Data Provenance for Recursive SQL Queries

Benjamin Dietrich  Tobias Müller  Torsten Grust
University of Tübingen
Tübingen, Germany
[b.dietrich,to.mueller,torsten.grust]@uni-tuebingen.de

ABSTRACT
The adoption of recursion in SQL—framed either in terms of recursive common table expressions (CTEs) or recursive user-defined functions (UDFs)—marked a jump in the expressivity of the query language. The resulting queries can perform complex computation close to database-resident data but, at the same time, often prove challenging to understand and debug. We build on earlier work on the derivation of where- and why-provenance for complex (yet non-recursive) SQL queries to also embrace recursive SQL CTEs and UDFs. Fine-grained data provenance for recursive SQL is derived through language-level query rewriting and a two-phase evaluation strategy that does not invade the underlying RDBMS.

CCS CONCEPTS
• Information systems → Structured Query Language; • Software and its engineering → Recursion; • Theory of computation → Data provenance.

KEYWORDS
SQL, recursion, CTEs, UDFs, data provenance, query debugging

1 INTRODUCTION
"Rekursiv geht meistens schief" (loosely translated: recursion typically goes awry) is a rhyming slogan widely known in German software development communities. While witty, this catchline expresses that it can indeed be challenging to fully wrap one’s head around the progress of a recursive computation. What are the intermediate states? Will this base case be reached eventually?

Since the advent of recursion in SQL—either in the form of SQL:1999’s recursive common table expressions (CTEs, WITH RECURSIVE) [7] or self-referential user-defined functions (recursive UDFs, as supported by PostgreSQL, for example [14])—these challenges also surface in the context of relational query languages:

1. The semantics of a recursive CTE (see Figure 1) is defined in terms of a fixpoint computation in which a query \( q_0(T) \) is iteratively evaluated over its most recent result table \( T \) until no (previously unseen) rows are produced. The intermediate results of \( q_T \) are appended to form the overall result of the CTE. To fully understand query progress, developers would need to keep track of the states of the working, intermediate, and union tables that implement this iteration [8].

2. A recursive SQL UDF may invoke itself, possibly at multiple call sites (non-linear recursion) deeply embedded into a SQL query that post-processes the results once the recursive calls return (non-tail recursion). Here, keeping track requires an understanding of the UDF call stack and its frames that hold the UDF’s query context and current arguments, for example.

Should such queries indeed go awry, the developer’s debugging toolbox is rather empty. Recursive CTEs are not easily instrumented without destructive effects on their monotonicity or termination properties. Likewise, a recursive UDF would need to be rewritten to return information about intermediate states along with the actual function result. Such manual query instrumentation may affect or hide existing bugs and introduce entirely new ones.

Data provenance for recursive SQL queries. We propose to bank on data provenance as one tool that can provide insight into recursive query computation. Here, we focus on where- and why-provenance [3, 12] derived at the level of individual table cells. We aim to derive

- **where-provenance** which identifies those table cells that were **copied or transformed** to compute the next intermediate (or final) state of a recursive query, and
- **why-provenance** to locate those cells that were **inspected to decide whether a particular value is part of the output at all**.

In tandem, both provenance kinds paint a complete picture of which table cells guided the recursive computation or were sourced to construct the overall result. Below, we study recursive CTEs as well as UDFs to demonstrate how such provenance information may help to explain unexpected results or visualize relevant input cells.

The derivation of provenance for (very) complex queries has proven to be notoriously difficult [3]. Yet, to render provenance useful and practical for SQL query debugging, the derivation strategy is required to embrace constructs that constitute a potential hurdle for query authoring and understanding. This certainly includes constructs like subqueries (including correlation), grouping and aggregation, window functions, complex types (like row values or arrays), or scalar and table-valued built-in and user-defined functions.
The present work continues our earlier effort to derive where- and why-provenance for a SQL dialect that admits all language constructs listed above. To this we add the ability to process recursive CTEs and recursive SQL UDFs. To analyze SQL query \( q \), we continue to pursue a two-phase evaluation process in which

- **Phase 1** evaluates a variant query \( q^1 \) that processes the original input tables while it also writes a protocol about the outcome of predicate evaluation, before
- **Phase 2** reads the interim protocol to evaluate variant \( q^2 \) which entirely operates over \( \text{sets of dependencies} \) between table cells. Where- and why-provenance for \( q \) may then be read off the resulting dependency sets.

The variant queries \( q^1 \) and \( q^2 \) are derived from \( q \) through systematic rewriting. Like \( q \), both variants are regular SQL queries that can be evaluated on top of off-the-shelf RDBMSs. No kernel-level changes are called for.

We recapitulate the two phases when we turn to the sample recursive queries below (Sections 2 and 3). For a full review of the approach we refer to [12]. We have laid out the present paper as a “companion” to this earlier work—in particular, here we provide an addendum of inference rules for SQL query rewriting that form a coherent whole with the rules found in the companion paper [12]. Two-stage evaluation and the processing of—potentially large—dependency sets has an impact on query evaluation time. We quantify the slowdown in Section 3.2 but will also show how Phase 2 can peruse the mentioned interim protocol to actually perform significantly better than regular query evaluation. A review of related efforts is found in Section 4.

## 2 UNRAVELING RECURSIVE CTEs

When a SQL query yields unexpected results, provenance helps to zoom in on the relevant input data items and (potentially buggy) query portions. We have made this observation for non-recursive queries in [12]—here we extend it to recursive CTEs.

Let us focus on CTE \( \text{bom} \) of Figure 2 which recursively computes the bill of materials (or: parts explosion) of a humanoid robot. (We have adapted this query from the PostgreSQL manual [14]—the manual contains just the bug we discuss here.) Robot parts, along with their sub-parts and required quantities, are held in input table \text{parts} (see Figure 3(a)). The CTE yields the output table of Figure 3(a) which lists the quantities (column \text{qty}) of all parts (column \text{sub_part}) required to assemble the robot. The numbers of required fingers and feet (5 and 1, respectively) certainly look suspicious: we had expected 18 and 2. To understand how CTE \text{bom} arrived at these questionable results, we explore the provenance of output value 5 (\( \text{(5)} \)) in Figure 3(a).

The why-provenance of \( \text{(5)} \) reveals all input table cells that have been inspected to decide whether \( \text{(5)} \) occurs in the output table. We find the highlighted input cells \text{arm}, \text{body}, and \text{humanoid} which the CTE has used to recursively descend into the part hierarchy encoded by table \text{parts} (see predicate \text{p.part} = \text{b.sub_part} in Line 9 of Figure 2). The chain humanoid→body→arm describes the expected path from root part humanoid to sub-part finger: the recursive traversal expressed by CTE \text{bom} appears to be in order.

The where-provenance of output \( \text{(5)} \), however, is shown to only refer to the quantity \( \text{(5)} \) of fingers on an arm. This is unexpected:

![Figure 2: Recursive CTE to find the bill of materials for a humanoid robot. The computation of qty in Line 6 is buggy.](image)

(a) Provenance derivation for the suspicious output value \( \text{(5)} \) yields where-provenance \( \blacklozenge \) and why-provenance \( \blacklozenge \).

![Figure 3: Input and output tables for CTE \text{bom} of Figure 2.](image)

(b) Provenance after the bug in CTE \text{bom} has been fixed.

no other value in column \text{qty} has been accessed to compute the output \( \text{(5)} \). Indeed, the query disregards parent part quantities while it walks down the hierarchy (and thus misses the fact that a body has two arms, for example). To correctly compute the quantity of a part \text{p}, we need to factor in the quantity of its parent \text{b}. Once we fix the quantity calculation to read \( \text{p.qty} \times \text{b.qty} \) in Line 6, CTE \text{bom} yields the expected finger count \( \text{(10)} \) in output table of Figure 3(b). We also find the expected \( \text{why-provenance} \) now: during hierarchy traversal, we multiply quantities \( \text{(1)} \times \text{(2)} \times \text{(5)} \) to arrive at \( \text{(10)} \), fingers.

### 2.1 Values Here, Dependencies There

This work pursues an approach to provenance derivation [12] that strictly separates

1. the realm of regular values (e.g., the string or integer data found in table cells) from
2. the realm of dependency sets which describe the table cells that influenced the computation of those values.

This strict separation applies to both, tables and queries, used during provenance derivation. We turn to the tables first.

#### Tables and their mirror images

Given a table \( T \), provenance derivation distinguishes between its
• variant \( T^2 \) which is an (almost exact) copy of \( T \) in which table cells hold regular values of the known SQL types, and
• its mirror image \( T^2 \) whose table cells hold dependencies of type \( P \) (the type of sets of cell identifiers).
Throughout, we maintain a one-to-one correspondence between both: any value cell in \( T^2 \) is associated with its dedicated dependency set in \( T^2 \). In consequence, \( T^2 \) has the same cardinality and columns as \( T^1 \). Since \( T^1, T^2 \) are relational tables (and thus are unordered), both carry a column \( q \) of row identifiers which tie corresponding rows together.

Figure 4 shows the input and output tables of the bom query of Figure 2, both in their value and their dependency set variants (disregard the overlaid arrows for row). In the input tables, you will find that any cell value in parts\(^2 \) is associated with a singleton dependency set in parts\(^2 \). To illustrate, value 5 of type int is associated with set \( \{ p_{13} \} \) of type \( P \), both in row \( q_5 \) and column \( q_{13} \) of their tables: cell identifier \( p_{13} \) represents the cell value of 5 and there is no dependency to any other cell.

In the (non-singleton, in general) dependency sets of output table output\(^2 \), provenance derivation has accumulated all cell identifiers that influenced the computation of the associated value cell in output\(^1 \). Row and cell identifiers suffice to trace the data provenance of any output table cell. For output value 10 in table output\(^1 \), for example, we find the following (following the arrows overlaid on Figure 4):

1. The computation of 10 (row \( q_{35}, \) column \( q_{74} \)) depended on the 8 input cells with identifiers \( \{ p_{6}, p_{8}, ..., p_{13} \} \), see table output\(^2 \). (The grey \( p_{13} \) indicates why-provenance—we come back to the distinction between provenance kinds in Section 2.2.)
2. Among these, \( p_{13} \) designates row \( q_{25} \), column part of input table parts\(^2 \).
3. The value associated with cell \( p_{13} \) is string arm in row \( q_{5} \), column part of table parts\(^2 \).

(If we trace all 8 input cells influencing output 10, we obtain the 8 green highlights in Figure 3(b).)

We assume that input table \( T^1 \) is provided. Its column \( q \) of row identifiers may either hold externalized RDBMS-internal row IDs or can be generated explicitly through row numbering. Table \( T^2 \) may be derived from \( T^1 \) in terms of a SQL view that invents singleton sets of arbitrary, yet unique cell identifiers (i.e., the \( \{ p_i \} \) discussed above). Several options exist to represent these dependency sets in a SQL system: in [12], we discuss arrays as well as variable-length arrays of arbitrary, yet unique cell identifiers (\( T^2 \) can be generated explicitly through row numbering. Table \( T^2 \) may be derived from \( T^1 \) in terms of a SQL view that invents singleton sets of arbitrary, yet unique cell identifiers (i.e., the \( \{ p_i \} \) discussed above). Several options exist to represent these dependency sets in a SQL system: in [12], we discuss arrays as well as variable-length arrays of arbitrary, yet unique cell identifiers (\( T^2 \) can be generated explicitly through row numbering. Table \( T^2 \) may be derived from \( T^1 \) in terms of a SQL view that invents singleton sets of arbitrary, yet unique cell identifiers (i.e., the \( \{ p_i \} \) discussed above). Several options exist to represent these dependency sets in a SQL system:

Queries and their mirror images. The strict separation of the value and dependency set realms is reflected by queries as well. Given a SQL query \( q \), we systematically rewrite it into a query pair \( q \Rightarrow (q^1, q^2) \) that we evaluate in two phases (see Figure 5):

1. In Phase 1, SQL query \( q^1 \) consumes tables \( T^1, ..., T^m \) of cell values of the regular SQL types and emits table output\(^1 \). Unlike \( q^1 \), \( q^2 \) additionally writes a protocol about value-based decisions performed during query evaluation.
2. In Phase 2, SQL query \( q^2 \) entirely operates over tables of dependency sets \( T^{i_1}, ..., T^{i_n} \), i.e., all table cells processed by \( q^2 \) are of type \( P \). While it executes, query \( q^2 \) consults the interim protocol and finally emits table output\(^2 \).

While the paired queries operate in separate realms, both (1) read and emit tables of identical shape (cardinality and column width) and (2) adhere to a common syntactic structure: the rewrite \( \Rightarrow \) maps subexpressions \( e^i \) of \( q^1 \) to their mirror image \( e^2 \) in a compositional fashion. In a nutshell, both expressions relate as follows:

- Assume \( e^1 \equiv x \circledast y \) (where \( \circledast \) denotes an arbitrary binary SQL operator). If \( e^1 \) yields value \( z \), the dependencies of \( z \) comprise the dependencies of both argument values \( x \) and \( y \).
- The mirror image of \( e^1 \) will be \( e^2 \equiv x \cup y \) in which \( x \) and \( y \) are the dependency sets of the values \( x \) and \( y \). \( e^2 \) thus reads and emits sets of type \( P \).

Interim protocols. If \( q^1 \) contains a clause \( \text{WHERE} e^1 \), the Boolean value of \( e^1 \) is used to perform value-based decisions during query evaluation in Phase 1. Note that \( q^2 \) will not be able to simply use \( e^2 \) to reexecute these decisions in Phase 2, since \( e^2 \) is of type \( P \). Instead, we instrument query \( q^2 \) to invoke function \( \text{write}_2((\circledast, v_1, q, ..., v_n, q), q) \) to record the fact that \( e^2 \) evaluated to true. In this call,

- \( q \) identifies the WHERE clause’s location in the SQL text (\( q^1 \) may contain multiple such clauses), and
- the \( v_1, ..., v_n \) denote the SQL row variables that occur free in \( e^2 \) (the Boolean value of \( e^2 \) depends on the rows bound to these variables).

Function \( \text{write}_2 \) then (1) returns a unique row identifier \( q \) representing the row that passed predicate \( e^2 \) under the current bindings of the \( v_i \), and (2) saves \( (\circledast, v_1, q, ..., v_n, q, q) \) to a persistent protocol to record this value-based decision.

Phase 2 can reexecute just this behavior through the invocation of \( \text{read}_2((\circledast, v_1, q, ..., v_n, q)) \). \( \text{read}_2 \) returns \( q \) if \( (\circledast, v_1, q, ..., v_n, q, q) \) is found in the protocol but yields \( \varnothing \) (represented as the empty table) otherwise. We have thus defined \( \Rightarrow \) to replace predicate evaluation in \( q^1 \) with corresponding \( \text{read}_2 \) invocations in \( q^2 \) (peek ahead at Lines 2 and 5 in Figures 7(a) and 7(b) to see how this plays out for the bom query). Analogously, the interim protocol can be used to communicate the formation of groups or the elimination of rows due to DISTINCT from Phase 1 to Phase 2.

Again, multiple implementation options exist to realize the protocol and the side effects performed by \( \text{read}_2 \) and \( \text{write}_2 \). In [12], both functions were realized as SQL UDFs that write to and read from a common table. We will report on protocol sizes in Section 3.2.

The companion paper [12] defined syntax-directed rewrites \( q \Rightarrow (q^1, q^2) \) for a broad range of SQL constructs. Below, we add rewriting rules for recursive CTEs (and recursive UDFs in Section 3).

2.2 Recursive Provenance Derivation

In tandem, the rewrite Rules With and UnionAll of Figure 6 extend the definition of \( \Rightarrow \) to embrace the syntactic shape of recursive CTEs (see Figure 1). Both rules follow the principle of compositionality: the rewrite of a complex query construct is assembled from the rewrites of its (simpler) constituent queries—see the rewrites \( q_1 \Rightarrow (q^1_1, q^2_1) \) (\( i = 0, ..., n \)) in Rule With as well as \( q_0 \Rightarrow (q^1_0, q^2_0) \) and \( q_1 \Rightarrow (q^1_1, q^2_1) \) in Rule UnionAll. Rule With extends the CTE’s column list by column \( q \) to preserve the row identifiers that tie values and their associated dependency sets together (recall Section 2.1). Other than that, both rules preserve the syntactic shape.
of the original query, a salient feature of the two-phase approach that aids the efficient evaluation of the rewritten queries [12].

Given these extensions, \( \Rightarrow \) rewrites the bom CTE of Figure 2 into the CTE pair \((\text{bom}^1, \text{bom}^2)\) depicted in Figure 7. Where these generated CTEs exhibit relevant changes from the original bom, we have added blue highlights.

**Phase 1, Figure 7(a).** In CTE \(\text{bom}^1\), Line 2 calls \(\text{write}_\text{FILTER}(\p, \p \cdot \varrho, \varrho)\) to record the fact that row \(p\) has passed the predicate \(p.\text{part} = \text{'humanoid'}\). Likewise, \(\text{write}_\text{JOIN}(\p, \p \cdot \varrho, \varrho)\) in Line 8 creates a protocol entry if the rows bound to row variables \(p\) and \(b\) joined under condition \(p.\text{part} = b.\text{sub_part}\).

When query \(\text{bom}^2\) is executed on input table \(\text{parts}^1\) of Figure 4(a), the recursive CTE performs the initial query \(q_0\) in Lines 2–5 once before it iterates query \(q_1\) in Lines 8–11 twice (recall Figure 1). Figure 8 shows how the intermediate results of these queries are assembled to form \(\text{output}^1\). Side effects on the protocol are shown under heading protocol writes (read Figure 8 top-down). The protocol reflects the characteristic iterative nature of a SQL CTE:

- In the first iteration of \(q_1\), rows \(q_3\) and \(q_4\) are joined with row \(q_2\) which has been generated by the initial query \(q_0\) (establishing that \(\text{arm}\) and \(\text{leg}\) are parts of the robot body).
- In the second iteration of \(q_1\), rows \(q_5\) and \(q_6\) join with rows \(q_3\) and \(q_2\), respectively, which have just been generated by the first iteration (\(\text{finger}\) is part of \(\text{arm}\), \(\text{foot}\) is part of \(\text{leg}\)).

The protocol contents thus contain a complete history of the iterated joins performed by the CTE.

**Phase 2, Figure 7(b).** CTE \(\text{bom}^2\) operates over dependency sets and thus trades value-based expressions like \(p.\text{qty} \cdot b.\text{qty}\) for \(p.\text{qty} \cup b.\text{qty}\) (see Line 9 in \(\text{bom}^1\) and \(\text{bom}^2\)) to compute the where-provenance of the arithmetic operation. A predicate like \(p.\text{part} = b.\text{sub_part}\) decides whether the current bindings for row variables \(p\) and \(b\) may contribute to the query result. We thus

1. derive the predicate’s dependency set by \(p.\text{part} \cup b.\text{sub_part}\), and
2. use function \(Y(\cdot)\) to mark all cell identifiers in that set as why-provenance, see Lines 6 and 12 in \(\text{bom}^2\) (in table \(\text{output}^2\) of Figure 4(b) we have colored these cell identifiers in grey: \(p_i\)). Since why-provenance affects all columns of a row that passed a predicate, we bind the resulting dependency set to \(\text{wh} \cdot y\) once and then refer to that alias as needed to avoid recomputation effort.

Since we exclusively operate over dependency sets, function call \(\text{read}_\text{FILTER}(\varrho, p, \varrho)\) in Line 5 assumes the role of predicate \(p.\text{part} = \text{‘humanoid’}\). Likewise, \(\text{read}_\text{JOIN}(\varrho, p, \varrho)\) reenacts the value-based predicate \(p.\text{part} = b.\text{sub_part}\) both functions return the empty table if the corresponding protocol entries are missing. In that case, the current bindings for row variables \(p\) and \(b\) will not contribute to the query result (just like in Phase 1). If a protocol entry is found, the functions return row identifier \(\log.\varrho\) of the row that passed the predicate (e.g., \(\text{read}_\text{JOIN}(\varrho, q_3, q_5)\) yields \(q_5\), see the last but one protocol row in Figure 8). In effect, \(\text{bom}^2\) will show the exact filtering/join behavior like its mirror CTE \(\text{bom}^1\). In particular, the queries will perform the exact same number of CTE iterations in both phases.

### WITH RECURSIVE ...

This approach to provenance derivation can be adapted to cover recursive CTEs with set semantics in which iteration ends when no new result rows are produced by \(\varrho\). Since the generation of unique row identifiers by the \(\text{write}_\text{CTE}\) functions can affect the selection of row duplicates, this calls for an appropriate implementation of equality on row identifiers (column \(\varrho\)). We do not elaborate on the details here, but examples of such CTEs are found in the GitHub repository accompanying this paper (see Section 5).
Applications of DTW abound and include speech recognition or feature comparison in machine-learning setups.

As input, dtw assumes a tabular encoding distances \((i, j, \delta)\) of the distance matrix of both time series: a row \((i, j, \delta)\) indicates that \(\delta = |x_i - y_j|\). Figure 10 shows the distances tables and its matrix representation for two sample time series of length 6 depicted in Figure 10(c). A UDF call \(\text{dtw}(6, 6, w=1)\) for these time series yields a distance score of 1.0.

If we derive \textit{where}- and \textit{why}-provenance for result value 1.0 to understand how UDF \textit{dtw} arrived at this score, we find the dependencies marked \(\bigodot\) and \(\bigcirc\) in Figure 10. We find that DTW has inspected all entries in distances that are close to the matrix’ main diagonal, i.e., if their indices satisfy \(|i - j| \leq w\) \((\textit{why}-\textit{provenance} \bigodot\textit{along the diagonal})\). Inside this window around the diagonal, DTW has added the minimal distances to find the overall score: in Figure 10(c). A UDF call \(\text{dtw}(6, 6, w=1)\) for these time series yields a distance score of 1.0.

We also learn that a \textit{warp} parameter \(w=2\) would have widened the

![Figure 9: A recursive SQL UDF that implements DTW.](image-url)
window around the diagonal to include the distances at \((i, j)\) coordinates \((3, 5)\) and \((4, 6)\) to obtain a perfect DTW score of 0.0. (Widening the window inspects more data as why-provenance has shown but may lead to better matches—a typical tradeoff in the use of DTW.)

### 3.1 Rewriting UDFs for Provenance Derivation

We stick to the principle of two-phase provenance derivation: the extended rules for \( \Rightarrow \) in Figure 11 rewrite a recursive UDF \( f \) into a pair of UDFs \( \langle f_1^1, f_2^1 \rangle \). The application of these rules to UDF \( \text{dtw} \) of Figure 9 yields the UDF pair \( \langle \text{dtw}^1, \text{dtw}^2 \rangle \) shown in Figure 12.

As expected, \( \text{dtw}^1 \) computes over regular values (consuming 1nt parameters, yielding the score of type float), while \( \text{dtw}^2 \) receives and returns dependency sets of type P.

The body of a (recursive) UDF is a regular SQL query \( q \). Rule UDF-Def thus recursively applies rewrite \( \Rightarrow \) to \( q \) to obtain function bodies \( \langle q_1^1, q_2^1 \rangle \) that perform provenance derivation. These rewrites handle SQL constructs like CASE...WHEN or the invocation of non-recursive functions like LEAST (n-way minimum). Note how literals like \( 0 \) or \( 1 \) are represented as empty dependency sets \( \emptyset \) in Phase 2: a literal does not depend on any input table cell [12]. Unlike the original UDF which we assume to be a true read-only (or: STABLE)
function, the body of \(\text{dtw}^1\) will perform protocol writing (recall the discussion in Section 2.1 as well as Figure 8). Rule UDF-\text{DEF} changes the function volatility category of \(\text{dtw}^1\) to VOLATILE to support UDFs that recurse deeply. Dependency sets of which were used for PostgreSQL’s buffer. We extended the stack on a Linux-based Intel Xeon machine with 72 GB of RAM, 32 GB of which were used for PostgreSQL’s buffer. We extended the stack size to 64 MB to support UDFs that recurse deeply. Dependency sets were represented using bit sets which efficiently support duplicate elimination and the central \(\cup\) operation (the associated PostgreSQL extension is discussed in [12]). We report the average runtime of five query runs (worst and best runs discarded).

**Recursive call counts.** UDF \(\text{dtw}\) indeed constitutes a true stress test for recursive query evaluation in general and provenance derivation in particular. The UDF features three-fold recursion which leads to a number of recursive calls that is exponential in the length of the time series [6]: the naive, non-memorizing implementation of \(\text{dtw}\) in Figure 9 already performs about 800 000 such calls for time series of length 10. In all plots of Figure 13, the x-axes range over the recursive call count. The 10 data points of each curve report on a series of \(\text{dtw}\) runs over time series lengths of 1 ... 10. The largest input instance consists of \(10 \times 10 = 100\) rows stored in the \(\text{dtw}\) instances table, effectively the cross product of both time series. (Note that this instance already leads to millions of recursive UDF invocations.)

**Runtime overhead of provenance derivation.** In the proposed approach, provenance derivation for a SQL query \(q\) is complete only once both rewritten variants \(q^1\) and \(q^2\) have been evaluated. The interim protocol forces the Phases 1 and 2 to be performed sequentially. If \(t(q)\) denotes the runtime required to evaluate query \(q\), we thus observe a provenance derivation overhead of \(\frac{t(q^1) + t(q^2)}{t(q)}\) relative to the original subject query \(q\).

The curves of Figure 13(a) report on this overhead for a SQL query that invokes UDF \(\text{dtw}\) (and performs no other computation). If we derive where- as well as why-provenance (see the \(\bullet\) curve), we observe slowdowns that range between 3.2\(\times\) and 13.4\(\times\). As we travel the x-axis left to right and thus perform more and more recursive calls while we process longer time series, the overhead increases: each call requires protocol access in both phases (\(\text{write}_{\text{CALL}}/\text{read}_{\text{CALL}}\)), protocol size grows, and more dependency set operations have to be performed. Beyond 10 000 \(\text{dtw}\) calls, the slowdown reaches a plateau indicating that the cost of UDF invocation itself becomes significant (\(q\) pays this price just like \(q^1\) and \(q^2\) do).

If we derive why-provenance only (curve \(\triangle\)), the overhead never exceeds 4.0\(\times\). We certainly expect the construction of smaller dependency sets (all cell identifiers \(p\) are missing), but this alone cannot explain the significant overhead reduction. Since \(q\) and \(q^1\) remain identical, savings must occur in Phase 2 and \(q^2\). That is exactly what we find below.

---

**Figure 13:** Quantifying the runtime and protocol size overhead of provenance derivation (recursive UDF \(\text{dtw}\)).

---

3.2 Provenance: Priceless, but Not for Free

It is expected that provenance derivation introduces tangible query runtime overhead:

- provenance derivation requires two query evaluation phases,
- the Phases 1 and 2 write and then read the interim protocol, and
- dependency sets may be of substantial cardinality and will certainly exceed the size of regular 1NF table cell values.

Below, we quantify this overhead for the recursive UDF \(\text{dtw}\) of Figure 9. We will also learn, however, that the availability of the protocol can be a true advantage.

These experiments were performed on PostgreSQL v14.1, hosted on a Linux-based Intel Xeon machine with 72 GB of RAM, 32 GB of which were used for PostgreSQL’s buffer. We extended the stack size to 64 MB to support UDFs that recurse deeply. Dependency sets were represented using bit sets which efficiently support duplicate...
Wall clock times. The original query $q$ runs fastest, $q^3$ requires additional time for protocol access, on top of that $q^2$ juggles potentially large dependency sets and will be slowest. This is just what the plot of query execution times in Figure 13(b) shows. Yet, the curves $\bullet$, $\circ$, and $\times$ run parallel which indicates that PostgreSQL is able to find query plans for the $q^2$ and $q^3$ that—despite the side-effecting operations of protocol writing and reading—exhibit runtime characteristics comparable to the plans for $q$ (this also explains the overhead plateau in Figure 13(a)).

Interestingly, the protocol can save Phase 2 and thus $q^2$ from the exponential complexity of three-fold recursion if we derive where-provenance only (see curve $\circ$). In the original query $q$ as well $q^3$, the evaluation of the three-way minimum $\text{LEAST}(e_1, e_2, e_3)$ requires the evaluation of all arguments $e_i$ for $dt\omega$, this leads to three recursive calls (see Lines 16–18 in Figure 12(a)). When $q^2$ evaluates $\text{LEAST}$, however, the protocol already reveals which of the three branches (say $e_m$, $m \in \{1, 2, 3\}$) yielded the minimum. (Internally, $\text{LEAST}$ uses $\text{writeCASE}/\text{readCASE}$; refer to Lines 3–7 in Figures 12(a) and 12(b) but disregard the terms $Y(\cdot)$.) Pursuing branch $e_m$ only leads to a single recursive call, effectively turning $dt\omega$ into a linear recursive function. Indeed, we find $q^2$ to use negligible time of no more than 26 ms across all time series lengths. Similar, yet less dramatic, Phase 2 savings have also been found in [12].

Protocol size. The interim protocol provides essential glue between Phases 1 and 2. All timings reported in this section are based on a tabular implementation of the protocol that $\text{writeCASE}/\text{readCASE}$ write to and read from. Figure 13(c) reports that both side-effecting functions move between 120 kB and 160 MB of protocol data (between 16 and 3 400 000 protocol entries), depending on the number of recursive $dt\omega$ calls.

### 4 MORE RELATED WORK

We view this work as a proof of the versatility of the two-phase approach to cell-level provenance derivation for SQL. Our focus in the companion paper [12] has been on the derivation of where- and why-provenance for a practically relevant (yet non-recursive) dialect of SQL, including correlation, grouping and aggregation, or window functions. The present findings blend with that work and also fit with the computation of how-provenance for SQL [13].

With Perm [9, 10], we share the goal to bring data provenance to rich SQL subsets. Perm derives provenance for aggregates, set operations, or correlated subqueries, but has not embraced recursion. Its implementation is based on a PostgreSQL kernel that has been modified to support data provenance. We have deliberately designed Phases 1 and 2 to build on non-invasive, language-level query transformations that apply to a variety of RDBMS backends. In this respect, we are closer to Glavic’s GProM [1].

Provenance support for Datalog—in a sense the prototypical relational query language to support recursion—has already been established by the early works of the field, notably in the perva-sional semirings approach [11]. These foundations are turned into an efficient implementation by [15] which instruments the internal relational algebra machine program for a given Datalog query to derive its semiring-based provenance. Indeed, query instrumentation is a common theme that spans our efforts, the just mentioned work of Senellart and his colleagues, as well as the effort of Deucht et al. [4]. The latter realizes a notion of how-provenance for recursive Datalog in which the relevant intensional facts (or: derivation trees) of a query are computed. The returned provenance can be queried—also to control its potentially huge size—an idea that we fully subscribe to: much like Perm, we return a relational representation of dependency sets (recall Figure 4) that is subject to exploration via SQL itself.

Recursive queries may yield infinite provenance if all possible derivations of an output row are considered [11]. Deucht et al. employ boolean circuits [5] to mitigate this. In contrast, the present approach derives a single dependency set per output cell.

### 5 WRAP-UP

Recursion is key to bring complex computation close to database-resident data. We pursue this thread of work to bring the derivation of data provenance in line with the recursive SQL query constructs found “in the wild,” recursive common table expressions and user-defined functions, in particular. Provenance derivation is especially valuable when the runtime behavior of a query can be intricate (or even puzzling), e.g., in the presence of fixpoint computation or $n$-fold, non-tail recursive functions.

While our discussion has revolved around CTE bom and UDF dt\omega, we have prepared a GitHub repository\(^1\) that holds the original as well as instrumented SQL sources for a series of further recursive queries. Applications include parsing, string matching, 2D pixel processing (Marching Squares), and graph algorithms. All queries are ready to run, include sample data, and can be evaluated on any contemporary PostgreSQL instance.

### REFERENCES


---

\(^1\)https://github.com/DBatUTuebingen/provenance-for-recursive-queries