. This provides a cell-level analog to *lineage* [15] or *influence contribution* [20, 21], concepts orignally established at the coarser row level. In deviation from Buneman's definition of *why*-provenance, we do not derive all possible witnesses but the particular set of input cells that were indeed used by the system to produce o. This is invaluable in declarative query debugging where such database size reductions can help to prevent users from "drowning in a sea of observations" [16]. Let us note that non-standard interpretation is a member of the *annotation propagation* family of approaches [5] which fail to derive provenance in the presence of empty intermediate results [8].

The shift from values to computation over dependency sets $\mathbb{P}$ relates to the *provenance semiring* that derives lineage for the positive bag algebra by Green *et al.* [25]. In a nutshell, Phase $\mathbb{2}$ realizes a SQL semantics interpreted in the particular semiring $(\mathbb{P}, \bot, \varnothing, \cup_L, \cup_S)$,[4] in which rows are annotated with dependency sets. To illustrate, in the treatment of $\sigma_{\mathbf{P}}(R)$ in [25], rows t that fail to satisfy predicate $\mathbf{P}$ are mapped to $\bot$ which effectively discards t's provenance contribution (case **selection** of Definition 3.2 in [25]). In the present work, this role of $\mathbf{P}$ is assumed by the `LATERAL` join with function $read_{\mathtt{JOIN}\langle m \rangle}$ which discards t if $t.\rho$ cannot be found in the associated log (see the redefinition of interpreter $i^{\mathbb{2}}$ of Rule JOIN in Section 4.3, set $m = 1$ to obtain a direct correspondence with [25]).

We understand provenance derivation as dynamic data dependency analysis and share this view with Cheney *et al.* [10,11]. The interpreters defined in the rules of Figure 12 propagate and accumulate dependency sets much like the *provenance tracking semantics* defined in Figures 5 and 6 of [10]. The authors state that "*[d]ynamic provenance may be [expensive to compute and] non-trivial to implement in a standard relational database system.*" Our present effort addresses just this challenge.

Given a piece o of the output, *backward slicing* [9, 45, 50] finds those parts—or: slices—of a program that are involved in producing o. In [36,41], we demonstrated the derivation of provenance through the application of slicing to imperative programs that simulate the semantics of SQL queries. In the present work, instead, we directly realize a dynamic variant of slicing for SQL but are only interested in input data slices on which o's value depends. If, however, we associate identifiers with SQL subexpressions (instead of cells), interpretation could instead identify the subject query slices relevant to the computation of o. This paves the way for a notion of *how*-provenance [12] whose findings directly relate to SQL's surface syntax (instead of algebraic plans, say).

C. Barry Jay has explored the decomposition of data structures into their *shape* and contained values [31, 32]. We have deliberately designed a two-phase approach that preserves data shape (original input and output tables share row width and cardinality with those of Phases $\mathbb{1}/\mathbb{2}$, respectively) and query shape (recall the discussion of parametricity of Section 2.1). We reap the benefits in terms of a straightforward, extensible formulation of inference rules and plans that do not swamp the DBMS's optimizer and executor. This focus on shape preservation tells this work apart from related efforts where data and its provenance are tightly bundled and then threaded jointly through the computation [5,10,11,18,21,49]. This reshapes input, intermediate, and output data as well as the computation process itself—sometimes dramatically so—and ultimately leads to restrictions on what data and query sizes are considered tractable [11, 39]. Bundling has the advantage that queries may post-process data and its provenance together, however. We can offer this integrated view through a join of the Phase $\mathbb{1}$ and $\mathbb{2}$ outputs: consider $\mathtt{output}^{\mathbb{1}} \bowtie_{\rho} \mathtt{output}^{\mathbb{2}}$ in Figure 16, for example.

[4]See [12, Sections 1.3 and 5.1] for the definitions of $\cup_L$, $\cup_S$ and their interaction with $\bot$.

6. WRAP-UP

The desire to move complex computation—like tasks in machine learning or graph processing—close to their data sources led to a steep growth in query complexity. As this trend will only continue, this work is an attempt to develop provenance derivation for SQL that catches up and helps to explain the resulting intricate queries. We shift from value- to dependency-based computation through a non-standard interpretation of the SQL semantics that can derive provenance at either the cell level or the coarser row granularity. The approach embraces a rich dialect of SQL constructs—including recursion, windowed aggregates, or table-valued and user-defined functions—and relies on a two-phase evaluation process designed to not overwhelm the underlying database system.

This work is extensible in several dimensions. We believe that the idea of non-standard interpretation does not break if further SQL constructs are added to the dialect. Currently, we explore the treatment of SQL DML statements (`INSERT`, `UPDATE`, `DELETE`) and functions defined in PL/SQL—this is also related to recent work on the re-enactment of transactions [38]. Further, the provenance model realized by the approach is subject to tuning. Phase $\mathbb{1}$, for example, may employ "lazy" or "greedy" variants of `EXISTS` to decide whether the provenance of a subquery includes one particular row or all rows that satisfied the quantifier (see [22] for a discussion of possible semantics).

We pursue optimizations that can help to boost Phase $\mathbb{1}$. Data flow analysis can reveal inclusion relationships between log files and thus render $write_{\square}$ at some call sites obsolete. Likewise, we can statically infer particular *write once* safeguards (Section 3.1) to be superfluous.

Lastly, the "onion-style" normalization of SQL has helped to keep the inference rule set of Figure 12 orthogonal and compact. We conjecture that this syntactic normal form can generally benefit efforts that rely on a source-level analysis and transformation of SQL. We will follow up in an independent thread of work.

Acknowledgments. We thank Boris Glavic for insightful discussions about provenance derivation for SQL and his help in configuring and operating *Perm*. Noah Doersing wrote a PostgreSQL extension that integrates *roaring bitmaps* into the system. This research is supported by the German Research Council (DFG) under grant no. GR 2036/4-1. We thank the anonymous reviewers for their observations and suggestions which greatly helped to improve this paper.

7. REFERENCES

[1] M. Armbrust, R. Xin, C. Lian, Y. Huai, D. Liu, J. Bradley, X. Meng, T. Kaftan, M. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proc. SIGMOD*, New York, USA, 2015.

[2] S. Bahareh, S. Feng, B. Glavic, S. Lee, X. Niu, and Q. Zeng. GProM—A Swiss Army Knife for Your Provenance Needs. *IEEE Data Engineering Bulletin*, 41(1), 2018.

[3] O. Benjelloun, A. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In *Proc. VLDB*, Seoul, Korea, 2006.

[4] P. Boncz, T. Neumann, and O. Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Proc. TPCTC*, Riva del Garda, Italy, 2013.

[5] P. Buneman, J. Cheney, and S. Vansummeren. On the Expressiveness of Implicit Provenance in Query and Update Languages. *ACM TODS*, 33(4), 2008.

[6] P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A Characterization of Data Provenance. In *Proc. ICDT*, London, UK, 2001.

[7] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better Bitmap Performance with Roaring Bitmaps. *Software: Practice and Experience*, 46(11), 2016.

[8] A. Chapman and H. Jagadish. Why Not? In *Proc. SIGMOD*, Providence, RI, USA, 2009.

[9] J. Cheney. Program Slicing and Data Provenance. *IEEE Data Engineering Bulletin*, 30(4), 2007.

[10] J. Cheney, A. Ahmed, and U. Acar. Provenance as Dependency Analysis. *Mathematical Structures in Computer Science*, 21(6), 2011.

[11] J. Cheney, A. Ahmed, and U. Acar. Database Queries that Explain their Work. In *Proc. PPDP*, 2014.

[12] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4), 2007.

[13] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe. Explaining Outputs in Modern Data Analytics. *PVLDB*, 9(12):1137–1148, 2016.

[14] Y. Cui and J. Widom. Lineage Tracing for General Data Warehouse Transformations. In *Proc. VLDB*, Rome, Italy, 2001.

[15] Y. Cui, J. Widom, and J. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM TODS*, 25(2), 2000.

[16] B. Dietrich, T. Müller, and T. Grust. The Best Bang for Your Bu(ck)g—When SQL Debugging and Data Provenance Go Hand in Hand. In *Proc. EDBT*, Bordeaux, France, 2016.

[17] J. Fan, A. Gerald, S. Raj, and J. Patel. The Case Against Specialized Graph Analytics Engines. In *Proc. CIDR*, Asilomar, CA, USA, 2015.

[18] S. Fehrenbach and J. Cheney. Language-Integrated Provenance. *Science of Computer Programming*, 2017.

[19] J. Freire, P. Bonnet, and D. Shasha. Computational Reproducibility: State-of-the-Art, Challenges, and Database Research Opportunities. In *Proc. ACM SIGMOD*, Scottsdale, AZ, USA, 2012.

[20] B. Glavic. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, University of Zürich, Switzerland, 2010.

[21] B. Glavic and G. Alonso. Perm: Processing Provenance and Data on the Same Data Model Through Query Rewriting. In *Proc. ICDE*, Shanghai, China, 2009.

[22] B. Glavic and G. Alonso. Provenance for Nested Subqueries. In *Proc. EDBT*, Saint Petersburg, Russia, 2009.

[23] B. Glavic and G. Alonso. Provenance for Nested Subqueries. In *Proc. EDBT*, Saint Petersburg, Russia, 2009.

[24] B. Glavic, R. Miller, and G. Alonso. Using SQL for Efficient Generation and Querying of Provenance Information. *Lecture Notes in Computer Science*, 8000, 2013.

[25] T. Green, G. Karvounarakis, and V. Tannen. Provenance Semirings. In *Proc. PODS*, Beijing, China, 2007.

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

[27] J. Hellerstein, C. Ré, F. Schoppmann, D. Wang, E. Fratkin, A. Gorajek, K. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[28] S. Helmer and G. Moerkotte. A Performance Study of Four Index Structures for Set-Valued Attributes of Low Cardinality. *VLDB Journal*, 12(3), 2003.

[29] M. Herschel, R. Diestelkämper, and H. Lahmar. A Survey on Provenance: What For? What Form? What From? *VLDB Journal*, 26(6), 2017.

[30] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling Calculation Eliminates Multiples Data Traversals. In *Proc. ICFP*, 1997.

[31] C. Jay. Shape in Computing. *ACM Computing Surveys*, 28(2), 1996.

[32] C. Jay and R. Cockett. Shapely Types and Shape Polymorphism. In *Proc. ESOP*, Edinburgh, UK, 1994.

[33] G. Karvounarakis, Z. Ives, and V. Tannen. Querying Data Provenance. In *Proc. SIGMOD*, Indianapolis, USA, 2010.

[34] A. Kemper and G. Moerkotte. Access Support Relations: An Indexing Method for Object Bases. *Information Systems*, 17(2), 1992.

[35] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP/OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proc. ICDE*, Hannover, Germany, 2011.

[36] T. Müller and T. Grust. Provenance for SQL Based on Abstract Interpretation: Value-less, but Worthwhile. *PVLDB*, 8(12):1872–1875, 2015.

[37] T. Neumann and A. Kemper. Unnesting Arbitrary Queries. In *Proc. BTW*, Hamburg, Germany, 2015.

[38] X. Niu, B. Arab, S. Lee, F. Su, X. Zou, D. Gawlick, V. Krishnaswamy, Z. Liu, and B. Glavic. Debugging Transactions and Tracking their Provenance with Reenactment. *PVLDB*, 10(12):1857–1860, 2017.

[39] X. Niu, R. Kapoor, B. Glavic, D. Gawlick, Z. Liu, V. Krishnaswamy, and V. Radhakrishnan.

Provenance-Aware Query Optimization. In *Proc. ICDE*, San Diego, CA, USA, 2017.

[40] X. Niu, R. Kapoor, B. Glavic, D. Gawlick, Z. Liu, V. Krishnaswamy, and V. Radhakrishnan. Heuristic and Cost-Based Optimization for Diverse Provenance Tasks. *IEEE TKDE*, 2018.

[41] D. O'Grady, T. Müller, and T. Grust. How 'How' Explains What 'What' Computes—How-Provenance for SQL and Query Compilers. In *Proc. TaPP*, London, UK, 2018.

[42] K. Ramasamy, J. Patel, J. Naughton, and R. Kaushik. Set Containment Joins: The Good, the Bad, and the Ugly. In *Proc. VLDB*, Cairo, Egypt, 2000.

[43] *Database Languages–SQL–Part 2: Foundation.* ISO/IEC 9075-2:2016.

[44] W.-C. Tan. Provenance in Databases: Past, Present, and Future. *IEEE Data Engineering Bulletin*, 32(4), 2007.

[45] F. Tip. A Survey of Program Slicing Techniques. Technical report, CWI, Amsterdam, The Netherlands, 1994.

[46] *The TPC Benchmark H.* `tpc.org/tpch`.

[47] P. Valduriez. Join Indices. *ACM TODS*, 12(2), 1987.

[48] P. Wadler. Theorems for Free! In *Proc. ICFP*, London, UK, 1989.

[49] Y. Wang and S. Madnick. A Polygen Model for Heterogeneous Database Systems: The Source Tagging Perspective. In *Proc. VLDB*, Brisbane, Australia, 1990.

[50] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4), 1984.