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

ACM Reference Format:
Benjamin Dietrich, Tobias Müller, and Torsten Grust. 2022. Data Provenance for Recursive SQL Queries. In *14th International Theory and Practice of Provenance (TaPP'22), June 17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3530800.3534536

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:

TaPP'22, June 17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9349-2/22/06…$15.00
https://doi.org/10.1145/3530800.3534536

1. The semantics of a recursive CTE (see Figure 1) is defined in terms of a fixpoint computation in which a query $q_{\circlearrowleft}(T)$ is iteratively evaluated over its most recent result table T until no (previously unseen) rows are produced. The intermediate results of $q_{\circlearrowleft}$ 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.

```
WITH RECURSIVE T(⋯) AS (
  q0     -- evaluated once
    UNION ALL
  q↺(T) -- iterated until ∅
)
TABLE T;
```

Figure 1: Recursive CTE.

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

The variant queries $q^{\mathbb{1}}$ and $q^{\mathbb{2}}$ 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 $\mathbb{2}$ 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 `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 `parts` (see Figure 3(a)). The CTE yields the `output` table of Figure 3(a) which lists the quantities (column `qty`) of all parts (column `sub_part`) required to assemble the robot. The numbers of required fingers and feet (5 and 1, respectively) certainly look suspicious: we had expected `10` and `2`. To understand how CTE `bom` arrived at these questionable results, we explore the provenance of output value 5 (`5` in Figure 3(a)).

The *why*-provenance of `5` reveals all input table cells that have been *inspected to decide* whether `5` occurs in the output table. We find the highlighted input cells `arm`, `body`, and `humanoid` which the CTE has used to recursively descend into the part hierarchy encoded by table `parts` (see predicate `p.part = 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 `bom` appears to be in order.

The *where*-provenance of output `5`, however, is shown to only refer to the quantity `5` of fingers on an arm. This is unexpected:

```
1  WITH RECURSIVE bom(part, sub_part, qty) AS (
2      SELECT p.part, p.sub_part, p.qty
3      FROM   parts AS p
4      WHERE  p.part = 'humanoid'
5    UNION ALL
6      SELECT p.part, p.sub_part, p.qty -- BUG (should read p.qty * b.qty)
7      FROM   bom AS b,
8             parts AS p
9      WHERE  p.part = b.sub_part
10 )
11 SELECT b.sub_part, b.qty
12 FROM   bom AS b
```

Figure 2: Recursive CTE to find the bill of materials for a humanoid robot. The computation of `qty` in Line 6 is buggy.

`parts`

part	sub_part	qty
humanoid	head	1
humanoid	body	1
body	arm	2
body	leg	2
arm	finger	5
leg	foot	1
chassis	wheel	4

`output (buggy)`

sub_part	qty
head	1
body	1
arm	2
leg	2
finger	5
foot	1

(a) Provenance derivation for the suspicious output value `5` yields *where*-provenance and *why*-provenance .

`parts`

part	sub_part	qty
humanoid	head	1
humanoid	body	1
body	arm	2
body	leg	2
arm	finger	5
leg	foot	1
chassis	wheel	4

`output`

sub_part	qty
head	1
body	1
arm	2
leg	2
finger	10
foot	2

(b) Provenance after the bug in CTE `bom` has been fixed.

Figure 3: Input and output tables for CTE `bom` of Figure 2.

no other value in column `qty` has been accessed to compute the output `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 `p`, we need to factor in the quantity of its parent part `b`. Once we fix the quantity calculation to read `p.qty * b.qty` in Line 6, CTE `bom` yields the expected finger count `10` in output table of Figure 3(b). We also find the expected *where*-provenance now: during hierarchy traversal, we multiply quantities `1` * `2` * `5` to arrive at `10` fingers.

2.1 Values Here, Dependencies There

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

$\mathbb{1}$. the *realm of regular values* (*e.g.*, the string or integer data found in table cells) from

$\mathbb{2}$. 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 $T^{\mathbb{1}}$ which is an (almost exact) copy of T in which table cells hold regular values of the known SQL types, and
- its mirror image $T^{\mathbb{2}}$ whose table cells hold dependencies of type $\mathbb{P}$ (the type of sets of cell identifiers).

Throughout, we maintain a one-to-one correspondence between both: any value cell in $T^{\mathbb{1}}$ is associated with its dedicated dependency set in $T^{\mathbb{2}}$. In consequence, $T^{\mathbb{2}}$ has the same cardinality and columns as $T^{\mathbb{1}}$. Since $T^{\mathbb{1}}$, $T^{\mathbb{2}}$ are relational tables (and thus are unordered), both carry a column ϱ 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 now). In the input tables, you will find that any cell value in `parts`$^{\mathbb{1}}$ is associated with a *singleton* dependency set in `parts`$^{\mathbb{2}}$. To illustrate, value 5 of type `int` is associated with set $\{p_{15}\}$ of type $\mathbb{P}$, both in row ϱ_5 and column `qty` of their tables: cell identifier p_{15} represents the cell of value 5 and there is no dependency to any other cell.

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

1. The computation of `10` (row ϱ_{35}, column `qty`) depended on the 8 input cells with identifiers $\{p_6, p_9, \dots, p_{13}\}$, see table `output`$^{\mathbb{2}}$. (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 ϱ_5, column `part` of input table `parts`$^{\mathbb{2}}$.
3. The value associated with cell p_{13} is string `arm` in row ϱ_5, column `part` of table `parts`$^{\mathbb{1}}$.

(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 $T^{\mathbb{1}}$ is provided. Its column ϱ of row identifiers may either hold externalized RDBMS-internal row IDs or can be generated explicitly through row numbering. Table $T^{\mathbb{2}}$ may be derived from $T^{\mathbb{1}}$ 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 bit sets as two possible implementations of type $\mathbb{P}$.

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 $q \Mapsto \langle q^{\mathbb{1}}, q^{\mathbb{2}} \rangle$ that we evaluate in two phases (see Figure 5):

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

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 $\Mapsto$ maps subexpressions $e^{\mathbb{1}}$ of $q^{\mathbb{1}}$ to their mirror image $e^{\mathbb{2}}$ of $q^{\mathbb{2}}$ in a compositional fashion. In a nutshell, both expressions relate as follows:

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

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

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

Function $write_{\square}$ then (1) returns a unique row identifier ϱ representing the row that passed predicate $e^{\mathbb{1}}$ under the current bindings of the v_i, and (2) saves $(\copyright, v_1.\varrho, \dots, v_n.\varrho, \varrho)$ to a persistent protocol to record this value-based decision.

Phase $\mathbb{2}$ can reenact just this behavior through the invocation of $read_{\square}(\copyright, v_1.\varrho, \dots, v_n.\varrho)$. $read_{\square}$ returns ϱ if $(\copyright, v_1.\varrho, \dots, v_n.\varrho, \varrho)$ is found in the protocol but yields $\varnothing$ (represented as the empty table) otherwise. We have thus defined $\Mapsto$ to replace predicate evaluation in $q^{\mathbb{1}}$ with corresponding $read_{\square}$ invocations in $q^{\mathbb{2}}$ (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 $\mathbb{1}$ to Phase $\mathbb{2}$.

Again, multiple implementation options exist to realize the protocol and the side effects performed by $read_{\square}$ and $write_{\square}$. 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 $q \Mapsto \langle q^{\mathbb{1}}, q^{\mathbb{2}} \rangle$ 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 $\Mapsto$ 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 $q_i \Mapsto \langle q_i^{\mathbb{1}}, q_i^{\mathbb{2}} \rangle$ $(i = 0, \dots, n)$ in Rule WITH as well as $q_0 \Mapsto \langle q_0^{\mathbb{1}}, q_0^{\mathbb{2}} \rangle$ and $q_{\circlearrowleft} \Mapsto \langle q_{\circlearrowleft}^{\mathbb{1}}, q_{\circlearrowleft}^{\mathbb{2}} \rangle$ in Rule UNIONALL. Rule WITH extends the CTE's column list by column ϱ 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 parts$^{\mathbb{1}}$

part	sub_part	qty	ϱ
humanoid	head	1	ϱ_1
humanoid	body	1	ϱ_2
body	arm	2	ϱ_3
body	leg	2	ϱ_4
arm	finger	5	ϱ_5
leg	foot	1	ϱ_6
chassis	wheel	4	ϱ_7

parts$^{\mathbb{2}}$

part	sub_part	qty	ϱ
$\{p_1\}$	$\{p_2\}$	$\{p_3\}$	ϱ_1
$\{p_4\}$	$\{p_5\}$	$\{p_6\}$	ϱ_2
$\{p_7\}$	$\{p_8\}$	$\{p_9\}$	ϱ_3
$\{p_{10}\}$	$\{p_{11}\}$	$\{p_{12}\}$	ϱ_4
$\{p_{13}\}$	$\{p_{14}\}$	$\{p_{15}\}$	ϱ_5
$\{p_{16}\}$	$\{p_{17}\}$	$\{p_{18}\}$	ϱ_6
$\{p_{19}\}$	$\{p_{20}\}$	$\{p_{21}\}$	ϱ_7

(a) Input tables (original and dependency set mirror image).

output$^{\mathbb{1}}$

sub_part	qty	ϱ
head	1	ϱ_{31}
body	1	ϱ_{32}
arm	2	ϱ_{33}
leg	2	ϱ_{34}
finger	10	ϱ_{35}
foot	2	ϱ_{36}

output$^{\mathbb{2}}$

sub_part	qty	ϱ
$\{p_2, p_1\}$	$\{p_3, p_1\}$	ϱ_{31}
$\{p_5, p_4\}$	$\{p_6, p_4\}$	ϱ_{32}
$\{p_8, p_4, p_5, p_7\}$	$\{p_6, p_9, p_4, p_5, p_7\}$	ϱ_{33}
$\{p_{11}, p_4, p_5, p_{10}\}$	$\{p_6, p_{12}, p_4, p_5, p_{10}\}$	ϱ_{34}
$\{p_{14}, p_4, p_5, p_7, p_8, p_{13}\}$	$\{p_6, p_9, p_{15}, p_4, p_5, p_7, p_8, p_{13}\}$	ϱ_{35}
$\{p_{17}, p_4, p_5, p_{10}, p_{11}, p_{16}\}$	$\{p_6, p_{12}, p_{18}, p_4, p_5, p_{10}, p_{11}, p_{16}\}$	ϱ_{36}

(b) Output tables generated by provenance derivation.

Figure 4: Input and output tables of the `bom` query of Figure 2 with row identifiers ϱ. Arrows trace output value `10` back to its dependency `arm` in input table `parts`$^{\mathbb{1}}$. In `output`$^{\mathbb{2}}$, p_i denotes *where*-provenance while p_j denotes *why*-provenance.

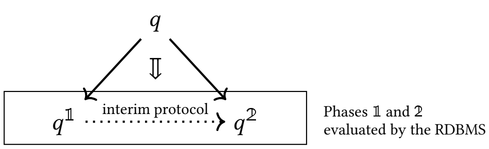


Figure 5: Two-phase provenance derivation.

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, $\Rrightarrow$ rewrites the bom CTE of Figure 2 into the CTE pair ⟨bom$^{\mathbb{1}}$, bom$^{\mathbb{2}}$⟩ depicted in Figure 7. Where these generated CTEs exhibit relevant changes from the original bom, we have added blue highlights.

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

When query bom$^{\mathbb{1}}$ is executed on input table parts$^{\mathbb{1}}$ of Figure 4(a), the recursive CTE performs the initial query q_0 in Lines 2–5 once before it iterates query $q_{\circlearrowleft}$ in Lines 8–11 twice (recall Figure 1). Figure 8 shows how the intermediate results of these queries are assembled to form output$^{\mathbb{1}}$. 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 $q_{\circlearrowleft}$, rows ϱ_3 and ϱ_4 are joined with row ϱ_{32} which has been generated by the initial query q_0 (establishing that `arm` and `leg` are parts of the robot body),
- in the second iteration of $q_{\circlearrowleft}$, rows ϱ_5 and ϱ_6 join with rows ϱ_{33} and ϱ_{34}, respectively, which have just been generated by the first iteration (`finger` is part of `arm`, `foot` is part of `leg`).

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

Phase $\mathbb{2}$, Figure 7(b). CTE bom$^{\mathbb{2}}$ operates over dependency sets and thus trades value-based expressions like `p.qty * b.qty` for `p.qty ∪ b.qty` (see Line 9 in bom$^{\mathbb{1}}$ and bom$^{\mathbb{2}}$) to compute the *where*-provenance of the arithmetic operation. A predicate like `p.part = b.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.part ∪ b.sub_part`, and
2. use function Y(·) to mark all cell identifiers in that set as *why*-provenance, see Lines 6 and 12 in bom$^{\mathbb{2}}$ (in table output$^{\mathbb{2}}$ of Figure 4(b) we have colored these cell identifiers in grey: p_j). Since *why*-provenance affects all columns of a row that passed a predicate, we bind the resulting dependency set to `wh.y` once and then refer to that alias as needed to avoid recomputation effort.

Since we exclusively operate over dependency sets, function call $read_{\texttt{FILTER}}$(①, p.ϱ) in Line 5 assumes the role of predicate `p.part = 'humanoid'`. Likewise, $read_{\texttt{JOIN}}$(②, b.ϱ, p.ϱ) reenacts the value-based predicate `p.part = b.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 $\mathbb{1}$). If a protocol entry is found, the functions return row identifier log.ϱ of the row that passed the predicate (*e.g.*, $read_{\texttt{JOIN}}$(②, ϱ_{33}, ϱ_5) yields ϱ_{35}, see the last but one protocol row in Figure 8). In effect, bom$^{\mathbb{2}}$ will show the exact filtering/join behavior like its mirror CTE bom$^{\mathbb{1}}$. In particular, the queries will perform the exact same number of CTE iterations in both phases.

`WITH RECURSIVE` ... `(`... `UNION DISTINCT` ...`)`. 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 $q_{\circlearrowleft}$. Since the generation of unique row identifiers by the $write_{\square}$ functions can affect the detection of row duplicates, this calls for an appropriate implementation of equality on row identifiers (column ϱ). 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).

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

$$\frac{q_0 \Mapsto \langle q_0^{\mathbb{1}},\, q_0^{\mathbb{2}} \rangle \qquad q_\circlearrowleft \Mapsto \langle q_\circlearrowleft^{\mathbb{1}},\, q_\circlearrowleft^{\mathbb{2}} \rangle}{q_0\ \texttt{UNION ALL}\ q_\circlearrowleft \Mapsto \langle q_0^{\mathbb{1}}\ \texttt{UNION ALL}\ q_\circlearrowleft^{\mathbb{1}},\ q_0^{\mathbb{2}}\ \texttt{UNION ALL}\ q_\circlearrowleft^{\mathbb{2}} \rangle} \text{(UnionAll)}$$

Figure 6: Rewrite rules $q \Mapsto \langle q^{\mathbb{1}}, q^{\mathbb{2}} \rangle$ for recursive SQL CTEs. Combine with the rules of [12].

```
WITH RECURSIVE bom𝟙(ϱ, part, sub_part, qty) AS (
    SELECT writeFILTER(①, p.ϱ) AS ϱ,
           p.part, p.sub_part, p.qty
    FROM   parts𝟙 AS p
    WHERE  p.part = 'humanoid'

  UNION ALL
    SELECT writeJOIN(②, b.ϱ, p.ϱ) AS ϱ,
           p.part, p.sub_part, p.qty * b.qty
    FROM   bom𝟙 AS b, parts𝟙 AS p
    WHERE  p.part = b.sub_part

)
SELECT b.ϱ, b.sub_part, b.qty
FROM   bom𝟙 AS b
```

(a) Phase 𝟙 (CTE bom𝟙).

```
WITH RECURSIVE bom𝟚(ϱ, part, sub_part, qty) AS (
    SELECT log.ϱ,
           p.part ∪ wh.y, p.sub_part ∪ wh.y, p.qty ∪ wh.y
    FROM   parts𝟚 AS p,
           readFILTER(①, p.ϱ) AS log(ϱ),
           Y(p.part) AS wh(y)
  UNION ALL
    SELECT log.ϱ, p.part ∪ wh.y,
           p.sub_part ∪ wh.y, p.qty ∪ b.qty ∪ wh.y
    FROM   bom𝟚 AS b, parts𝟚 AS p,
           readJOIN(②, b.ϱ, p.ϱ) log(ϱ),
           Y(p.part ∪ b.sub_part) AS wh(y)
)
SELECT b.ϱ, b.sub_part, b.qty
FROM   bom𝟚 AS p
```

(b) Phase 𝟚 (CTE bom𝟚).

Figure 7: Rewriting CTE bom (Figure 2) for provenance derivation yields a pair of CTEs.

output$^{\mathbb{1}}$

	part	sub_part	qty	ϱ	protocol writes
q_0	humanoid	head	1	ϱ_{31}	$write_{\text{FILTER}}(①,\ \varrho_1) = \varrho_{31}$
	humanoid	body	1	ϱ_{32}	$write_{\text{FILTER}}(①,\ \varrho_2) = \varrho_{32}$
$q_\circlearrowleft$ iteration 1	body	arm	2	ϱ_{33}	$write_{\text{JOIN}}(②,\ \varrho_{32},\ \varrho_3) = \varrho_{33}$
	body	leg	2	ϱ_{34}	$write_{\text{JOIN}}(②,\ \varrho_{32},\ \varrho_4) = \varrho_{34}$
$q_\circlearrowleft$ iteration 2	arm	finger	10	ϱ_{35}	$write_{\text{JOIN}}(②,\ \varrho_{33},\ \varrho_5) = \varrho_{35}$
	leg	foot	2	ϱ_{36}	$write_{\text{JOIN}}(②,\ \varrho_{34},\ \varrho_6) = \varrho_{36}$

Figure 8: Result rows and protocol entries generated during the evaluation of bom$^{\mathbb{1}}$. Result and protocol grow top-down.

3 TRACING RECURSIVE SQL UDFS

CTEs are but one option to express recursive computation right inside the RDBMS (and thus close to the data). In what follows, we focus on *recursive user-defined functions* which provide an—often concise and readable, even elegant—alternative to CTEs. In a recursive UDF f, recursion is directly expressed through self-invocation of f, much like in a functional programming language. While recursive CTEs underly syntactic monotonicity and linearity restrictions [8], no such constraints apply to recursive UDFs. A number of RDBMSs, including PostgreSQL, Oracle, and Microsoft SQL Server, support recursive UDFs off-the-shelf. Provenance derivation helps to untangle, debug, or visualize the potentially complex recursive computation that such UDFs can perform.

Function `dtw` of Figure 9 follows the recursive UDF style. `dtw` uses three-fold recursion (see the recursive calls in Lines 8–10) to compute the *Dynamic Time Warping* (short: DTW) score of two time series $X = (x_i)$ and $Y = (y_j)$ [2]. DTW has the ability to locally stretch and compress the series to minimize the distance between both. The *warp* parameter `w` of `dtw` determines how far such stretching may go: x_i may be matched with any y_j such that $|i - j| \leqslant$ `w`.

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,`δ`)` of the distance matrix of both time series: a row (i, j, δ) 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 `dtw(6,6,w=1)` for these time series yields a distance score of `1.0`.

If we derive *where*- and *why*-provenance for result value `1.0` to understand how UDF `dtw` arrived at this score, we find the dependencies marked ● and ○ 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 $|i - j| \leqslant$ `w` (*why-provenance* ○ along the diagonal). Inside this window around the diagonal, DTW has added the minimal distances to find the overall score: indeed, `0` + ⋯ + `1` + `0` + `0` yields `1.0` (*where-provenance*). We also learn that a *warp* parameter of `w=2` would have widened the

```
1  CREATE FUNCTION dtw(i int, j int, w int) RETURNS float AS $$
2  SELECT CASE
3            WHEN dtw.i=0 AND dtw.j=0 THEN 0.0
4            WHEN dtw.i=0 OR  dtw.j=0 THEN ∞
5            WHEN abs(dtw.i-dtw.j)>dtw.w THEN ∞
6            ELSE (SELECT d.δ
7                         + LEAST(
8                             dtw(dtw.i-1, dtw.j-1, dtw.w),
9                             dtw(dtw.i-1, dtw.j  , dtw.w),
10                            dtw(dtw.i  , dtw.j-1, dtw.w))
11                  FROM   distances AS d
12                  WHERE  (d.i,d.j)=(dtw.i,dtw.j))
13         END
14 $$ LANGUAGE SQL STABLE;
```

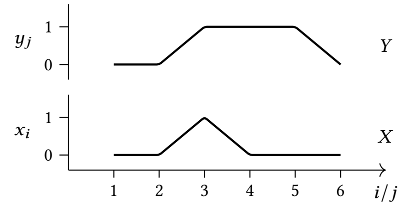
Figure 9: A recursive SQL UDF that implements DTW.

i	j	δ
1	1	0
1	2	0
1	3	1
⋮	⋮	⋮
3	5	0
4	5	1
4	6	0
⋮	⋮	⋮

(a) Distance table.

$$
\begin{array}{c|c}
j & \\
6 & \\
5 & \\
4 & \\
3 & \\
2 & \\
1 & \\
\end{array}
\begin{pmatrix}
0 & 0 & 1 & 0 & 0 & 0\\
1 & 1 & 0 & 1 & 1 & 1\\
1 & 1 & 0 & 1 & 1 & 1\\
1 & 1 & 0 & 1 & 1 & 1\\
0 & 0 & 1 & 0 & 0 & 0\\
0 & 0 & 1 & 0 & 0 & 0
\end{pmatrix}
\quad
\begin{array}{l} i = 1\ 2\ 3\ 4\ 5\ 6 \end{array}
$$

(b) Distance matrix $\delta(i, j)$.



(c) Sample time series $X = (x_i)$ and $Y = (y_j)$.

Figure 10: Distance table, distance matrix, and visualization of the time series $X = (x_i)$, $Y = (y_j)$. Provenance derivation for the recursive UDF `dtw` of Figure 9 generates *where*-provenance ● and *why*-provenance ◯ that explains the workings of DTW.

$$
\frac{
\begin{array}{l}
e_i \Mapsto \langle e_i^{\mathbb{1}}, e_i^{\mathbb{2}} \rangle \Big|_{i=1\ldots n} \qquad ⓒ := \textit{unique call site identifier} \\
\{v_1, \ldots, v_m\} := \textit{free row variables in } e_i, \ldots, e_n \\
i^{\mathbb{1}} := f^{\mathbb{1}}(\textit{write}_{\text{CALL}}(ⓒ, v_1.\varrho, \ldots, v_m.\varrho), e_1^{\mathbb{1}}, \ldots, e_n^{\mathbb{1}}) \\
i^{\mathbb{2}} := f^{\mathbb{2}}(\textit{read}_{\text{CALL}}(ⓒ, v_1.\varrho, \ldots, v_m.\varrho), e_1^{\mathbb{2}}, \ldots, e_n^{\mathbb{2}})
\end{array}
}{
f(e_1, \ldots, e_n) \Mapsto \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle
} \text{(UDF-CALL)}
$$

$$
\frac{
\begin{array}{c}
q \Mapsto \langle q^{\mathbb{1}}, q^{\mathbb{2}} \rangle \\
i^{\mathbb{1}} := \begin{array}{l} \texttt{CREATE FUNCTION} \\ \texttt{udf}^{\mathbb{1}}(\varrho\ \texttt{rowID}, a_1\ \tau_1, \ldots, a_n\ \tau_n) \\ \texttt{RETURNS}\ \tau_{\texttt{res}}\ \texttt{AS \$\$} \\ \quad \texttt{SELECT}\ q^{\mathbb{1}} \\ \texttt{\$\$ LANGUAGE SQL VOLATILE} \end{array}
\qquad
i^{\mathbb{2}} := \begin{array}{l} \texttt{CREATE FUNCTION} \\ \texttt{udf}^{\mathbb{2}}(\varrho\ \texttt{rowID}, a_1\ \mathbb{P}, \ldots, a_n\ \mathbb{P}) \\ \texttt{RETURNS}\ \mathbb{P}\ \texttt{AS \$\$} \\ \quad \texttt{SELECT}\ q^{\mathbb{2}} \\ \texttt{\$\$ LANGUAGE SQL STABLE} \end{array}
\end{array}
}{
\begin{array}{l} \texttt{CREATE FUNCTION udf}(a_1\ \tau_1, \ldots, a_n\ \tau_n) \\ \texttt{RETURNS}\ \tau_{\texttt{res}}\ \texttt{AS \$\$} \\ \quad \texttt{SELECT}\ q \\ \texttt{\$\$ LANGUAGE SQL STABLE} \end{array}
\Mapsto \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle
} \text{(UDF-DEF)}
$$

Figure 11: Rewrite rules $q \Mapsto \langle q^{\mathbb{1}}, q^{\mathbb{2}} \rangle$ for the definition and invocation of recursive UDFs. Combine with the rules of [12].

```
CREATE FUNCTION dtw𝟙(ϱ rowID, i int, j int, w int) RETURNS float AS $$
SELECT
  CASE writeCASE(⑥, dtw𝟙.ϱ, CASE
       WHEN dtw𝟙.i=0 AND dtw𝟙.j=0 THEN 1
       WHEN dtw𝟙.i=0 OR  dtw𝟙.j=0 THEN 2
       WHEN abs(dtw𝟙.i-dtw𝟙.j)>dtw𝟙.w THEN 3
       ELSE 0 END)
  WHEN 1 THEN 0.0
  WHEN 2 THEN '∞'
  WHEN 3 THEN '∞'
  WHEN 0 THEN
   (SELECT t.score FROM (
     SELECT writeJOIN(⑤,dtw𝟙.ϱ, d.ϱ) AS ϱ,
            d.δ + LEAST𝟙(
              writeCALL(④, dtw𝟙.ϱ),
              dtw𝟙(writeCALL(①, dtw𝟙.ϱ), dtw𝟙.i-1, dtw𝟙.j-1, dtw𝟙.w),
              dtw𝟙(writeCALL(②, dtw𝟙.ϱ), dtw𝟙.i-1, dtw𝟙.j  , dtw𝟙.w),
              dtw𝟙(writeCALL(③, dtw𝟙.ϱ), dtw𝟙.i  , dtw𝟙.j-1, dtw𝟙.w)
             ) AS score
     FROM   distances𝟙 AS d
     WHERE  (d.i,d.j)=(dtw𝟙.i,dtw𝟙.j)
    ) AS t(ϱ, score))
  END
$$ LANGUAGE SQL VOLATILE;
```

(a) Phase 𝟙 (UDF dtw𝟙).

```
CREATE FUNCTION dtw𝟚(ϱ rowID, i ℙ, j ℙ, w ℙ) RETURNS ℙ AS $$
SELECT
  CASE readCASE(⑥, dtw𝟚.ϱ)
  WHEN 1 THEN ∅ ∪ Y(dtw𝟚.i ∪ dtw𝟚.j)
  WHEN 2 THEN ∅ ∪ Y(dtw𝟚.i ∪ dtw𝟚.j)
  WHEN 3 THEN ∅ ∪ Y(dtw𝟚.i ∪ dtw𝟚.j ∪ dtw𝟚.w)
  WHEN 0 THEN
    (SELECT t.score FROM (
       SELECT log.ϱ AS ϱ,
              d.δ ∪ LEAST𝟚(
                readCALL(④, dtw𝟚.ϱ),
                dtw𝟚(readCALL(①, dtw𝟚.ϱ),
                     dtw𝟚.i ∪ ∅, dtw𝟚.j ∪ ∅, dtw𝟚.w),
                dtw𝟚(readCALL(②, dtw𝟚.ϱ),
                     dtw𝟚.i ∪ ∅, dtw𝟚.j, dtw𝟚.w),
                dtw𝟚(readCALL(③, dtw𝟚.ϱ),
                     dtw𝟚.i, dtw𝟚.j ∪ ∅, dtw𝟚.w)
              ) ∪ Y(d.i ∪ d.j ∪ dtw𝟚.i ∪ dtw𝟚.j) AS score
       FROM   distances𝟚 AS d,
              readJOIN(⑤, dtw𝟚.ϱ, d.ϱ) AS log(ϱ)
     ) AS t(ϱ, score)) ∪ Y(dtw𝟚.i ∪ dtw𝟚.j ∪ dtw𝟚.w)
  ELSE ∅
  END
$$ LANGUAGE SQL STABLE;
```

(b) Phase 𝟚 (UDF dtw𝟚).

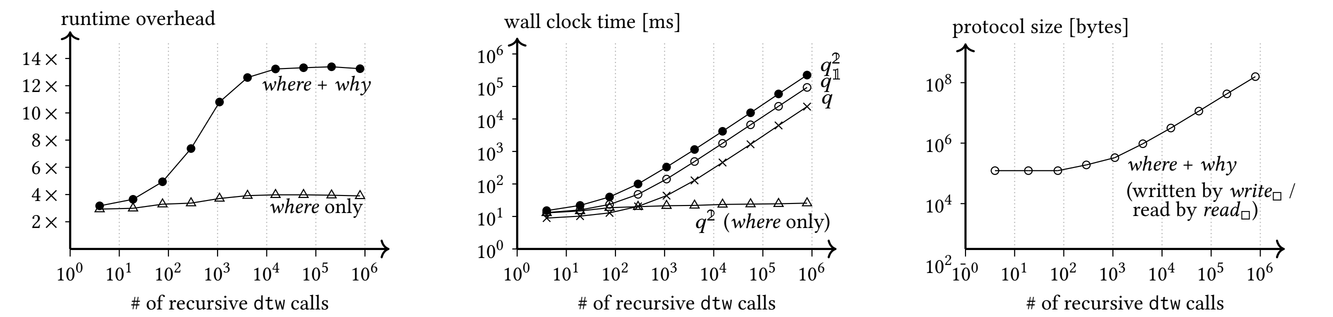
Figure 12: Generated pair of recursive UDFs to derive *where*- and *why*-provenance for SQL UDF `dtw` (Figure 9).

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 $\Mapsto$ in Figure 11 rewrite a recursive UDF f into a pair of UDFs $\langle f^{\mathbb{1}}, f^{\mathbb{2}} \rangle$. The application of these rules to UDF `dtw` of Figure 9 yields the UDF pair $\langle \texttt{dtw}^{\mathbb{1}}, \texttt{dtw}^{\mathbb{2}} \rangle$ shown in Figure 12. As expected, $\texttt{dtw}^{\mathbb{1}}$ computes over regular values (consuming `int` parameters, yielding the score of type `float`), while $\texttt{dtw}^{\mathbb{2}}$ receives and returns dependency sets of type $\mathbb{P}$.

The body of a (recursive) UDF is a regular SQL query q. Rule UDF-DEF thus recursively applies rewrite $\Mapsto$ to q to obtain function bodies $\langle q^{\mathbb{1}}, q^{\mathbb{2}} \rangle$ 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 $\varnothing$ 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`)



(a) Provenance derivation overhead. **(b) Runtime (original, Phases $\mathbb{1}$ + $\mathbb{2}$).** **(c) Protocol table size.**

Figure 13: Quantifying the runtime and protocol size overhead of provenance derivation (recursive UDF `dtw`).

function, the body of dtw$^{\mathbb{1}}$ will perform protocol writing (recall the discussion in Section 2.1 as well as Figure 8). Rule UDF-Def changes the function volatility category of dtw$^{\mathbb{1}}$ to `VOLATILE` to announce this to the RDBMS [14].

In the case of recursive CTEs, extra row identifiers ϱ were introduced to tie values and their dependency sets together. Here, Rule UDF-Def introduces an *extra UDF parameter* ϱ of type `rowID` to serve the same purpose: if the invocation $f^{\mathbb{1}}(\varrho^*, \dots)$ yields a value z, the invocation of the mirror UDF $f^{\mathbb{2}}(\varrho^*, \dots)$ using the same *call identifier* ϱ^* derives z's dependency set. To rewrite a UDF call $f(e_1, \dots, e_n)$, Rule UDF-Call thus collects the free row variables $v_1, \dots, v_m$ among the function arguments e_i and proceeds as follows:

1. In Phase $\mathbb{1}$, instrument the UDF to invoke $\mathit{write}_{\mathtt{CALL}}(Ⓒ, v_1.\varrho, \dots, v_m.\varrho)$ to map the row identifiers of the v_j to a unique call identifier ϱ^*. This mapping is also saved in the interim protocol as entry $(Ⓒ, v_1.\varrho, \dots, v_m.\varrho, \varrho^*)$. ϱ^* is then passed as the first parameter to $f^{\mathbb{1}}$ to identify this particular call.
2. In Phase $\mathbb{2}$, read the protocol via $\mathit{read}_{\mathtt{CALL}}(Ⓒ, v_1.\varrho, \dots, v_m.\varrho)$ to retrieve ϱ^* and pass it to $f^{\mathbb{2}}$. Functions $f^{\mathbb{1}}$ and $f^{\mathbb{2}}$ will thus agree on the call identifier as required.

Note how the body queries of dtw$^{\mathbb{1}}$ and dtw$^{\mathbb{2}}$ pass the call identifier (referred to as dtw$^{\mathbb{1}}.\varrho$ and dtw$^{\mathbb{2}}.\varrho$ in Figures 12(a) and 12(b)) to all embedded $\mathit{write}_{\Box}/\mathit{read}_{\Box}$ invocations: at query runtime, this ensures that protocol entries can be unambiguously associated with the current UDF call in progress.

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 $\mathbb{1}$ and $\mathbb{2}$ 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 `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 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 `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-memoizing implementation of `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 `dtw` runs over time series lengths of $1 \dots 10$. The largest input instance consists of $10 \times 10 = 100$ rows stored in the `distances` 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 $q^{\mathbb{1}}$ and $q^{\mathbb{2}}$ have been evaluated. The interim protocol forces the Phases $\mathbb{1}$ and $\mathbb{2}$ to be performed sequentially. If $t(q)$ denotes the runtime required to evaluate query q, we thus observe a provenance derivation overhead of $\big(t(q^{\mathbb{1}}) + t(q^{\mathbb{2}})\big)/t(q)$ relative to the original subject query q.

The curves of Figure 13(a) report on this overhead for a SQL query that invokes UDF `dtw` (and performs no other computation). If we derive *where*- as well as *why*-provenance (see the ● curve), we observe slowdowns that range between 3.2× and 13.4×. 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 ($\mathit{write}_{\mathtt{CALL}}/\mathit{read}_{\mathtt{CALL}}$), protocol size grows, and more dependency set operations have to be performed. Beyond 10 000 UDF calls, the slowdown reaches a plateau indicating that the cost of UDF invocation itself becomes significant (q pays this price just like $q^{\mathbb{1}}$ and $q^{\mathbb{2}}$ do).

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

Wall clock times. The original query q runs fastest, $q^{\mathbb{1}}$ requires additional time for protocol access, on top of that $q^{\mathbb{2}}$ 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 ●, ○, and × largely run in parallel which indicates that PostgreSQL is able to find query plans for the $q^{\mathbb{1}}$ and $q^{\mathbb{2}}$ 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 $\mathbb{2}$ and thus $q^{\mathbb{2}}$ from the exponential complexity of three-fold recursion if we derive *where*-provenance only (see curve △). In the original query q as well $q^{\mathbb{1}}$, the evaluation of the three-way minimum `LEAST(`e_1, e_2, e_3`)` requires the evaluation of *all* arguments e_i—for `dtw`, this leads to three recursive calls (see Lines 16–18 in Figure 12(a)). When $q^{\mathbb{2}}$ evaluates `LEAST`, however, the protocol already reveals which of the three branches (say $e_m, m \in \{1, 2, 3\}$) yielded the minimum. (Internally, `LEAST` uses $\mathit{write}_{\mathsf{CASE}}/\mathit{read}_{\mathsf{CASE}}$; refer to Lines 3–7 in Figures 12(a) and 12(b) but disregard the terms `Y(·)`). Pursuing branch e_m only leads to a single recursive call, effectively turning `dtw` into a linear recursive function. Indeed, we find $q^{\mathbb{2}}$ to use negligible time of no more than 26 ms across all time series lengths. Similar, yet less dramatic, Phase $\mathbb{2}$ savings have also been found in [12].

Protocol size. The interim protocol provides essential glue between Phases $\mathbb{1}$ and $\mathbb{2}$. All timings reported in this section are based on a tabular implementation of the protocol that $\mathit{write}_{\square}/\mathit{read}_{\square}$ 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 `dtw` 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 $\mathbb{1}$ and $\mathbb{2}$ 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' *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 pervasive 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 Deutch *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]. Deutch *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 `dtw`, 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] S.A. Arab, S. Feng, B. Glavic, S. Lee, X. Niu, and Q. Zeng. 2018. GProM - A Swiss Army Knife for Your Provenance Needs. *IEEE Data Eng. Bulletin* 41, 1 (2018).
[2] D.J Berndt and J. Clifford. 1994. Using Dynamic Time Warping to Find Patterns in Time Series. In *AAAI Workshop*. Seattle, WA, USA.
[3] J. Cheney, L. Chiticariu, and W.-C. Tan. 2007. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2007).
[4] D. Deutch, A. Gilad, and Y. Moskovitch. 2018. Efficient Provenance Tracking for Datalog Using Top-k Queries. *VLDB J.* 27, 2 (2018).
[5] D. Deutch, T. Milo, S. Roy, and V. Tannen. 2014. Circuits for Datalog Provenance. In *Proc. ICDT*. Athens, Greece.
[6] C. Duta and T. Grust. 2020. Functional-Style SQL UDFs with a Capital 'F'. In *Proc. SIGMOD*. Portland, OR, USA.
[7] A. Eisenberg and J. Melton. 1999. SQL:1999, Formerly Known as SQL3. *ACM SIGMOD Record* 28, 1 (March 1999).
[8] S.J. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh. 1996. Expressive Recursive Queries in SQL. Joint Technical Committee ISO/IEC JTC 1/SC 21 WG 3, Document X3H2-96-075r1.
[9] B. Glavic and G. Alonso. 2009. PERM: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *Proc. ICDE*. Shanghai, China.
[10] B. Glavic and G. Alonso. 2009. Provenance for Nested Subqueries. In *Proc. EDBT*. Saint Petersburg, Russia.
[11] T.J. Green, G. Karvounarakis, and V. Tannen. 2007. Provenance Semirings. In *Proc. PODS*. Beijing, China.
[12] T. Müller, B. Dietrich, and T. Grust. 2018. You Say 'What', I Hear 'Where' and 'Why'? (Mis-)Interpreting SQL to Derive Fine-Grained Provenance. *PVLDB* 11, 11 (2018).
[13] T. Müller and P. Engel. 2022. How, Where, and Why Data Provenance Improves Query Debugging. In *Proc. ICDE*. Kuala Lumpur, Malaysia.
[14] PostgreSQL [n.d.]. *PostgreSQL 14 Documentation.* `http://www.postgresql.org/docs/14/`.
[15] Y. Ramusat, S. Maniu, and P. Senellart. 2021. A Practical Dynamic Programming Approach to Datalog Provenance Computation. *CoRR* (2021).

[1] `https://github.com/DBatUTuebingen/provenance-for-recursive-queries`