How, Where, and Why Data Provenance Improves Query Debugging

A Visual Demonstration of Fine–Grained Provenance Analysis for SQL

Tobias Müller
University of Tübingen
Tübingen, Germany
to.mueller@uni-tuebingen.de

Pascal Engel
University of Tübingen
Tübingen, Germany
pascal.engel@student.uni-tuebingen.de

***Abstract*—Data provenance is meta–information about the origin and processing history of data. We demonstrate the provenance analysis of SQL queries and use it for query debugging. How–provenance determines which query expressions have been relevant for evaluating selected pieces of output data. Likewise, Where– and Why–provenance determine relevant pieces of input data. The combined provenance notions can be explored visually and interactively. We support a feature–rich SQL dialect with correlated subqueries and focus on bag semantics. Our fine–grained provenance analysis derives individual data provenance for table cells and SQL expressions.**

***Index Terms*—Data Provenance, Databases, Debugging, SQL**

I. Three Notions of Data Provenance

We demonstrate the interactive provenance analysis and its visualization for SQL queries. In short, *data provenance* is *the description of the origins of a piece of data and the process by which it arrived in a database* (Buneman et al. [1, Sec. 1]). Our demonstration shows that data provenance is helpful in understanding and debugging SQL queries. Especially the formulation of complex queries bears the potential for a class of subtle bugs which do not trigger static or dynamic errors but deliver erroneous result values. These bugs are hard to detect. Through data provenance, we can unveil the possibly intricate processes that have produced unexpected or buggy outputs.

This work is focussed on read–only query scenarios as depicted in Figure 1. We make a fundamental distinction between *query logic* and *input data* and map these onto three provenance notions. Below, an intuition for these notions is provided. Cheney et al. [2] provide a comprehensive presentation.

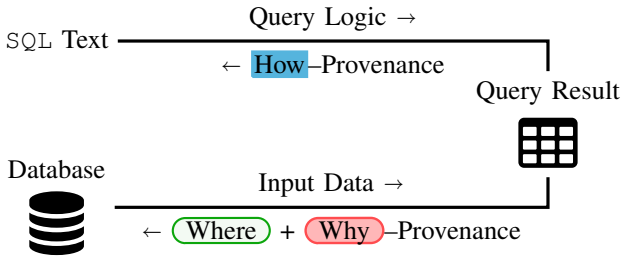
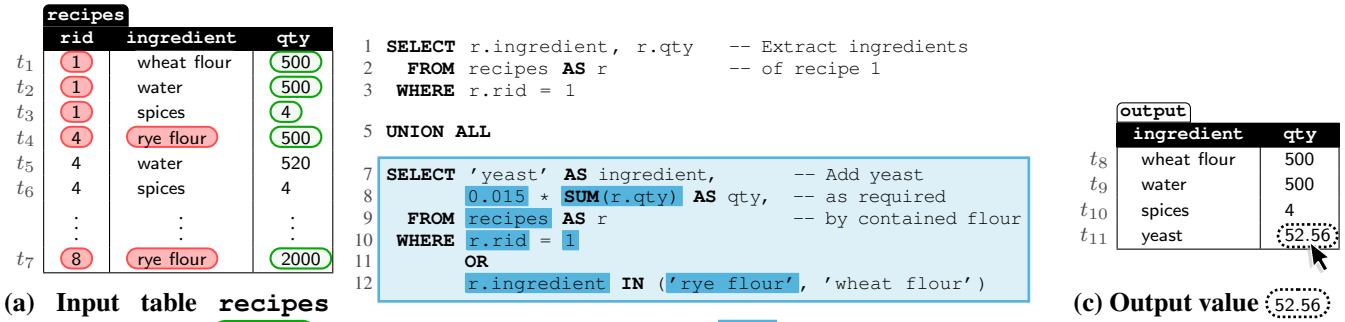


Fig. 1: Query evaluation and associated provenance notions. Data provenance associates pieces of the query result with the relevant input data and SQL expressions.

- How–provenance uncovers the relationship between output data and query logic. The provenance consists of all SQL (sub–)expressions which have been relevant in producing certain output data. We aim at fine–grained How–provenance which analyzes SQL on the level of individual subexpressions.
- Where–provenance identifies the pieces of *input data* directly related to the output data. Input data may just be copied to the output or multiple pieces of input data can be combined to compute one output value. Where–provenance tracks both.
- Why–provenance identifies relevant pieces of *input data* which have been involved in deciding about the existence of output data. For example, this happens when rows have to qualify against **`WHERE`** predicates. Both Where– and Why–provenance track data at the level of individual table cells.

This demonstration integrates all of the above provenance notions. The GUI allows for the interactive exploration of the provenance results. Both data and SQL query are presented on–screen and the data provenance gets highlighted. A distinguishing feature of our approach is that there is no intermediate representation (e.g. relational algebra) the user has to understand. Provenance can be explored by looking directly at the surface language (i.e., SQL expressions just as they were entered) and the relational data. Notable features of the supported SQL dialect are SFWGHO queries with aggregations, correlated subqueries, CTEs, **`IN`**, and **`CASE`** expressions.

Our language–centered approach to How–provenance is inspired by *Program Slicing* by Weiser [7]. Their basic idea is to identify all relevant lines of (imperative) code which determine the contents of a selected variable. In our work, How–provenance determines the SQL expressions which have been relevant in producing a selected output table cell.



(a) Input table `recipes` with highlighted Where– and Why–provenance.

(b) SQL query with highlighted How–provenance. Nesting of subexpressions results in nested provenance.

(c) Output value 52.56 is selected for provenance inspection.

Fig. 2: Debugging example: what ingredients and quantities are required for bread recipe `1`?

How–provenance with similar focus is subject of our short paper [5]. SQL queries are translated into an imperative language and only then the How–provenance is derived. In the present work, this detour via an imperative language is replaced by a SQL–to–SQL rewrite. One advantage is that existing DBMS infrastructure and query planners can be used as the runtime system for provenance analysis. We elaborate on the technical realization in Section III.

The *Provenance Semirings* approach by Green, Karvounarakis, and Tannen [3] is where the term *How–provenance* originates. They develop a formalism (based on exchangeable semirings) that derives tuple–grained provenance for a query dialect inspired by relational algebra (RA). Based on the semirings approach, Senellart, Jachiet, Maniu, and Ramusat [6] demonstrate that How–provenance is practical. However, their work exhibits a disconnect between the SQL expressions (entered by the user) and the How–provenance which gets derived for an RA–based internal query representation. Our work features two main advantages. Firstly, our provenance derivation is cell–granular which improves accuracy. Secondly, we try to avoid the disconnect between the user–facing language and the provenance result. Our How–provenance is derived *directly* for the SQL surface language.

This is the first system to integrate all three notions of provenance, a distinct step forward with respect to earlier work. In the next section, we will present an example query and argue that the combination of all three provenance notions boosts query debugging.

II. How to Read Provenance Annotations

The query in Figure 2(b) extracts the ingredients and quantities for bread recipe 1. Determined by the recipe's flour quantities, the query calculates the specific quantity of yeast. One pound of flour requires $\approx$ 7 grams of yeast, i.e. $7g/500g \approx 0.015$.

The output table in Figure 2(c) consists of three ingredients (rows t_8–t_{10}) copied from the input table plus the computed quantity of yeast. We observe a massive quantity of yeast (ca. 10% of flour quantity). Do not use that recipe or your dough may spill!

We are going to show how the three integrated provenance notions help in uncovering the query's fault. The exploration of this scenario underlines the value of fine–grained data provenance for SQL debugging and also provides an impression of what the demo audience will be able to experience on–site.

In a first step, the suspicious yeast value of 52.56 will probably be in the user's focus. After a click on that value, the system will highlight all three provenance notions associated with this value in response.

Where? The green markers in Figure 2(a) are the Where–provenance of value 52.56. Where–provenance tells us which input values have been contributing directly to that output value. For example, value 500 in input row t_1 has been used to produce 52.56. It is unexpected that in t_2 (ingredient: water) another marker for Where–provenance can be found. The query is supposed to derive the yeast quantity from flour quantities only. Why–provenance can help to drill deeper at this point.

Why? The values of Figure 2(a) marked in red have been inspected in order to decide whether data is included in the query's output. We learn from t_1 that id 1 has been inspected while ingredient wheat flour apparently has been ignored. However, the query is supposed to check all ingredient names to distinguish between flour ingredients and non–flour ingredients. Unexpectedly, the inspection of recipe id 1 already sufficed to decide the inclusion of t_1 in the yeast computation. This is also true for all other rows of recipe 1. This Why–provenance raises the suspicion that something has gone wrong with the predicate evaluation. In t_5 and t_6 we find no Why–provenance because these tuples did not qualify against the **`WHERE`** predicate and were dropped before aggregation.

How? The nested blue boxes in Figure 2(b) constitute the How–provenance and tell us which SQL expressions were relevant in computing the selected output. On the top level, we learn that only the second subquery of **`UNION ALL`** is relevant to the computation of 52.56. On line 8, output column `qty` is derived from the constant value `0.015` and a **`SUM(·)`** aggregate of the `qty` column. That is all unsuspicious so far. Looking at the **`WHERE`** predicate, we find that `r.rid`

and 1 are highlighted. That comparison should already make the Why– and Where–provenance for t_4 impossible. This predicate would throw out any `rid` which is $\neq 1$ — would it not? On line 11, the **WHERE** predicate employs an **OR** expression. Most likely, we have found our bug: an **AND** operator is needed instead of **OR**. After fixing the **OR** bug, the yeast quantity (for 500 grams of wheat flour) indeed changes to $500 \cdot 0.015 = 7.5[g]$.

Short Circuiting. The annotations on line 12 exhibit short circuiting used during the evaluation of the **IN** expression. Successful comparison of data value rye flour (tuples t_4 and t_7) with the literal `'rye flour'` produces Why– and How–provenance. The second comparison with `'wheat flour'` has been skipped (*short circuiting* or *early–out* semantics) or the tuples have not qualified (i.e., t_5 and t_6). Therefore, How–provenance has not been found.

The demo will enable debugging sessions much like the above in which query results and provenance are recomputed interactively.

III. Integrated Provenance Analysis

We have exemplified how query debugging can profit from a seamless integration of multiple provenance notions. This integration is also reflected in the technical realization.

Under the hood, this demonstration is based on a two–step approach of provenance analysis [4]. That approach features compositional rewrite rules $Q \mapsto (Q^{\mathbb{1}}, Q^2)$ which translate a SQL expression Q into a pair of rewritten SQL expressions. The two expressions $Q^{\mathbb{1}}$ and Q^2 implement a *split of responsibilities* which we will address below. Upon evaluation, the two queries yield the data provenance of Q. The compositional and recursive rewrite function $\mapsto$ from [4, Def. 5] facilitates the derivation of Where– and Why–provenance.

In this work, we employ an extended rewrite function $\mapsto^+$ which integrates How–provenance with the existing provenance notions. The formal specification of $\mapsto^+$ is not in the scope of this demonstration. Instead, we are going to exemplify the rewrite using a query fragment from Figure 2(b), line 8. The query fragment is replicated in Figure 3 and identified as Q.

Rewrite function $\mapsto^+$ sticks to the double rewrite policy and creates two new expressions $Q^{\mathbb{1}}$ and Q^2. $Q^{\mathbb{1}}$ operates in the domain of SQL values (just like Q), Q^2 entirely operates in the domain of *sets of provenance annotations* and disregards values. In simple cases (such as this one), we find that $Q^{\mathbb{1}} = Q$ which evaluates to the familiar value 52.56. Additionally, $Q^{\mathbb{1}}$ writes a log of its value–based decisions (e.g., the outcome of **WHERE** predicates). This is essential input for the value–agnostic Q^2 which cannot perform these decisions on its own. The most prominent example of such decisions is the filtering of rows according to predicates. Such a decision (e.g., keep the current row or drop it) is recorded by $Q^{\mathbb{1}}$ and made available to Q^2. When Q^2 is evaluated, it does not perform predicate evaluation and instead specializes on provenance derivation only. For more details, we refer to [4, Sec. 3.1].

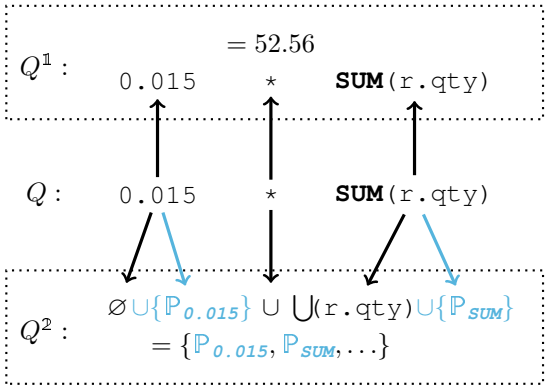


Fig. 3: Example rewrite $Q \mapsto^+ (Q^{\mathbb{1}}, Q^2)$. In $Q^{\mathbb{1}}$, the value semantics is retained. In Q^2, set semantics is employed. Black terms are according to the original rewrite from [4] and other terms derive How–provenance.

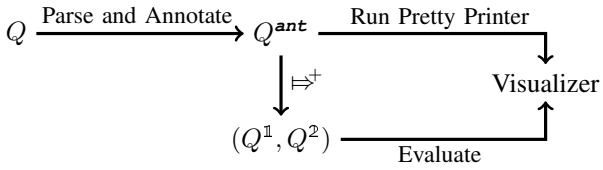


Fig. 4: Overview of the provenance derivation. Query Q is to be analyzed. The rewrite $\mapsto^+$ operates on an AST structure. The visualizer renders the annotated query, the involved tables, and the provenance markers.

Figure 3 illustrates the set of provenance annotations being derived. Semantically, *the data provenance of SQL operator $\star$ is the combined data provenance of its operands*. Rewrites in black are associated with $\mapsto$ while blue rewrites are specific to How–provenance and $\mapsto^+$. The original rewrite of `0.015` produces the empty provenance annotation $\varnothing$ because a literal has no relationship with the input database (as illustrated in Figure 1). However, the How–provenance is non–empty. The singleton set $\{\mathbb{P}_{0.015}\}$ is produced where the identifier $\mathbb{P}_{0.015}$ represents the provenance relationship with the literal SQL subexpression `0.015` on line 8 of the query in Figure 2(b). That item of data provenance will be propagated (together with Where– and Why–provenance) and eventually arrives in the result table of Q^2. Once it does, the visualization will set an according provenance marker 0.015 in the query text. Regarding **SUM**, an analogous provenance identifier $\mathbb{P}_{SUM}$ is employed.

From the viewpoint of How–provenance, literals, and aggregates are treated uniformly. Each expression generates a singleton set carrying a unique provenance identifier $\mathbb{P}$. At run time of Q^2, the combined set $\{\mathbb{P}_{0.015}, \mathbb{P}_{SUM}, \ldots\}$ is derived. In general, these sets contain identifiers for Where– and Why–provenance as well.

A. Software Components

An overview of the demo's toolchain is presented in Figure 4. A SQL query Q (provided by the user via the GUI)

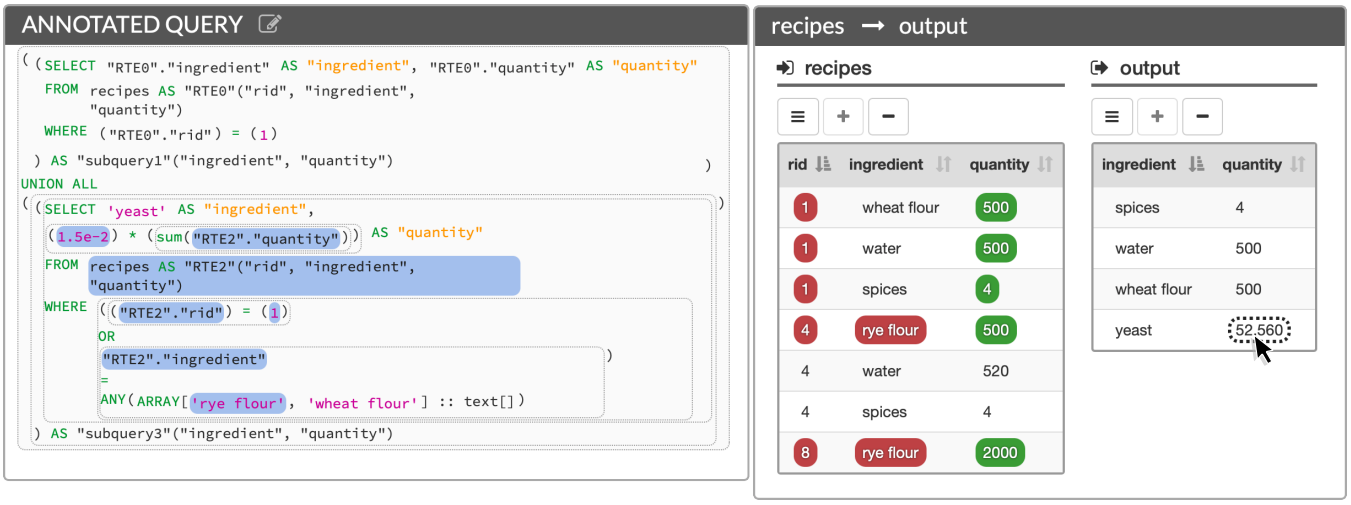


Fig. 5: Screenshot of the interactive demo GUI running in a web browser. More predefined examples are available.

is supposed to be analyzed for its How–, Where–, and Why–provenance. First, the query is parsed into an AST structure and the query expressions are annotated with unique annotations for How–provenance (denoted Q^{ant}). A pretty printer brings the query back on screen in human–readable form. Invisible for the user, each subexpression is tagged with its provenance identifier (cf. $\mathbb{P}_{0.015}$ above). The main rewrite function $\rightarrowtail^{+}$ produces the query pair $(Q^{\mathbb{1}}, Q^{2})$ (as discussed in Section III). Through evaluation of both queries, data provenance is derived. The visualizer combines provenance result and query, i.e. shows the How–provenance.

The DBMS backend for this demo is an unmodified PostgreSQL server in version 12. The DBMS is used for query parsing and evaluation of both $Q^{\mathbb{1}}$ and Q^{2}. The provenance sets in Q^{2} are implemented in terms of PostgreSQL's arrays in which we eliminate duplicates explicitly. Our implementation of the rewrite function $\rightarrowtail^{+}$ is written in Haskell and operates solely on AST structures. The browser–based visualizer queries a tiny local HTTP server. This server bridges between web page, DBMS, and query rewriting.

IV. Demo Setup and Visitor Experience

Figure 5 shows is an authentic screenshot of the GUI and the example query from Section II. On the left, the query and its How–provenance are presented. In comparison to Figure 2, the actual demo supports the rendering of deeply nested expressions with their provenance annotations. The innermost expressions are rendered in a blue background color. Due to the pretty printing step (cf. Figure 4), the syntax is more verbose and the **IN** expression has been desugared using **ANY**.

The entire provenance result is cached, i.e. the GUI is highly responsive. Any output value can be clicked and the according provenance markers show up in an instant. Further, the perspective can be changed at will: it is possible to select input values and query expressions. The dependent output values will be highlighted in response.

If the conference will be held virtually, we can offer ad–hoc meetings to interested participants. Our GUI integrates a list of additional example queries. The participant may ask for a certain example to be openend. Then, its data provenance can be inspected and we can walk the participant through the debugging process. Existing queries can be changed and new queries can be entered if desired. All query rewrite steps are carried out automatically and will not take longer than a few seconds for typical examples.

The annotated and rewritten queries $(Q^{\mathbb{1}}, Q^{2})$ are automatically stored as plain text files and can be presented to interested participants. We can run the main provenance analysis (i.e., Q^{2}) in an open DBMS shell and show the raw, set–based provenance results. The compositional provenance rewrite rules are implemented in Haskell and the source code can be shown and discussed.

Acknowledgment

Martin Lutz and Denis Hirn have contributed to the software components of this demo.

References

[1] P. Buneman, S. Khanna, and W.-C. Tan, "Why and Where: A Characterization of Data Provenance," in *Proc. ICDT*, London, UK, 2001, pp. 316–330.

[2] J. Cheney, L. Chiticariu, and W.-C. Tan, "Provenance in Databases: Why, How, and Where," *Foundations and Trends in Databases*, vol. 1, no. 4, 2007.

[3] T. Green, G. Karvounarakis, and V. Tannen, "Provenance Semirings," in *Proc. PODS*, Beijing, China, 2007, pp. 31–40.

[4] T. Müller, B. Dietrich, and T. Grust, "You Say 'What', I Hear 'Where' and 'Why'? (Mis-)Interpreting SQL to Derive Fine-Grained Provenance," in *Proc. VLDB*, Rio de Janeiro, Brazil, 2018, pp. 1536–1549.

[5] 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.

[6] P. Senellart, L. Jachiet, S. Maniu, and Y. Ramusat, "ProvSQL: Provenance and Probability Management in PostgreSQL," in *Proc. VLDB*, Rio de Janeiro, Brazil, 2018, pp. 2034–2037.

[7] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.