Thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science

Rewriting Enables PL/pgSQL Functions to Derive Their Own Data Provenance

Author:

Thora Daneyko

March 12, 2021

Examiner:

Prof. Dr. Torsten Grust

Supervisor:

Benjamin Dietrich

Eberhard Karls Universität Tübingen

Mathematisch-Naturwissenschaftliche Fakultät

Wilhelm-Schickard-Institut für Informatik

Antiplagiatserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst habe, dass ich keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe, dass ich alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe, dass die Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist, dass ich die Arbeit weder vollständig noch in wesentlichen Teilen bereits veröffentlicht habe, und dass das in Dateiform eingereichte Exemplar mit dem eingereichten gebundenen Exemplar übereinstimmt.

Tübingen, den ____________ __________________________

Thora Daneyko

Abstract

For understanding, debugging and verifying complex SQL queries, it is useful to know *where* a piece of data comes from and *why* it appears in the result. Such *data provenance* is particularly helpful when the SQL dialect is enriched by a procedural programming language, like the PL/pgSQL language of the PostgreSQL RDBMS, since it enables the creation of highly complex functions for retrieving and manipulating data. Building on previous work by Müller et al. (2018), I present a system that rewrites PL/pgSQL functions so that they can collect their own cell-level where- and why-provenance. Unlike other provenance systems, this approach does not require modifications to the underlying RDBMS, since the provenance derivation process is expressed entirely in PL/pgSQL. Also, the structure of the original functions and queries is preserved, limiting the impact on performance.

Contents

Introduction **1**

1 Data Provenance for Relational Databases **4**

1.1 Where-Provenance 5
1.2 Why-Provenance 6
1.2.1 Lineage 7
1.2.2 Witness Basis 8
1.3 Existing Provenance Systems 9

2 Deriving Provenance from Decision Logs **12**

2.1 The Two-Phase Approach 13
2.1.1 Dependency Sets 13
2.1.2 Phase 𝟙: Instrumentation 15
2.1.3 Phase 𝟚: Interpretation 16
2.2 Rewrite Rules 17
2.2.1 Join 18
2.2.2 Order By 18
2.3 Implementation 21
2.3.1 Dependency Sets 22
2.3.2 Logging Functions 23

3 Rewrite Rules for PL/pgSQL Functions **25**

3.1 Function State Variables 26
3.2 Data-Modifying Statements 27
3.2.1 Insertions 28
3.2.2 Updates 29

3.3 Functions 29
3.4 Control Structures 31
3.4.1 Blocks 33
3.4.2 Conditionals 33
3.4.3 Unconditional Loop 33
3.4.4 Foreach Loop 34
3.5 Minor Rewrite Rules 37
3.5.1 Dependency Set Conversion 37
3.5.2 Injecting Cumulative Why-Provenance 37

4 Implementation **40**
4.1 Usage 41
4.1.1 Options 42
4.2 Program Overview 42
4.3 Parsing SQL in Haskell 44
4.3.1 Haskell's Record Syntax 44
4.3.2 Structure of the ASTs 46
4.4 Implementing Rewrite Rules 48
4.4.1 The OperSem Monad 48
4.4.2 Injecting Query Call Sites 49
4.4.3 Where-Provenance 50
4.4.4 Why-Provenance 52

5 Evaluation **56**
5.1 Setup 56
5.2 Results and Discussion 57

6 Conclusion **60**
6.1 Future Work 61

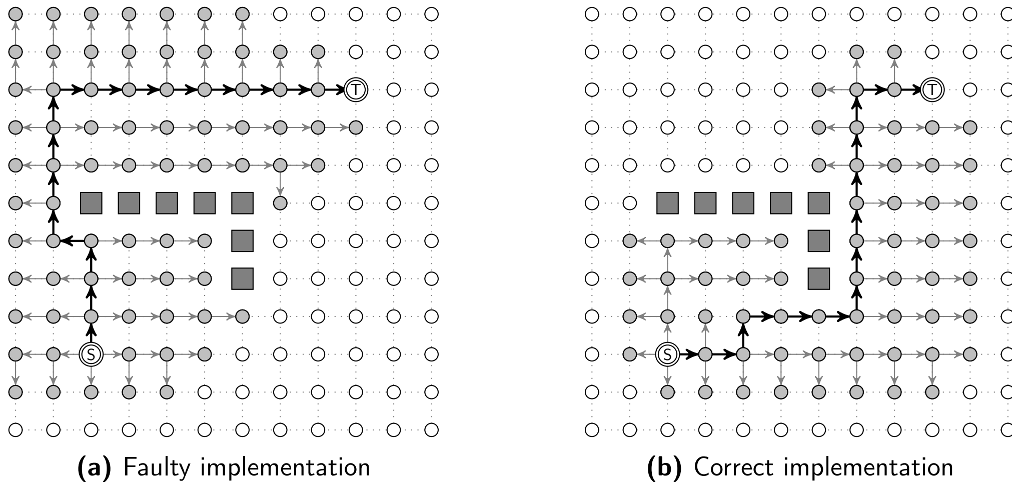
Bibliography **63**

Introduction

Understanding how a data point ended up in the result of a complex SQL query is often challenging by looking at the query alone. This holds especially when SQL is extended by a fully expressive programming language, such as PostgreSQL's PL/pgSQL. Functions written in PL/pgSQL can create, manipulate, and query relational databases using control structures such as conditionals and loops, which makes it particularly difficult to trace the data flow or evaluate the correctness of these functions. Consider a PL/pgSQL implementation of the A* star algorithm for calculating the length of the shortest path between two nodes Ⓢ and Ⓣ in a graph: If it returns 16, how can we verify this result? If we know that the shortest path is actually of length 14, how can we find out how the false value was computed? Which edges and nodes in the graph were touched by the algorithm, which ones actually constituted to the result?

Data provenance derivation, the task of collecting information on the origins of a piece of data and the transformations by which it arrived in the output, offers answers to all of these questions. The *where-provenance* of our return value will tell us exactly of which input cells the result is composed, whereas its *why-provenance* contains all input values that guided the computation, e.g. in `WHERE` clauses or `IF-THEN-ELSE` conditions.

Figure 1a shows the where- and why-provenance of our A* computation. The where-provenance is composed of all edges along the 'shortest' path; their lengths are summed up to compute the total length of the path. The why-provenance consists of all nodes and edges that were explored or considered candidates for the path. We can now clearly see that the result is wrong: The algorithm seems to be dragged to the left, even though that means making a detour around the obstacle, instead of taking the direct way to the right.



(a) Faulty implementation

(b) Correct implementation

Figure 1: PL/pgSQL computation of the shortest path length from Ⓢ to Ⓣ using the A* algorithm, with ■ nodes blocked. Edges ⟶ are in the where-provenance, nodes ● and edges ⟶ in the why-provenance of the result.

There seems to be a problem with the heuristic that is consulted to pick the next candidate edge for exploring the graph. Indeed, there is a copy-paste error in the Manhattan distance function used as a heuristic: It is computed as $|p_2 - q_2| + |p_2 - q_2|$ instead of $|p_1 - q_1| + |p_2 - q_2|$. Once this is corrected, we can inspect the provenance again (Figure 1b) to verify that the function now takes the correct path to compute a shortest distance of 14.

Data provenance thus is not only useful for debugging complex queries and functions, but also helps to build trust and confidence in the results by enabling us to verify the correctness of a query and trace the data flow to identify the source of a piece of data. This is particularly important but also particularly challenging when the SQL dialect is expanded into a full-fledged programming language like PL/pgSQL.

In this thesis, I present a program that rewrites PL/pgSQL functions such that they can derive the where- and why-provenance of their return value alongside the actual result, extending the work of Müller et al. (2018) for regular SQL queries. This approach is non-invasive in that it does not extend or modify the underlying RDBMS, using only PL/pgSQL constructs to derive provenance, and keeps the impact of provenance computation on performance at bay, since the shape of the original function is mostly preserved.

The thesis is structured as follows: Chapter 1 defines data provenance and its subcategories in detail and provides a brief overview over other existing provenance systems. In chapter 2 I then introduce the approach to provenance derivation of Müller et al. (2018) which this thesis aims to extend by adding rewrite rules for PL/pgSQL constructs. Chapter 3 specifies these additional rules. The rewriting system that implements them is presented in chapter 4. Finally, I evaluate the performance impact of the rewritten functions in chapter 5.

1 Data Provenance for Relational Databases

The term *data provenance* refers to the origins of a piece of data d and the derivation and transformation process that led to its presence in the current view or output. We typically distinguish between *where-provenance*, the locations from which d was copied, *why-provenance*, the data that contributed to d's presence in the output, and *how-provenance*, the transformation process by which d was derived (Cheney et al. 2009; Herschel et al. 2017).

In the context of relational databases, d is a tuple or attribute in the result of a query q. Where-provenance then refers to those cells in the source relations of which d's value is composed, why-provenance to the tuples or attributes that q touched upon to derive d, and how-provenance to the structure of q itself.

Provenance may be computed *eagerly* during the evaluation of the query and stored for later inspection, or *lazily* at the user's request any time after the actual query happened (Cheney et al. 2009). It can be captured at *tuple-based* or *attribute-based granularity*, i.e. we can link output tuples back to input tuples, or output attributes back to input attributes, deriving a more fine-grained provenance.

In the following sections, I introduce the notions of where- and why-provenance in more detail (sections 1.1 and 1.2), and give an overview over existing systems for deriving and investigating data provenance (section 1.3). I do not delve further into how-provenance or other types of data provenance, since this thesis is only concerned with where- and why-provenance.

students

	sid	name	ects
ρ_1	1	Mary	150
ρ_2	2	Peter	84
ρ_3	3	Ann	36

enrolled

	sid	cid
ρ_8	1	ASW-BA-09
ρ_9	1	ISCL-MA-05
ρ_{10}	1	INF4140
ρ_{11}	3	INF3212
ρ_{12}	3	INF4140

classes

	cid	name	ects
ρ_4	ASW-BA-09	Loanword Phonology	12
ρ_5	INF3212	Functional Programming	6
ρ_6	INF4140	Advanced SQL	9
ρ_7	ISCL-MA-05	Computational Lexicography	6

Figure 1.1: A database modeling students with their ECTS balance and the classes they are currently enrolled in. Tuples (or rows) are uniquely identified as ρ_i.

```sql
SELECT s.sid, s.name
FROM students AS s,
     classes AS c,
     enrolled AS e
WHERE s.sid = e.sid
  AND e.cid = c.cid
  AND c.name = 'Advanced SQL';
```

$\Rightarrow$

	sid	name
ρ'_1	1	Mary
ρ'_2	3	Ann

Figure 1.2: A query retrieving all students currently enrolled in the class “Advanced SQL”.

1.1 Where-Provenance

The term *where-provenance* was coined by Buneman et al. (2001). Where-provenance is concerned with the source *locations* from which an output value is composed, by copying or transforming the values stored at these locations (Cheney et al. 2009; Herschel et al. 2017).

Consider, for instance, the database in Figure 1.1 describing university students and classes, and the query in Figure 1.2 that retrieves the students attending the “Advanced SQL” class. It produces two output tuples, $\rho'_1 = \{\text{sid} : 1, \text{name} : \text{Mary}\}$ and $\rho'_2 = \{\text{sid} : 3, \text{name} : \text{Ann}\}$. At tuple-based granularity, the where-provenance of ρ'_1 consists only of $\rho_1 = \{\text{sid} : 1, \text{name} : \text{Mary}, \text{ects} : 150\}$ from the students

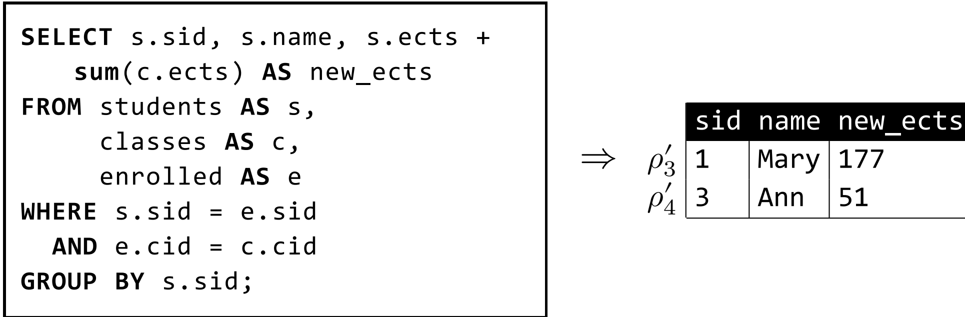


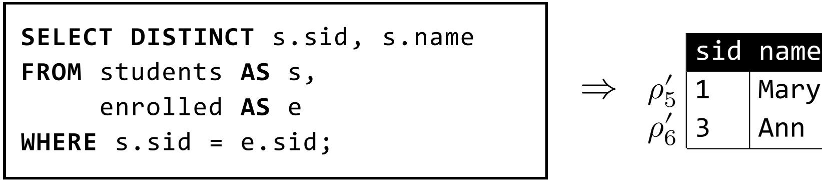
Figure 1.3: A query computing the new total ECTS for each active student after the current semester.

relation, since this is where its values were copied from. All of the other tuples touched by this query, like those from the classes or enrolled relation, are not part of the where-provenance, since they did not directly contribute to the value of ρ'_1. Similarly, at attribute-based granularity, the where-provenance of ρ'_1.sid is $\{\rho_1.\text{sid}\}$ and that of ρ'_1.name is $\{\rho_1.\text{name}\}$.

Where-provenance becomes more interesting once the query involves data transformations. The aggregate query in Figure 1.3 computes the new total of ECTS points for students currently attending classes. The where-provenance of ρ'_3.new_ects, Mary's new ECTS total, is $\{\rho_1.\text{ects}, \rho_4.\text{ects}, \rho_6.\text{ects}, \rho_7.\text{ects}\}$, i.e. the locations of Mary's previous ECTS total and the ECTS points gained by attending the courses "Loanword Phonology", "Advanced SQL", and "Computational Lexicography". Similarly, the where-provenance of the A* example from Figure 1 in the introduction consists of the distance attributes of all edges in the shortest path. For complex queries, where-provenance can thus provide valuable hints as to whether the result is computed correctly.

1.2 Why-Provenance

In contrast to where-provenance, why-provenance is concerned with the input *values* that led to a tuple's presence in the query result. In general, the why-provenance of a tuple (or attribute) t' are those tuples (or attributes) that need to be present in the input for t' to appear in the output (Cheney et al. 2009; Herschel et al. 2017). The why-provenance of the results in Figures 1.2 and 1.3 will therefore also contain the tuples (or attributes) referenced in the `WHERE` clauses. There



```
SELECT DISTINCT s.sid, s.name
FROM students AS s,
     enrolled AS e
WHERE s.sid = e.sid;
```

⇒

	sid	name
ρ'_5	1	Mary
ρ'_6	3	Ann

Figure 1.4: A query retrieving all students currently enrolled any class.

are two different definitions of why-provenance commonly cited in the literature: The *lineage* (or *derivation*) type introduced by Cui et al. (2000) and the *witness basis* type first described by Buneman et al. (2001).

1.2.1 Lineage

In their 2000 paper, Cui et al. define the *tuple derivation* (also referred to as *lineage*) as "the set of base relation tuples that produce a given view tuple" (p. 185). More formally:

Definition 1.2.1 (adapted from Cui et al. (2000), definition 4.1)**.** Given base relations $R_1, \dots, R_n$[1] and a query q over those relations, the lineage of a tuple $t \in q(R_1, \dots, R_n)$ is $q^{-1}_{R_1,\dots,R_n}(\{t\}) = \langle R'_1, \dots, R'_n \rangle$, such that

(1) R'_i is a maximal subset of R_i

(2) $q(R'_1, \dots, R'_n) = \{t\}$

(3) $\forall t' \in R'_i, 1 \leq i \leq n : q(R'_1, \dots, \{t'\}, \dots, R'_n) \neq \emptyset$

This means that the lineage of t is the set of all (1) base relation tuples which, given as input to q, produce exactly t as output (2), with no base tuple being irrelevant to the production of t (3) (Cheney et al. 2009).

Figure 1.4 shows a query selecting all students that take a class during this semester. The lineage of ρ'_5 is $\{\rho_1, \rho_8, \rho_9, \rho_{10}\}$, i.e. the entry of Mary in the students relation and all tuples referencing Mary taking a class in the enrolled relation. This set satisfies the conditions of definition 1.2.1:

(1) No other tuple from students or enrolled would contribute anything to the result.

[1] In case of a self-join of a relation R, this relation will count as two equal base relations R_i and R_j (Cheney et al. 2009).

(2) Running the query on a database reduced to just those four tuples would yield exactly ρ'_5.

(3) If Mary only took a single class, i.e. if we would only have one of $\{\rho_8, \rho_9, \rho_{10}\}$ in our lineage, ρ'_5 would still be in the result.

Note that while condition (3) makes sure that each tuple t in the lineage is *relevant* to the result, t might not be *necessary*, because the result is also guaranteed by other tuples from the same relation (Cheney et al. 2009). In our example, subsets of the lineage, like $\{\rho_1, \rho_8\}$ or $\{\rho_1, \rho_9, \rho_{10}\}$, would suffice for producing ρ'_5. It is not necessary for Mary to attend all three courses in order for her to be considered an active student. This issue is resolved by the witness basis definition of why-provenance of Buneman et al. (2001).

1.2.2 Witness Basis

Buneman et al. (2001) were the first to introduce the term 'why-provenance', though it is often used for both their definition and Cui et al.'s (2000) data lineage. To avoid confusion, I will therefore explicitly refer to Buneman et al.'s definition as the *witness basis* type of why-provenance in this chapter.

According to Buneman et al. (2001), the why-provenance of a tuple t consists of *witnesses* for t, where a witness is "the collection of values taken from D [the input database] that proves an output" (p. 8). More formally:

Definition 1.2.2 (adapted from Buneman et al. (2001), p. 8)**.** Given base relations $R_1, \dots, R_n$ and a query q over those relations, the witness of a tuple $t \in q(R_1, \dots, R_n)$ are the relations $\langle R'_1, \dots, R'_n \rangle$, such that $t \in q(R'_1, \dots, R'_n)$.

This definition is similar to the definition 1.2.1 for lineage, and indeed, the lineage of t is also the witness of t. However, it poses no restriction on the number or relevance of tuples in the witness. In the case of Figure 1.4, both the entire input database and subsets of the lineage like $\{\rho_1, \rho_8\}$ or $\{\rho_1, \rho_9, \rho_{10}\}$ are also witnesses of ρ'_5.

The why-provenance is a specific subset of the set of all witnesses of t which is called the *witness basis*:

Definition 1.2.3 (adapted from Buneman et al. (2001), section 5.1)**.** The witness

basis of $t \in q(R_1, \ldots, R_n)$ is the set $\mathcal{W}(t)$ such that $\forall W \in \mathcal{W}(t)$:

(1) W is a witness for t

(2) The values in W correspond to the leaves of the Datalog "proof tree" for t

As Cheney et al. (2009) point out, the "proof tree" of a Datalog query corresponds to the operator tree of a relational algebra query. Since the leaves of an operator tree in turn correspond to the input relations of the query, this basically means that no two tuples in a member of the witness basis may come from the same base relation. Thus, (2) can be reformulated as:

(2)* $\forall w_1, w_2 \in W : w_1 \neq w_2 \wedge w_1 \in R_i \Rightarrow w_2 \notin R_i$

Going back to the query in Figure 1.4, this means that the witness basis for ρ'_5 is $\{\{\rho_1, \rho_8\}, \{\rho_1, \rho_9\}, \{\rho_1, \rho_{10}\}\}$, i.e. all combinations of Mary's entry in the `students` relation and a class she visits from the `enrolled` relation. This captures the intuition that her visiting any single class is sufficient for her appearance in the 'active students' query result (cf. Cheney et al. 2009)

1.3 Existing Provenance Systems

One of the earlier applications supporting provenance computation is *DBNotes*, a system for attaching annotations to the attributes in a relational database (Chiticariu et al. 2004). These annotations are eagerly propagated to views and query results, and can be selected inside a query as a relation in the `FROM` clause. Attribute-based where-provenance is automatically recorded as annotations as well, exploiting the propagation mechanism already in place. Using these provenance annotations, DBNotes generates out detailed explanations for a query result as well as diagrams visualizing the provenance for a specific output attribute and the flow of an input attribute into different views and query results. It does only support a "fragment of SQL" (Chiticariu et al. 2004, p. 1), however. DBNotes is a Java program sending rewritten queries to an underlying relational database management system (RDBMS).

The *Trio* database management system for *uncertainty-lineage databases* (ULDBs) comes with eager tuple-based lineage why-provenance computation (Benjelloun et al. 2006). It is a Python program extending the standard Python DB 2.0 API

for sending queries in its custom SQL dialect (TriQL) to a PostgreSQL database. Lineage information is stored in separate relations and can be used as a join condition in the `WHERE` clause of a query. The supported SQL constructs are also rather limited, since Trio only allows non-nested `SELECT-FROM-WHERE` queries without aggregation or `DISTINCT`.

A more extensive approach to provenance processing is the *Perm* system (Glavic and Alonso 2009). It provides tuple-based witness-base why-provenance "for the complete SQL language except correlated subqueries" (p. 175). To eagerly compute provenance, the result relation is extended with provenance attributes for all attributes in each of the base relations. This allows the provenance tuples to be stored directly inside the result relation as a side effect of the rewritten query, but also leads to an exponential growth in result size for some set operations, since the output contains one tuple per query result tuple t and witness in the witness base of t. Perm's implementation is also tightly coupled with the PostgreSQL RDBMS, since the provenance rewriter sits as an extension module below the regular PostgreSQL query rewriter and requires modifications of the PostgreSQL parser and analyzer. Thus, Perm can only be used together with PostgreSQL and is sensitive to changes in the RDBMS.

In contrast, the *GProM* system takes an entirely non-invasive approach, supporting both SQL and Datalog queries as input and a variety of RDBMS backends (Arab et al. 2018). Similar to Perm, GProM uses a query rewriter to store witness-base why-provenance information in additional attributes while performing the query. In addition to regular queries, GProM also supports updates and transactions, and can compute why-not provenance, i.e. explanations for why an expected tuple is missing from the result.

Another recent approach to provenance computation is *ProvSQL*, a non-invasive PostgreSQL extension that supports various types of provenance, among those where-provenance and both lineage and witness-base why-provenance (Senellart et al. 2018). When applied to a database, it assigns a unique provenance id to every tuple in the affected relations. The rewriting module it inserts between PostgreSQL's parser and planner then rewrites an input query to build a 'provenance circuit' referencing the tuples via their ids. A number of functions are provided for inspecting and accessing provenance inside queries. ProvSQL covers

many SQL constructs including nested queries and set operations (but excluding aggregation).

Finally, Müller et al. (2018) introduce a set of rewrite rules for various SQL constructs not covered by the other approaches, among them window aggregation, recursive queries, and user-defined functions, that log the value-based decisions of the query in one phase to assemble attribute-based where-provenance and (optionally) lineage in a second phase. Their query logs use considerably less space than Perm's explicit provenance representations, which leads to a much smaller overhead on execution times. Since this thesis aims at extending their rewrite rule set with rules for PL/pgSQL constructs, the next chapter elaborates their approach in more detail.

2 Deriving Provenance from Decision Logs

While most of the provenance systems introduced in the previous chapter either directly deliver provenance together with the query result in a single rewritten query (Perm, GProM) or store the provenance itself in a relation as a side effect of the query for later inspection (DBNotes, Trio, ProvSQL), Müller et al. (2018) instead log the value-based decisions of the query to derive provenance by retracing these decisions in a second step. The two queries required for these two phases can be derived directly from the input query via a number of compositional rewrite rules and are expressed in regular SQL. Thus, their approach does not require any modifications to the underlying RDBMS.

Müller et al. (2018) derive both where-provenance and lineage-type why-provenance. They opted for the lineage rather than the witness basis definition due to its smaller size, which is more manageable for complex queries. From this point on, I will use the term 'why-provenance' to refer specifically to the lineage type, since this is the only type of why-provenance that is treated in the remainder of this thesis.

In this chapter, I first elaborate the two phases of their approach in greater detail (section 2.1). I then explain the rewrite rules relevant for this work (section 2.2). Finally, I introduce the implementation of their logging system and provenance set type used in this thesis (section 2.3).

2.1 The Two-Phase Approach

One of the main goals of Müller et al.'s (2018) approach is to design rewrite rules that preserve both

(a) the shape of the query, and

(b) the shape of the query result.

This has the benefit of largely retaining the query plan of the original query, keeping the overhead for the query processor at bay (Müller et al. 2018, p. 3). Hence, a method that alters the result relation, e.g. by duplicating rows and appending provenance attributes, as with Perm and GProM, or considerably changes the structure of the query is not suitable.

The authors therefore decide to split provenance derivation into two phases. *Phase* $\mathbb{1}$ turns input query q into an *instrumented* query $q^{\mathbb{1}}$ which computes and returns the exact same result as q, but as a side effect writes logs about the value-based decisions it makes during each subquery. In *Phase* $\mathbb{2}$, they derive *interpreter* $q^{\mathbb{2}}$ from q, which uses these logs to derive the provenance for q (Müller et al. 2018, p. 2). Thus, the user can use $q^{\mathbb{1}}$ to perform the actual query, and then later process $q^{\mathbb{2}}$ to inspect provenance.

2.1.1 Dependency Sets

While the output of Phase $\mathbb{1}$ is that of the original query, Phase $\mathbb{2}$ should return the provenance of the query result in the form of *dependency sets* (or *provenance sets*). Unlike Perm and GProM, which directly store duplicates of the original attributes in their provenance attributes, Müller et al. (2018) assign unique identifiers ρ_i to the tuples of every relation and use these for reference in their provenance output (similar to ProvSQL). These *tuple identifiers* are not only unique within a single relation, but within the whole database, so that any tuple id also encodes the relation in which the tuple lives. We have already used such ids to refer to individual tuples of the students and classes database example in Figure 1.1, repeated here in Figure 2.1. As a preparation for Phase $\mathbb{1}$, these tuple ids now need to be added as additional attributes to the relations that participate in our target query.

students

	sid	name	ects
ρ_1	$1_{\rho_{1,1}}$	Mary$_{\rho_{1,2}}$	$150_{\rho_{1,3}}$
ρ_2	$2_{\rho_{2,1}}$	Peter$_{\rho_{2,2}}$	$84_{\rho_{2,3}}$
ρ_3	$3_{\rho_{3,1}}$	Ann$_{\rho_{3,2}}$	$36_{\rho_{3,3}}$

enrolled

	sid	cid
ρ_8	$1_{\rho_{8,1}}$	ASW-BA-09$_{\rho_{8,2}}$
ρ_9	$1_{\rho_{9,1}}$	ISCL-MA-05$_{\rho_{9,2}}$
ρ_{10}	$1_{\rho_{10,1}}$	INF4140$_{\rho_{10,2}}$
ρ_{11}	$3_{\rho_{11,1}}$	INF3212$_{\rho_{11,2}}$
ρ_{12}	$3_{\rho_{12,1}}$	INF4140$_{\rho_{12,2}}$

classes

	cid	name	ects
ρ_4	ASW-BA-09$_{\rho_{4,1}}$	Loanword Phonology$_{\rho_{4,2}}$	$12_{\rho_{4,3}}$
ρ_5	INF3212$_{\rho_{5,1}}$	Functional Programming$_{\rho_{5,2}}$	$6_{\rho_{5,3}}$
ρ_6	INF4140$_{\rho_{6,1}}$	Advanced SQL$_{\rho_{6,2}}$	$9_{\rho_{6,3}}$
ρ_7	ISCL-MA-05$_{\rho_{7,1}}$	Computational Lexicography$_{\rho_{7,2}}$	$6_{\rho_{7,3}}$

Figure 2.1: A database modeling students and classes, repeated from Figure 1.1 and annotated with explicit cell identifiers $\rho_{i,j}$.

```
SELECT s.sid, s.name
FROM students AS s,
     classes AS c,
     enrolled AS e
WHERE s.sid = e.sid
  AND e.cid = c.cid
  AND c.name = 'Advanced SQL';
```

$\Rightarrow$

	sid	name
ρ'_1	1	Mary
ρ'_2	3	Ann

Figure 2.2: A query retrieving all students currently enrolled in the class "Advanced SQL", repeated from Figure 1.2.

Since we want to derive attribute-based provenance, we additionally need a way to unambiguously refer to individual cells. Thus, for any relation R with attributes $a_1, \dots, a_n$ and any tuple $\rho_i \in R$, we use the cell id $\rho_{i,j}$ to refer to $\rho_i.a_j$. This is also illustrated in Figure 2.1.

A dependency set $P \in \mathbb{P}$ for attribute $\rho_{k,l}$, where $\mathbb{P}$ is the type of dependency sets, then is a set of cell ids referring to those attributes that form the provenance of $\rho_{k,l}$ (cf. Müller et al. 2018, Definition 1). We further use $Y(\rho_{i,j})$ inside dependency sets to mark a cell $\rho_{i,j}$ as being part of the why-provenance. Since $Y(\rho_{i,j}) \neq \rho_{i,j}$, both $\rho_{i,j}$ and $Y(\rho_{i,j})$ can be part of the same dependency set if $\rho_{i,j}$ contributes to both where- and why-provenance.

```
SELECT write_JOIN⟨3⟩(ℓ, φ_v, s.ρ, c.ρ, e.ρ) AS ρ,
       s.sid, s.name
FROM   students AS s,
       classes AS c,
       enrolled AS e
WHERE  s.sid = e.sid
  AND  e.cid = c.cid
  AND  c.name = 'Advanced SQL';
```

Figure 2.3: The Phase 𝟙 rewrite of the query from Figure 2.2.

Consider, for instance, the query from Figure 1.2, repeated here in Figure 2.2. The attribute-based where-provenance of ρ'_1`.sid` is only ρ_1`.sid` ($\rho_{1,1}$), whereas the why-provenance consists of all cells referenced in the where-clause for ρ_1`.sid`, namely ρ_1`.sid` itself ($\rho_{1,1}$) plus ρ_{10}`.sid` ($\rho_{10,1}$), ρ_{10}`.cid` ($\rho_{10,2}$), ρ_6`.cid` ($\rho_{6,1}$), and ρ_6`.name` ($\rho_{6,2}$). Thus, the dependency set for ρ'_1`.sid` is $\{\rho_{1,1}$, $Y(\rho_{1,1})$, $Y(\rho_{10,1})$, $Y(\rho_{10,2})$, $Y(\rho_{6,1})$, $Y(\rho_{6,2})\}$.

2.1.2 Phase 𝟙: Instrumentation

In Phase 𝟙, the original query is rewritten in a way that preserves its structure, but as a side-effect logs the value-based decisions made during the evaluation of that query. Figure 2.3 shows the Phase 𝟙 version of our example from the previous section. It is basically the same query as before, but selects an additional column ρ by calling the logging function $\mathit{write}_{\texttt{JOIN}\langle 3\rangle}$.

The exact functionality of these logging functions is left unspecified in the main body of Müller et al.'s (2018) article, but we will discuss an SQL implementation of logging in section 2.3. For now, it suffices to note that a logging function $\mathit{write}_{\texttt{X}}$ takes two arguments ℓ and φ_v plus a number of values encoding the particular decision `X`. In our example, we use $\mathit{write}_{\texttt{JOIN}\langle 3\rangle}$ to log the join of three tuples via their respective tuple identifiers. The logging function's return value is a new tuple id ρ for the result tuple.

The values ℓ and φ_v refer to the *call sites* of the logging function. ℓ, which is the sole call site employed by Müller et al. (2018), will be termed the *query call site* in this thesis. In a complex nested query, its value is used to uniquely identify each

students_2

	sid	name	ects
ρ_1	$\{\rho_{1,1}\}$	$\{\rho_{1,2}\}$	$\{\rho_{1,3}\}$
ρ_2	$\{\rho_{2,1}\}$	$\{\rho_{2,2}\}$	$\{\rho_{2,3}\}$
ρ_3	$\{\rho_{3,1}\}$	$\{\rho_{3,2}\}$	$\{\rho_{3,3}\}$

classes_2

	cid	name	ects
ρ_4	$\{\rho_{4,1}\}$	$\{\rho_{4,2}\}$	$\{\rho_{4,3}\}$
ρ_5	$\{\rho_{5,1}\}$	$\{\rho_{5,2}\}$	$\{\rho_{5,3}\}$
ρ_6	$\{\rho_{6,1}\}$	$\{\rho_{6,2}\}$	$\{\rho_{6,3}\}$
ρ_7	$\{\rho_{7,1}\}$	$\{\rho_{7,2}\}$	$\{\rho_{7,3}\}$

enrolled_2

	sid	cid
ρ_8	$\{\rho_{8,1}\}$	$\{\rho_{8,2}\}$
ρ_9	$\{\rho_{9,1}\}$	$\{\rho_{9,2}\}$
ρ_{10}	$\{\rho_{10,1}\}$	$\{\rho_{10,2}\}$
ρ_{11}	$\{\rho_{11,1}\}$	$\{\rho_{11,2}\}$
ρ_{12}	$\{\rho_{12,1}\}$	$\{\rho_{12,2}\}$

Figure 2.4: The Phase 𝟚 version of the students and classes database from Figure 2.1.

```
SELECT t.ρ, s.sid ∪ y.y, s.name ∪ y.y
FROM   students_2 AS s,
       classes_2 AS c,
       enrolled_2 AS e,
       LATERAL read_JOIN⟨3⟩(ℓ, φ_v, s.ρ, c.ρ, e.ρ) AS t(ρ),
       LATERAL Y(s.sid ∪ e.sid ∪ e.cid ∪ c.cid ∪ c.name)
           AS y(y);
```

Figure 2.5: The Phase 𝟚 rewrite of the query from Figure 2.2. The grayed out parts optionally compute why-provenance.

subquery. I additionally introduce the *function call site* φ_v which is used inside a PL/pgSQL function definition to disambiguate between multiple executions of the same query call site (e.g. in loops). φ_v will be discussed in more detail in the next chapter.

2.1.3 Phase 𝟚: Interpretation

Phase 𝟚 operates on dependency sets, not values. In order to still preserve the basic query structure, we create copies $t_i^{𝟚}$ of the base relations t_i accessed by the original query, where every value (except the tuple identifier) is replaced by a dependency set containing only the corresponding cell id from t_i. Put differently, every value is turned into its own where-provenance. The Phase 𝟚 version of the relations from our example are shown in Figure 2.4.

We can then rewrite the input query into a similar query operating on the Phase 𝟚 copies of the original relations that consults the logs written in the previous phase to assemble where- and why-provenance. Figure 2.5 shows the rewritten version of our example query. The `WHERE` clause has been dropped, since filtering is now

performed by the logging function, but the `SELECT` and `FROM` clauses are still mostly preserved.

We still select the `s.sid` and `s.name` cells, albeit from the Phase 2 relations. Hence, we select dependency sets, not values. A lateral call to the logging function $read_{\texttt{JOIN}\langle 3 \rangle}$ with the current call sites and the tuple identifiers of our participating tuples is also joined to the result. This has a filtering effect: If the tuples were logged, i.e. passed the `WHERE` clause of the original query, the lateral join simply adds the tuple id of the joined tuple. If, however, the tuples did not fulfill the `WHERE` condition, they do not exist in the log, the logging function returns an empty relation, and the tuples will not appear in the result. Thus, the logging function reflects the value-based decision made by the original query.

This alone collects the where-provenance of the query: Since the result values were simply copied from the base relation `students`, a copy of the values of the Phase 2 relation `students_2` produces the 'where' dependency sets for the result. In addition, we can derive why-provenance by unioning the dependency sets of the cells participating in the original `WHERE` clause and adding the resulting dependency set to our where-provenance. As why-provenance sets tend to be significantly larger than where-provenance sets, Müller et al. (2018) design their rewrite rules such that why-provenance derivation is an optional add-on. Thus, in Figure 2.5, those parts of the query responsible for assembling why-provenance are colored gray to show that they can be left out to reduce the computation cost.

2.2 Rewrite Rules

The previous section already went through an example of the rewrite rule for joins. In this section, we now take a closer look at the prerequisites for rewrite rules and the formal notation of those two that are also implemented by this thesis, the Join and OrderBy rules.

Before the rewrite rules may be applied to the input query, it has to be *normalized*. During normalization, a complex query is decomposed into several nested subqueries, such that every subquery of the normalized query is either

- a (possibly conditional) join of one or more tables,

- a query with a `GROUP BY` clause, possibly with `HAVING` and aggregation,
- a window aggregation,
- a `DISTINCT` selection, possibly with an `ORDER BY` clause, or
- an `OFFSET` and/or `LIMIT` selection with an `ORDER BY`.

Any query except the explicit join may only refer to a single relation in its `FROM` clause. This *clause isolation* ensures that rewrite rules do not need to take too many optional constructs into account and any subquery is covered by a single rewrite rule only. In addition, syntactic sugar is removed to make subqueries *explicit*, e.g. by replacing the wildcard `*` in any `SELECT` clause by the actual column references. Again, this reduces the number of query variants that the rewrite rules need to cover (Müller et al. 2018, p. 5f).

With normalization taken care of, the rewrite rules are formalized as compositional bottom-up inference rules $i \Rightarrow \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle$ that transform a normalized query i into the instrumented query $i^{\mathbb{1}}$ and the interpreter $i^{\mathbb{2}}$ (Müller et al. 2018, p. 6).

2.2.1 Join

Figure 2.6 shows the rewrite rule Join for simple joins of $m \geq 1$ relations that we have already seen in action in section 2.1. The transformation $i \Rightarrow \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle$ first requires the recursive transformation of the `SELECT` expressions e_i as well as the `FROM` expressions q_i and the `WHERE` condition p into their Phase $\mathbb{1}$ and $\mathbb{2}$ equivalents. It also retrieves a unique query call site id ℓ via $site()$ that it can insert into both $i^{\mathbb{1}}$ and $i^{\mathbb{2}}$ to link their logging function calls together. The function call site φ_v works a bit differently, as we will see in the next chapter. These values are then used to construct instrumented query and interpreter (with optional why-provenance) as discussed in the previous section.

2.2.2 Order By

The OrderBy rule in Figure 2.7 takes a query with a single `FROM` expression q (as ensured by normalization) and `ORDER BY`, `OFFSET`, and `LIMIT` clauses. Just as Join, it retrieves the call sites and recursively transforms `SELECT` expressions e_i, q, and the ordering criteria o_i.

$$\varphi_v$$
$$\ell = site()$$
$$e_i \Rightarrow \langle e_i^{\mathbb{1}}, e_i^{\mathbb{2}} \rangle \ \ \forall i = 1, \dots, n$$
$$q_i \Rightarrow \langle q_i^{\mathbb{1}}, q_i^{\mathbb{2}} \rangle \ \ \forall i = 1, \dots, m$$
$$p \Rightarrow \langle p^{\mathbb{1}}, p^{\mathbb{2}} \rangle$$

$i^{\mathbb{1}} =$

```
SELECT write_JOIN⟨m⟩(ℓ, φ_v, t_1.ρ, …, t_m.ρ) AS ρ,
       e_1^1, …, e_n^1
FROM   q_1^1 AS t_1, …, q_m^1 AS t_m
WHERE  p^1
```

$i^{\mathbb{2}} =$

```
SELECT t_JOIN.ρ,
       e_1^2 ∪ Y.y, …, e_n^2 ∪ Y.y
FROM   q_1^2 AS t_1, …, q_m^2 AS t_m
       LATERAL read_JOIN⟨m⟩(ℓ, φ_v, t_1.ρ, …, t_m.ρ) AS t_JOIN(ρ),
       LATERAL Y(p^2) AS Y(y)
```

(Join)

```
SELECT e_1, …, e_n
FROM   q_1, …, q_m      ⇒ ⟨i^1, i^2⟩
WHERE  p
```

Figure 2.6: The rewrite rule for m-fold joins, adapted from Müller et al. (2018), Figure 12.

Müller et al. (2018, p. 15f) point out that the combination of `ORDER BY` with `OFFSET` and/or `LIMIT` acts like a filter, throwing away tuples that do not fall into the specified range. Thus, Phase $\mathbb{1}$ has the original filtering clause in a subquery and simply logs the tuple identifiers of the surviving rows. In Phase $\mathbb{2}$, the filter effect is again taken over by the $read_{\texttt{FILTER}}$ logging function, just as with Join. The order of the result tuples is irrelevant here, so the `ORDER BY` clause is dropped.

My adaption of the OrderBy rule differs from the original of Müller et al. (2018) in one crucial aspect: The computation of why-provenance. They construct a lateral join with the union of all ordering expressions o_i, just as in Join with p, and add the resulting dependency set as why-provenance to the result sets. However, this implies that the why-provenance is only evaluated with respect to the current tuple. This goes against the intuition that in an `ORDER BY` clause, all tuples in the relation should contribute to the why-provenance, since they are all compared against each other, so that each tuple influences the position of another and, ultimately, which tuples pass the filter and appear in the result.

Consider the left query in Figure 2.8, for instance. It uses a combination of `OR-`

$$
\frac{
\begin{array}{l}
\varphi_v \\
\ell = site() \\
e_i \Rightarrow \langle e_i^{1}, e_i^{2} \rangle \ \ \forall i = 1, \dots, n \\
q \Rightarrow \langle q^{1}, q^{2} \rangle \\
o_i \Rightarrow \langle o_i^{1}, o_i^{2} \rangle \ \ \forall i = 1, \dots, m \\
i^{1} = \begin{array}{ll}
\texttt{SELECT} & write_{\texttt{FILTER}}(\ell,\ \varphi_v,\ t.\rho)\ \texttt{AS}\ \rho, \\
 & t.c_1,\ \dots,\ t.c_n \\
\texttt{FROM} & (\texttt{SELECT}\ \ t.\rho\ \texttt{AS}\ \rho,\ e_1^{1}\ \texttt{AS}\ c_1,\ \dots,\ e_n^{1}\ \texttt{AS}\ c_n \\
 & \ \ \texttt{FROM}\ \ q^{1}\ \texttt{AS}\ t \\
 & \ \ \texttt{ORDER BY}\ o_1^{1},\ \dots,\ o_m^{1} \\
 & \ \ \texttt{OFFSET}\ \ k \\
 & \ \ \texttt{LIMIT}\ \ l)\ \texttt{AS}\ t
\end{array} \\
i^{2} = \begin{array}{ll}
\texttt{WITH} & t\ \texttt{AS}\ q^{2} \\
\texttt{SELECT} & t_{\texttt{FILTER}}.\rho,\ e_1^{2} \cup \texttt{Y}.y,\ \dots,\ e_n^{2} \cup \texttt{Y}.y \\
\texttt{FROM} & t, \\
 & \texttt{LATERAL}\ read_{\texttt{FILTER}}(\ell,\ \varphi_v,\ t.\rho)\ \texttt{AS}\ t_{\texttt{FILTER}}(\rho), \\
 & (\texttt{SELECT}\ Y(\bigcup(o_1^{2} \cup \dots \cup o_m^{2}))\ \texttt{FROM}\ t)\ \texttt{AS}\ \texttt{Y}(y)
\end{array}
\end{array}
}{
\begin{array}{ll}
\texttt{SELECT} & e_1,\ \dots,\ e_n \\
\texttt{FROM} & q\ \texttt{AS}\ t \\
\texttt{ORDER BY} & o_1,\ \dots,\ o_m \\
\texttt{OFFSET} & k \\
\texttt{LIMIT} & l
\end{array} \Rightarrow \langle i^{1}, i^{2} \rangle
} \quad (\textsc{OrderBy})
$$

Figure 2.7: The rewrite rule for ordered queries, extended from Müller et al. (2018), Figure 21.

`DER BY` and `LIMIT` to retrieve the student(s) with the maximum of ECTS points. According to the OrderBy rewrite rule suggested by Müller et al. (2018, Figure 21), the why-provenance of the result ('Mary') would only consist of $\rho_{1,3}$, the cell containing Mary's ECTS points. However, Mary would not come out as the result if her fellow students did not have less ECTS points than her. We would therefore expect the why-provenance to also contain $\rho_{2,3}$ and $\rho_{3,3}$, the ECTS points of Peter and Ann.

This is actually implemented in their AggWin rule (Müller et al. 2018, Figure 12), where an aggregate function over a given window w is transformed into the big union over w in Phase 2. This rule would apply to the right query in Figure 2.8, which also retrieves the student(s) with the most ECTS points, by first computing

```sql
SELECT s.name
FROM students AS s
ORDER BY s.ects DESC
LIMIT 1;
```

```sql
SELECT s.name
FROM (SELECT s.name, s.ects,
             MAX(s.ects) OVER () AS m
      FROM students AS s) AS s
WHERE s.ects = s.m;
```

Figure 2.8: Two queries retrieving the name of the student with the most ECTS ('Mary'), using `ORDER BY-LIMIT` and window aggregation, respectively.

the maximum ECTS and then filtering the students accordingly. Here, the dependency set of `m` in the inner query is $\{\rho_{1,3}, \rho_{2,3}, \rho_{3,3}\}$ due to AggWin and ends up in the why-provenance of the result due to its appearance in the `WHERE` clause of the outer query. AggWin thus reflects our intuitions about the why-provenance of a maximum computation.

Therefore, I have altered the why-provenance derivation of OrderBy to be the big union over the ordering criteria o_i of all tuples in the relation and not just those of the current row, as shown in Figure 2.7. A consequence of this change is that the computation of why-provenance cannot be outsourced to a lateral table anymore. The naive solution is to use a window aggregate instead. However, this duplicates the potentially complex aggregation and union of why-provenance for each result attribute, which leads to a significant performance overhead, especially for the dependency set implementation used in this thesis. Thus, I have decided to aggregate why-provenance once in table `Y`, and move the computation of q^2 into a common table expression, so that it only has to be evaluated once for both the main query and the subquery that assembles `Y`.

2.3 Implementation

One of the main goals of Müller et al. (2018) is to design a system for provenance derivation that can be expressed directly in SQL and does not require any modification of the RDBMS. While they do provide rewrite rules into SQL for the instrumented query and interpreter, the question of how to represent dependency sets and implement the $write_{\mathsf{X}}$ and $read_{\mathsf{X}}$ logging functions is not prescribed. In theory, an implementation that is more deeply nested into the RDBMS is possible. However, in the appendix, they describe how to realize dependency sets and log-

ging in the PostgreSQL RDBMS using SQL data types and user defined functions. This approach was implemented by Paradzik (2018) and is also the basis for my implementation.

2.3.1 Dependency Sets

Unfortunately, PostgreSQL does not have a native set data type. Hence, we must define a custom data type $\mathbb{P}$ that

(a) ensures no duplicate elements are inserted, and

(b) implements union and big union operators $\cup$ and $\bigcup$.

The suggestion of Müller et al. (2018, p. 18f) is to use integer arrays (`INT[]` type) for this. Since arrays are not duplicate-free, an explicit duplicate removal function `set` is implemented. A union operator `|` is then defined as a_1 `|` a_2 `= set(array_cat(`a_1`,` a_2`))`. For big union, we can make use of the `array_agg` aggregation function to assemble an arbitrary number of dependency sets plus a custom `flatten` function to merge them and remove duplicates.

Duplicate elimination in arrays is costly, since finding a single element in an unsorted array of length n already has time complexity $\mathcal{O}(n)$. In our case, duplicate elimination is implemented as a `SELECT DISTINCT ON` query that requires `unnest`ing the array first and aggregating it again afterwards. Since dependency sets can grow quite large depending on the type of the input query and, especially for complex nested queries, many unions are performed during Phase 2, this can lead to a considerable overhead. This is also the reason why OrderBy's why-provenance should not be repeatedly computed as a window aggregate.

Müller et al. (2018) also experimented with a compressed bit set representation based on *roaring bitmaps*. This significantly improved the running time of Phase 2 compared to the array realization, especially for queries that spawn very large dependency sets. However, roaring bitmaps are not supported natively by PostgreSQL, but must be added as an extension module, which goes against the goal of not interfering with the RDBMS (Müller et al. 2018, p. 19). Thus, I stick with the array representation of dependency sets in this thesis.

2.3.2 Logging Functions

Since logs mainly store site and tuple identifiers and never actual attribute values, their shape only depends on the logged decision. Logging a join via $write_{\texttt{JOIN}\langle 2\rangle}$, for instance, specifically requires logging the tuple ids of the two joined tuples, while logging a filtering decision via $write_{\texttt{FILTER}}$, e.g. in an `ORDER BY` clause, requires only the single tuple identifier of a selected row. Thus, Müller et al. (2018, p. 16f) suggest to store each type of log in a relation that can be written to and read from via a user defined function (UDF).

The log for a simple join between two relations, for instance, contains five attributes:

- `location` and `phi` for query and function call sites ℓ and φ_v,
- `tuid1` and `tuid2` for tuple identifiers $t_1.\rho$ and $t_2.\rho$ of the two join relations t_1 and t_2, and
- `tuidout`, a new automatically generated unique tuple identifier for the joined tuple.

The implementations of the logging functions $write_{\texttt{JOIN}\langle 2\rangle}$ and $read_{\texttt{JOIN}\langle 2\rangle}$ (see Figure 2.9) are then straightforward: The PL/pgSQL function `log.writejoin` inserts the given call sites and tuple ids into the log relation and returns the freshly generated join tuple id. In case the log entry already exists, a `UNIQUE_VIOLATION` is thrown by the log relation and the function instead retrieves the previously assigned join tuple id via a call to `log.readjoin`. This plain SQL function, in turn, simply selects the matching join tuple id from the log relation.

```
CREATE FUNCTION log.writejoin(v_location :T_LOC, v_phi :T_LOC,
    v_tuid1 :T_TUID, v_tuid2 :T_TUID)
RETURNS :T_TUID AS
$$
DECLARE
    res log.join2.tuidout%TYPE;
BEGIN
    INSERT INTO log.join2 (location, phi, tuid1, tuid2)
        VALUES (v_location, v_phi, v_tuid1, v_tuid2)
        RETURNING tuidout INTO res;
        RETURN res;
EXCEPTION
    WHEN UNIQUE_VIOLATION THEN
        RETURN log.readjoin(v_location, v_phi, v_tuid1, v_tuid2);
END;
$$ LANGUAGE PLPGSQL VOLATILE;

CREATE FUNCTION log.readjoin(v_location :T_LOC, v_phi :T_LOC, v_tuid1
    :T_TUID, v_tuid2 :T_TUID)
RETURNS TABLE(tuid :T_TUID) AS
$$
    SELECT j.tuidout
      FROM log.join2 AS j
     WHERE j.location=v_location
       AND j.phi=v_phi
       AND j.tuid1=v_tuid1
       AND j.tuid2=v_tuid2
$$ LANGUAGE SQL STABLE;
```

Figure 2.9: The (Postgre)SQL implementation of the logging functions $write_{\texttt{JOIN}\langle 2\rangle}$ and $read_{\texttt{JOIN}\langle 2\rangle}$. `:T_LOC` and `T_TUID` are PostgreSQL variables referring to the types of call sites and tuple ids, respectively. Both are set to `INT` in the current implementation.

3 Rewrite Rules for PL/pgSQL Functions

PL/pgSQL (short for 'Procedural Language/PostgreSQL Structured Query Language') is a procedural programming language that comes with the PostgreSQL RDBMS. It can be used to create user-defined procedures and functions using regular SQL queries together with additional control structures like conditionals and loops (PostgreSQL Global Development Group 2021).

The procedural structure of PL/pgSQL functions poses new challenges for both logging and provenance derivation. Consider, for instance, the function snippet in Figure 3.1. Rewrite rule Join, as proposed by Müller et al. (2018), will inject a log call into the `IF` condition's `SELECT` clause. However, the arguments ℓ, ρ to our log function are not unique across repeated evaluations of the query inside a loop structure. Thus, for provenance derivation in PL/pgSQL, we need a new way to differentiate between distinct log calls inside the same query.

Also, the cells contributing to the why-provenance of an expression e may not be local to e anymore. In plain SQL, the why-provenance typically stems from the `WHERE` or `ORDER BY` clause of the same query that returns the result. PL/pgSQL control structures like the `IF-THEN-ELSE` in Figure 3.1, however, have wider scope: Just like the `WHERE` clause in a regular query is a filter on its output, the `IF` condition acts as an input filter for its `THEN` and `ELSE` branch. The data it uses (i.e. the selected `cid` and `ects` cells) should therefore contribute to the why-provenance of all statements inside the `THEN` and `ELSE` bodies. We thus need to find a way to prop-

```
c := 1;
LOOP
    IF NOT EXISTS(SELECT cid          => SELECT write_JOIN⟨1⟩(ℓ, ρ), cid
                  FROM classes           FROM classes
                  WHERE ects >= c)       WHERE ects >= c
        RETURN c - 1;
    ELSE
        c := c + 1;
    END IF;
END LOOP;
```

Figure 3.1: Excerpt from a (very inefficient) PL/pgSQL function retrieving the maximum ECTS obtainable in any class. In gray: The Phase $\mathbb{1}$ rewrite of the embedded `SELECT-FROM-WHERE` query.

agate why-provenance to these statements without breaking the compositionality of the rewrite rules.

In this chapter, I develop rewrite rules $i \Rightarrow \langle i^{\mathbb{1}}, i^{2} \rangle$ for PL/pgSQL constructs as an extension to those defined by Müller et al. (2018) for SQL. In section 3.1, I introduce two PL/pgSQL variables, the *function call site* φ_v and *cumulative why-provenance* y_v, that are added to every PL/pgSQL function to resolve the issues discussed above. Section 3.2 then elaborates rewrite rules for the data-modifying `INSERT` and `UPDATE` statements, which are part of the standard SQL dialect but were not covered by Müller et al. (2018). Afterwards, I discuss the new rewrite rules for PL/pgSQL constructs, namely function definitions and calls (section 3.3), control structures (section 3.4), and other minor constructs (section 3.5).

3.1 Function State Variables

For distinguishing between nested subqueries in regular SQL queries, Müller et al. (2018) introduced the (query) call site ℓ, which uniquely identifies each subquery for decision logging. The value of ℓ is assigned by the rewriter: Diving into the nested query, it labels each subquery with a unique ℓ and hard-codes this value into the rewritten queries, ensuring that subqueries have the same ℓ in both $i^{\mathbb{1}}$ and i^{2}. As mentioned in the introduction to this chapter, this is not sufficient anymore to distinguish query calls in PL/pgSQL functions, since the same query may be

called multiple times inside a loop.

To resolve this, I introduce the *function call site* φ_v that was already mentioned in the previous chapters. φ_v is used by the logging functions to distinguish between different calls of the same structure with identifier ℓ. Since we do not know in advance which directions control structures take (e.g. how many iterations a loop will have), the value of φ_v cannot be inserted by the rewriter but has to be computed at runtime. Thus, φ_v is a PL/pgSQL variable that has to be declared in every provenance-aware PL/pgSQL function. As we will see later in section 3.4, the rewriter inserts dynamic updates of φ_v into loops to ensure that each iteration receives a unique function call site. Outside of PL/pgSQL functions, φ_v can be hardcoded to any value, e.g. `-1`.

As noted above, the computation of why-provenance is also not as straightforward anymore as with plain SQL queries: Control structures may contribute to the why-provenance of all statements within their scope. The condition of an `IF` statement, for instance, adds to the why-provenance of every value derived or altered in both its `THEN` and `ELSE` branch. But since the rewriter operates compositionally on a single query layer at a time, this information is not readily available at the level of the child statements. I thus introduce an additional PL/pgSQL variable y_v that holds the *cumulative why-provenance* of parent control structures. This y_v is then appended to the dependency sets of all values derived inside the function.

3.2 Data-Modifying Statements

Before we can start to consider PL/pgSQL expressions, we first need to revisit some basic SQL statements, namely those for modifying a relation: `INSERT` and `UPDATE`. Since PL/pgSQL functions are a convenient way to perform conditional database updates or sequential updates to ensure consistency between several relations, these two statements are frequently used inside PL/pgSQL functions and any provenance system for PL/pgSQL needs to cover them. They cannot be nested inside regular queries though, which is why Müller et al. (2018) did not design any rewrite rules for them.

φ_v, y_v
$\ell = site()$
$q \Rightarrow \langle q^{\mathbb{1}}, q^{2} \rangle$
$r_i \Rightarrow \langle r_i^{\mathbb{1}}, r_i^{2} \rangle \;\; \forall i = 1, \ldots, m$

$i^{\mathbb{1}} =$

```
INSERT INTO t (ρ, c_1, …, c_n)
    (SELECT write_ENV(ℓ, φ_v, t.ρ) AS ρ,
            t.c_1, …, t.c_n
       FROM q^𝟙 AS t)
[ RETURNING r_1^𝟙, …, r_m^𝟙
  [ INTO STRICT v_1, … v_m ]; ]
```

$i^{2} =$

```
INSERT INTO t (ρ, c_1, …, c_n)
    (SELECT read_ENV(ℓ, φ_v, t.ρ) AS ρ,
            t.c_1 ∪ y_v, …, t.c_n ∪ y_v
       FROM q^2 AS t)
[ RETURNING r_1^2, …, r_m^2
  [ INTO STRICT v_1, … v_m ]; ]
```

```
INSERT INTO t (c_1, …, c_n) q
[ RETURNING r_1, …, r_m
  [ INTO STRICT v_1, … v_m ]; ]
```

$\Rightarrow \langle i^{\mathbb{1}}, i^{2} \rangle$ (INSERT)

Figure 3.2: The rewrite rule for insertions.

3.2.1 Insertions

Figure 3.2 shows the rewrite rule INSERT for `INSERT` statements. It basically only ensures that every inserted tuple receives a new unique identifier by calling $write/read_{\mathsf{ENV}}$ on the subquery's tuple identifier $t.\rho$.

We know that $t.\rho$ must be present if q is a `SELECT` query due to JOIN, but there is no such rewrite rule yet for a `VALUES` expression, which is rather commonly used inside `INSERT` statements. Thus, Figure 3.3 introduces the rule VALUES that simply prepends a literal identifier (e.g. integers $1 \ldots m$) to each row in both phases. This 'dummy' ρ can then be used to disambiguate rows in a logging function call as in the INSERT rule above.

$$
\begin{array}{ll}
i^{1} = & \texttt{VALUES } (\rho_1,\ v_{1,1},\ \dots,\ v_{1,n}), \\
 & \quad \dots, \\
 & \quad (\rho_m,\ v_{m,1},\ \dots,\ v_{m,n}) \\
i^{2} = & i^{1}
\end{array}
$$

$$
\frac{}{\texttt{VALUES } (v_{1,1},\ \dots,\ v_{1,n}),\ \dots,\ (v_{m,1},\ \dots,\ v_{m,n}) \Rightarrow \langle i^{1}, i^{2} \rangle} \quad (\textsc{Values})
$$

Figure 3.3: The rewrite rule for VALUES expressions.

3.2.2 Updates

The rewrite rule Update (see Figure 3.4) works very similar to Join on a single relation: The logging function $write_{\texttt{FILTER}}$ records the tuple identifiers of those tuples passing the filter imposed by the WHERE clause. In contrast to $write_{\texttt{JOIN}\langle m\rangle}$, it does not create a new tuple identifier, but simply returns the one it is given as an argument. Thus, SET ρ = $write_{\texttt{FILTER}}(\dots)$ does not alter ρ but simply performs logging.

In Phase 2, the log can then be queried via $read_{\texttt{FILTER}}$ to reenact the filtering. Unlike in regular queries, why-provenance derivation cannot be outsourced to a FROM subquery here, because UPDATE only allows to directly operate on a single source relation. Hence, the same why-provenance derivation must be performed for each target attribute, which might lead to some overhead if the WHERE clause contained a complex query resulting in a large dependency set.

Finally, inside PL/pgSQL functions, expressions r_i on the updated relation may be fed into variables v_i using a RETURNING-INTO clause. These expressions are simply transformed recursively.

3.3 Functions

PL/pgSQL code is organized in user-defined functions. Thus, the outermost layer encountered by any PL/pgSQL rewrite routine is the function definition specifying the number and types of arguments as well as the return type of the function. For provenance derivation, we need to slightly adjust these parameters via the rewrite

$$
\begin{array}{l}
\varphi_v, y_v \\
\ell = site() \\
e_i \Rightarrow \langle e_i^{\mathbb{1}}, e_i^{2} \rangle \;\; \forall i = 1, \dots, n \\
r_i \Rightarrow \langle r_i^{\mathbb{1}}, r_i^{2} \rangle \;\; \forall i = 1, \dots, m \\
p \Rightarrow \langle p^{\mathbb{1}}, p^{2} \rangle \\
\\
i^{\mathbb{1}} = \begin{array}{l}
\texttt{UPDATE } t \\
\texttt{SET } \rho \texttt{ = } write_{\texttt{FILTER}}\texttt{(}\ell\texttt{, } \varphi_v\texttt{, } t\texttt{.}\rho\texttt{), } c_1 \texttt{ = } e_1^{\mathbb{1}}\texttt{, ..., } c_n \texttt{ = } e_n^{\mathbb{1}} \\
\texttt{WHERE } p^{\mathbb{1}} \\
\texttt{[RETURNING } r_1^{\mathbb{1}}\texttt{, ..., } r_m^{\mathbb{1}} \\
\quad \texttt{[INTO STRICT } v_1\texttt{, ... } v_m \texttt{];]}
\end{array} \\
\\
i^{2} = \begin{array}{l}
\texttt{UPDATE } t \\
\texttt{SET } c_1 \texttt{ = } e_1^{2} \cup y_v \cup Y(p^{2})\texttt{, ..., } c_n \texttt{ = } e_n^{2} \cup y_v \cup Y(p^{2}) \\
\texttt{WHERE EXISTS (}read_{\texttt{FILTER}}\texttt{(}\ell\texttt{, } \varphi_v\texttt{, } t\texttt{.}\rho\texttt{))} \\
\texttt{[RETURNING } r_1^{2}\texttt{, ..., } r_m^{2} \\
\quad \texttt{[INTO STRICT } v_1\texttt{, ... } v_m \texttt{];]}
\end{array} \\
\hline
\begin{array}{l}
\texttt{UPDATE } t \\
\texttt{SET } c_1 \texttt{ = } e_1\texttt{, ..., } c_n \texttt{ = } e_n \\
\texttt{WHERE } p \\
\texttt{[RETURNING } r_1\texttt{, ..., } r_m \\
\quad \texttt{[INTO STRICT } v_1\texttt{, ... } v_m \texttt{];]}
\end{array} \Rightarrow \langle i^{\mathbb{1}}, i^{2} \rangle
\end{array}
\quad (\textsc{Update})
$$

Figure 3.4: The rewrite rule for updates.

rule FUNCDEF, shown in Figure 3.5, and adapt function calls accordingly via the rewrite rule FUNCCALL, shown in Figure 3.6.

First of all, we add an additional argument to the list, namely our function call site φ_v. The reason for declaring it as a function argument rather than as a local variable is that we would like to use it for logging (and thus linking) nested function calls, as implemented by FUNCCALL. Here, we use $write/read_{\mathsf{ENV}}$ to generate and log a fresh function call site for the inner function, to be able to distinguish between different calls of the same function.

In Phase 2, we additionally append the cumulative why-provenance y_v to the argument list, since function calls might occur inside control structures of another function, and we want the why-provenance of that control structure to carry over to the statements inside the inner function. This ensures that actual function calls derive the same provenance as if their content was inlined.

$$
\frac{
\begin{array}{l}
s \Rightarrow \langle s^{\mathbb{1}}, s^{\mathbb{2}} \rangle \\
i^{\mathbb{1}} = \begin{array}{l}\texttt{CREATE FUNCTION } f(\varphi_v\ \tau_\rho,\ v_1\ \tau_1,\ \ldots)\texttt{ RETURNS } \tau_r \texttt{ AS \$\$} \\ \quad s^{\mathbb{1}}; \\ \texttt{\$\$ LANGUAGE PLPGSQL}\end{array} \\
i^{\mathbb{2}} = \begin{array}{l}\texttt{CREATE FUNCTION } f(\varphi_v\ \tau_\rho,\ y_v\ \mathbb{P},\ v_1\ \mathbb{P},\ \ldots)\texttt{ RETURNS } \mathbb{P} \texttt{ AS \$\$} \\ \quad s^{\mathbb{2}}; \\ \texttt{\$\$ LANGUAGE PLPGSQL}\end{array}
\end{array}
}{
\begin{array}{l}\texttt{CREATE FUNCTION } f(v_1\ \tau_1,\ \ldots)\texttt{ RETURNS } \tau_r \texttt{ AS \$\$} \\ \quad s; \\ \texttt{\$\$ LANGUAGE PLPGSQL}\end{array} \Rightarrow \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle
}\ (\text{FuncDef})
$$

Figure 3.5: The rewrite rule for function definitions.

$$
\frac{
\begin{array}{l}
\varphi_v, y_v \\
\ell = site() \\
e_i \Rightarrow \langle e_i^{\mathbb{1}}, e_i^{\mathbb{2}} \rangle \quad \forall i = 1, \ldots, n \\
i^{\mathbb{1}} = f(write_{\mathsf{ENV}}(\ell,\ \varphi_v),\ e_1^{\mathbb{1}},\ \ldots,\ e_n^{\mathbb{1}}) \\
i^{\mathbb{2}} = f(read_{\mathsf{ENV}}(\ell,\ \varphi_v),\ y_v,\ e_1^{\mathbb{2}},\ \ldots,\ e_n^{\mathbb{2}})
\end{array}
}{
f(e_1,\ \ldots,\ e_n) \Rightarrow \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle
}\ (\text{FuncCall})
$$

Figure 3.6: The rewrite rule for function calls.

Finally, in Phase $\mathbb{2}$, we also need to convert the regular argument types τ_i as well as the return type τ_r to $\mathbb{P}$, since we only operate on dependency sets there.

3.4 Control Structures

Control structures, i.e. those structures that determine the order in which the statements within their scope are executed, form the core of PL/pgSQL. This section introduces the rewrite rules for four of them, namely statement blocks, conditionals (`IF-THEN-ELSE`), unconditional loops (`LOOP`) and foreach loops (`FOR-IN`).

y_v
$q_i \Rightarrow \langle q_i^{\mathbb{1}}, q_i^{2} \rangle \;\; \forall i = 1, \ldots, n$
$s_i \Rightarrow \langle s_i^{\mathbb{1}}, s_i^{2} \rangle \;\; \forall i = 1, \ldots, m$

$i^{\mathbb{1}} =$

[<<label>>]
[**DECLARE**
    $v_1 \;\; \tau_1$ [:= $q_1^{\mathbb{1}}$];
    …
    $v_n \;\; \tau_n$ [:= $q_n^{\mathbb{1}}$];]
BEGIN
    $s_1^{\mathbb{1}}$;
    …
    $s_m^{\mathbb{1}}$;
END;

$i^{2} =$

[<<label>>]
[**DECLARE**
    $v_1 \;\; \mathbb{P}$ [:= $q_1^{2} \cup y_v$];
    …
    $v_n \;\; \mathbb{P}$ [:= $q_n^{2} \cup y_v$];]
BEGIN
    s_1^{2};
    …
    s_m^{2};
END;

(Block)

[<<label>>]
[**DECLARE**
    $v_1 \;\; \tau_1$ [:= q_1];
    …
    $v_n \;\; \tau_n$ [:= q_n];] $\Rightarrow \langle i^{\mathbb{1}}, i^{2} \rangle$
BEGIN
    s_1;
    …
    s_m;
END;

Figure 3.7: The rewrite rule for statement blocks.

3.4.1 Blocks

A statement block surrounds one or more statements s_i that should be executed sequentially, i.e. one after the other. They are enclosed by `BEGIN` and `END` tags, optionally preceded by a label `<<lbl>>` and/or a `DECLARE` block declaring new (or overwriting old) variables v_i of type τ_i that should be accessible within the statement block. These variables may be initialized to a value q_i or left uninitialized. The rewrite rule Block (see Figure 3.7) is simple: The queries q_i and statements s_i are recursively transformed and in Phase 2, the type of all declared variables is set to $\mathbb{P}$.

3.4.2 Conditionals

PL/pgSQL also comes with a conditional `IF`-statement that executes statement s_1 or s_2 depending on condition c, where c is an SQL query resulting in a single boolean value. Like Update, rule IfThenElse (see Figure 3.8) uses the logging function $write_{\texttt{FILTER}}$ to record its decision: A log entry is created if c evaluated to `TRUE`, but not if it evaluated to `FALSE`. In Phase 2, the statement then does not need to evaluate c again, but simply has to ask whether there is a log entry or not.

Whether we end up executing s_1 or s_2 depends on c. Hence, c constitutes to the why-provenance of both statements. To ensure this, the Phase 2 is surrounded by a statement block which overwrites y_v with its union with the why-provenance embodied by c. The freshly declared y_v shadows the previous y_v, but only within the statement block. After the `IF` statement and its children have been executed, the symbol y_v again refers to the previous state of y_v.

3.4.3 Unconditional Loop

An unconditional loop repeats the statements s_i in its body until either the enclosing function returns or an `EXIT` statement is executed. No provenance may be derived from this control structure, but we need to ensure that each iteration of the loop uses a different φ_v. Thus, the rewrite rule Loop (see Figure 3.9) inserts an assignment of a new value to φ_v via the logging function $write/read_{\texttt{ENV}}$ at the beginning of the `LOOP` block.

φ_v, y_v
$\ell = site()$
$c \Rightarrow \langle c^{\mathbb{1}}, c^{\mathbb{2}} \rangle$
$s_1 \Rightarrow \langle s_1^{\mathbb{1}}, s_1^{\mathbb{2}} \rangle$
$s_2 \Rightarrow \langle s_2^{\mathbb{1}}, s_2^{\mathbb{2}} \rangle$

$i^{\mathbb{1}} =$

```
IF c¹
THEN
    BEGIN
        PERFORM write_FILTER(ℓ, φ_v);
        s₁¹;
    END
[ ELSE s₂¹ ]
END IF;
```

$i^{\mathbb{2}} =$

```
DECLARE
    y_v := y_v ∪ Y(c²);
BEGIN
    IF EXISTS (read_FILTER(ℓ, φ_v))
    THEN
        s₁²
    [ ELSE s₂² ]
    END IF;
END;
```

(IfThenElse)

```
IF c
THEN s₁
[ ELSE s₂ ]
END IF;
```

$\Rightarrow \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle$

Figure 3.8: The rewrite rule for conditions.

3.4.4 Foreach Loop

A foreach loop performs statements s_i for each tuple $(v_1, \dots, v_n)$ resulting from a query q. The rewrite rule Foreach in Figure 3.10 for this control structure involves the most complex transformations of the rewrite rules discussed in this thesis, because it must

(a) ensure that each loop iteration is performed on a distinct φ_v,

(b) guarantee that the result tuples of q are processed in the same order in both Phase $\mathbb{1}$ and Phase $\mathbb{2}$, and

φ_v
$\ell = site()$
$s_i \Rightarrow \langle s_i^{\mathbb{1}}, s_i^{2} \rangle \;\; \forall i = 1, \dots, n$

$i^{\mathbb{1}} =$

```
[ <<label>> ]
LOOP
    φ_v := write_ENV(ℓ, φ_v);
    s_1^𝟙;
    …
    s_n^𝟙;
END LOOP;
```

$i^{2} =$

```
[ <<label>> ]
LOOP
    φ_v := read_ENV(ℓ, φ_v);
    s_1^2;
    …
    s_n^2;
END LOOP;
```

(Loop)

```
[ <<label>> ]
LOOP
    s_1;                ⇒ ⟨i^𝟙, i^2⟩
    …
    s_n;
END LOOP;
```

Figure 3.9: The rewrite rule for unconditioned loops.

(c) derive why-provenance from q for all statements s_i in the loop body.

To implement (a), the Loop transformation is applied to the loop body, updating φ_v via calls to $write/read_{\mathsf{ENV}}$. The tuple order (b) is explicitly recorded by another logging function $write_{\mathsf{ORD}}$ which links the tuple identifiers of the query result to their row number. In Phase 2, the tuples are then ordered by that row number which is retrieved from the log via $read_{\mathsf{ORD}}$.

For (c), we need to extract the local why-provenance of q, because the conditions that shape the result of q also shape our loop, namely the number of iterations and the data it is performed on. As imposed by Join, this local why-provenance is computed anyway into a lateral table `Y` for any normal `SELECT` query. Thus, Foreach edits q to retrieve its `Y` table (this is notated as $Y(t)$ with q^2 `AS` t). The

φ_v, y_v
$\ell_1 = site()$
$\ell_2 = site()$
$q \Rightarrow \langle q^{\mathbb{1}}, q^{2} \rangle$
$s_i \Rightarrow \langle s_i^{\mathbb{1}}, s_i^{2} \rangle \ \ \forall i = 1, \dots, n$

$i^{\mathbb{1}} =$

```
FOR _, v1, …, vn IN (
    SELECT write_ORD(ℓ1, φv, t.ρ, ROW_NUMBER() OVER ()),
           t.c1, …, t.cn
    FROM   q^𝟙 AS t)
LOOP
    φv := write_ENV(ℓ2, φv);
    s1^𝟙;
    …
    sn^𝟙;
END LOOP;
```

$i^{2} =$

```
FOR _, yl, v1, …, vn IN (
    SELECT   t_ORD.ρ, Y.y, t.c1, …, t.cn
    FROM     q^2 AS t,
             LATERAL read_ORD(ℓ1, φv, t.ρ) AS t_ORD(ρ, o),
             LATERAL Y(t) AS Y(y)
    ORDER BY t_ORD.o)
LOOP
    DECLARE
        yv := yv ∪ yl;
    BEGIN
        φv := read_ENV(ℓ2, φv);
        s1^2;
        …
        sn^2;
    END;
END LOOP;
```

(Foreach)

```
FOR v1, …, vn IN q
LOOP
    s1;
    …
    sn;
END LOOP;
```

$\Rightarrow \langle i^{\mathbb{1}}, i^{2} \rangle$

Figure 3.10: The rewrite rule for foreach loops.

dependency set contained in `Y` is then returned into an additional loop variable y_l. To enter y_l into the cumulative why-provenance, a statement block is wrapped around the loop body and y_v is shadowed by the union of itself and y_l, just as in IfThenElse.

3.5 Minor Rewrite Rules

Many smaller (PL/pg)SQL structures like literals, binary operators or variable assignments do not directly contribute to provenance derivation, and they do not change their form at all in Phase $\mathbb{1}$, except for recursive transformation of their child structures. However, they do require modifications in Phase $\mathbb{2}$, for one of two reasons:

(1) They assume that they operate on values rather than dependency sets and/or return values rather than dependency sets.

(2) They persist a result, so the cumulative why-provenance must be appended to that result.

3.5.1 Dependency Set Conversion

The first type subsumes literal values or constructors and operators. Figure 3.11 shows the rewrite rules for some frequent constructs of this type. Literals are converted to empty dependency sets, because they do not have any provenance. An array is transformed to the union of its items, since together they constitute its provenance. Binary operators (such as arithmetic or boolean operators) are replaced by the union operator, as Phase $\mathbb{2}$ operates on dependency sets, not numeric/boolean values, and because the provenance of e.g. a sum is assumed to consist of the provenance of its addends. Finally, boolean sublinks like `IN` or `EXISTS` over a single-column query result are reduced to the big union over all dependency sets inside that query result.

3.5.2 Injecting Cumulative Why-Provenance

Inside PL/pgSQL functions, we would like to ensure that the cumulative why-provenance is added to any dependency set that is persisted in a relation or vari-

$$\frac{}{l \;\Rightarrow\; \langle l, \emptyset \rangle} \quad (\text{LIT})$$

$$\frac{e_i \Rightarrow \langle e_i^{\mathbb{1}}, e_i^{2} \rangle \;\; \forall i = 1, \dots, n}{\texttt{ARRAY[}e_1\texttt{, ..., }e_n\texttt{]} \;\Rightarrow\; \langle \texttt{ARRAY[}e_1^{\mathbb{1}}\texttt{, ..., }e_n^{\mathbb{1}}\texttt{]}, \bigcup e_i^{2} \rangle} \quad (\text{ARRAY})$$

$$\frac{\begin{array}{l}\oplus \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{AND}, \texttt{OR}, \dots\} \\ e_1 \Rightarrow \langle e_1^{\mathbb{1}}, e_1^{2} \rangle, e_2 \Rightarrow \langle e_2^{\mathbb{1}}, e_2^{2} \rangle\end{array}}{e_1 \oplus e_2 \;\Rightarrow\; \langle e_1^{\mathbb{1}} \oplus e_2^{\mathbb{1}}, e_1^{2} \cup e_2^{2} \rangle} \quad (\text{BINOP})$$

$$\frac{\begin{array}{l} e \Rightarrow \langle e^{\mathbb{1}}, e^{2} \rangle \\ q \Rightarrow \langle q^{\mathbb{1}}, q^{2} \rangle \\ i^{\mathbb{1}} = \begin{array}{l}\texttt{[} e^{\mathbb{1}} \texttt{ [NOT] IN | [NOT] EXISTS | [NOT] ANY | [NOT] ALL]} \\ \quad \texttt{(SELECT } t.c \texttt{ FROM } q^{\mathbb{1}} \texttt{ AS } t\texttt{)}\end{array} \\ i^{2} = \begin{array}{l}\texttt{SELECT [} e^{2} \cup \texttt{] } \bigcup \{t\texttt{.}c\} \\ \texttt{FROM } q^{2} \texttt{ AS } t\end{array}\end{array}}{\begin{array}{l}\texttt{[} e \texttt{ [NOT] IN | [NOT] EXISTS} \\ \quad \texttt{| [NOT] ANY | [NOT] ALL]} \;\Rightarrow\; \langle i^{\mathbb{1}}, i^{2} \rangle \\ \quad \texttt{(SELECT } t.c \texttt{ FROM } q \texttt{ AS } t\texttt{)}\end{array}} \quad (\text{SUBLINK})$$

Figure 3.11: Rewrite rules for literals, Array declarations, binary operators, and sublinks. LIT, BINOP (orig. BUILTIN), SUBLINK (orig. NESTEDSUBQUERY) were extended from Müller et al. (2018, Figure 12).

able, i.e. when that dependency set might survive outside of the current y_v scope. It is already appended during insertions and updates via rules INSERT and UPDATE (see section 3.2), which takes care of the first scenario. In addition, the three rules in Figure 3.12 add y_v to direct variable assignments, `SELECT-INTO` queries that write query results into variables, and to return values.

$$\frac{y_v, e \Rightarrow \langle e^{\mathbb{1}}, e^{2} \rangle}{v \texttt{ := } e\texttt{;} \Rightarrow \langle v \texttt{ := } e^{\mathbb{1}},\ v \texttt{ := } e^{2} \cup y_v\texttt{;} \rangle} \quad \text{(VarAssign)}$$

$$y_v, q \Rightarrow \langle q^{\mathbb{1}}, q^{2} \rangle$$

$$i^{\mathbb{1}} = \begin{array}{ll} \texttt{SELECT} & c_1\texttt{, ..., } c_n \\ \texttt{FROM} & q^{\mathbb{1}} \\ \texttt{INTO} & v_1\texttt{, ..., } v_n\texttt{;} \end{array}$$

$$i^{2} = \begin{array}{ll} \texttt{SELECT} & c_1 \cup y_v\texttt{, ..., } c_n \cup y_v \\ \texttt{FROM} & q^{2} \\ \texttt{INTO} & v_1\texttt{, ..., } v_n\texttt{;} \end{array}$$

$$\overline{\begin{array}{ll} \texttt{SELECT} & c_1\texttt{, ..., } c_n \\ \texttt{FROM} & q \\ \texttt{INTO} & v_1\texttt{, ..., } v_n\texttt{;} \end{array} \Rightarrow \langle i^{\mathbb{1}}, i^{2} \rangle} \quad \text{(SelectInto)}$$

$$\frac{y_v, e \Rightarrow \langle e^{\mathbb{1}}, e^{2} \rangle}{\texttt{RETURN [NEXT] } e\texttt{;} \Rightarrow \langle \texttt{RETURN [NEXT] } e^{\mathbb{1}},\ \texttt{RETURN [NEXT] } e^{2} \cup y_v\texttt{;} \rangle} \quad \text{(Return)}$$

Figure 3.12: The rewrite rules for variable assignments, selection into variables, and returns.

4 Implementation

In this chapter, I present `PLSQLProv`, a command-line program that applies the rewrite rules introduced in the previous chapters to a given SQL query q to derive Phase $\mathbb{1}$ and $\mathbb{2}$ representations for q and all PL/pgSQL functions called by it. The source code can be found in the private repository `PLpgSQLProv` on the Git server of the Database Systems Research Group of the University of Tübingen.

`PLSQLProv` has been implemented in Haskell, relying heavily on the `LogParser` package developed by Hirn (2017) for parsing SQL queries and functions into abstract syntax trees (ASTs). Haskell is a purely functional programming language whose static algebraic data types lend themselves naturally to AST parsing and traversal. As we will see later, the Haskell functions for applying the rewrite rules closely resemble the rules' mathematical notation, rendering their implementation rather straightforward.

In section 4.1, I demonstrate how to use the program and explain its settings and options. Afterwards, section 4.2 gives an overview over the modular structure of the implementation, introducing the general flow of the program. I then dive deeper into two parts of the implementation: First, section 4.3 introduces the external `LogParser` package in detail and shows how Haskell's record syntax can be used to elegantly build and process an AST. In section 4.4 I then show my implementation of the provenance rewrite rules.

```
----------------------- Phase 0: Original -----------------------
SELECT "RTE0"."sid" AS "sid", "RTE0"."name" AS "name"
FROM students AS "RTE0"("sid", "name", "ects", "tuid")
WHERE "RTE0"."ects" > (50)

------------------- Phase 1: Instrumentation --------------------
SELECT log.writefilter(1, -1, "RTE0"."tuid") AS "tuid",
       "RTE0"."sid" AS "sid",
       "RTE0"."name" AS "name"
FROM students AS "RTE0"("sid", "name", "ects", "tuid")
WHERE "RTE0"."amount" > (50)

---------------- Phase 2: Provenance Derivation -----------------
SELECT "LFilter"."tuid" AS "tuid",
       "RTE0"."sid" AS "sid",
       "RTE0"."name" AS "name"
FROM students_2 AS "RTE0"("sid", "name", "ects", "tuid"),
     LATERAL log.readfilter(1, -1, "RTE0"."tuid") AS "LFilter"("tuid")
```

Figure 4.1: Example output of `PLSQLProv` for a simple `SELECT-FROM-WHERE` query.

4.1 Usage

The program can be executed most conveniently via the Haskell Tool Stack (Stack contributors 2020) running the following command:

```
stack exec PLSQLProv-exe -- -i <IN_FILE> <options>
```

The obligatory argument `IN_FILE` should be the path to a file containing the SQL statement to be transformed. Assume we had a file `query.sql` containing the following query:

```
SELECT sid, name
FROM students
WHERE ects > 50;
```

Running the command on `-i query.sql` would then print the output in Figure 4.1 to the console, namely the parsed original query and the Phase 𝟙 and 𝟚 where-provenance rewrites. They are formatted by the SQL Pretty-Printer that comes with the `LogParser` with color coding for syntax highlighting.

4.1.1 Options

When redirecting this color-coded output into a file, e.g. using `>`, the markup symbols become visible in the text file, rendering it invalid SQL. Therefore, I introduced the option `-o <OUT_FILE>` to write the plain SQL output into a text file and not to the console.

By default, the program prints the original query as well as the Phase 𝟙 and 𝟚 rewrites. This can be restricted using the `-r <RANGE>` option, where `<RANGE>` can be either of the form `n` (to print phase n), `m-n` (to print phases $m \dots n$, inclusive) or `m,n` (to print phases m and n). The integer `0` is used to refer to the original query.

For debugging purposes, it is possible to print the abstract syntax tree (AST, will be discussed in section 4.3) of the queries instead of their pretty-printed forms using the option `-f ast` (the default being `-f pretty`). Since the AST can get quite large, it is recommended to combine this option with `-r` to print only a single query and `-o` to redirect it into a file.

Why-Provenance rewriting can be switched on via the flag `-y`.

Finally, the three options `-h <HOST>`, `-p <PORT>`, and `-d <DB>` can be used to configure the PostgreSQL database to be used in provenance derivation. The SQL parser used by `PLSQLProv`, as we will see later, interacts with the database during parsing and will throw errors if it cannot, for instance, find the referenced relations. Thus, it is important to provide access to the correct database.

An overview over all program arguments and options is also available via the option `--help`.

4.2 Program Overview

The features and functionalities of `PLSQLProv` are distributed across several Haskell modules. Figure 4.2 shows the dependencies between these modules and thus illustrates the flow of the program.

The user only interacts with the `Main` module, which functions as the entry-point of the program and is only concerned with parsing the command-line arguments

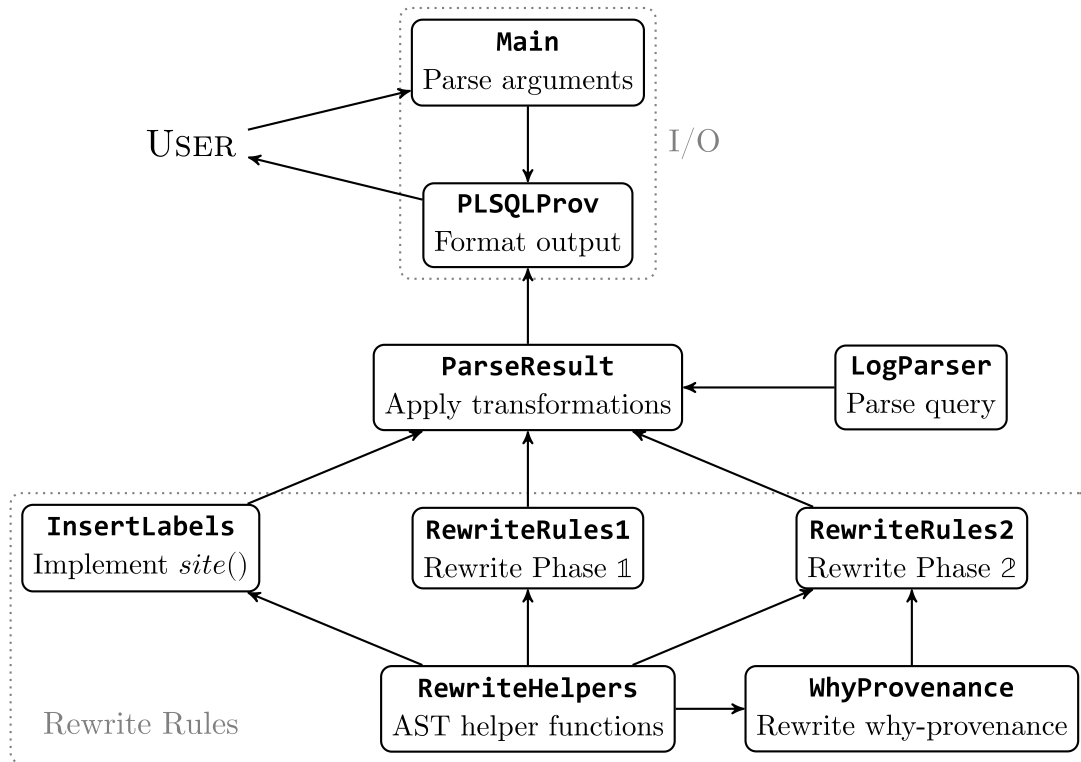


Figure 4.2: The module structure of `PLSQLProv`.

submitted by the user. These are then handed over to the `PLSQLProv` module which retrieves the requested queries and either writes them to the console or a file in the specified format.

The `ParseResult` module is the main interface between the external `LogParser` package, the rewrite rule implementations, and the output formatting in `PLSQLProv`. It manages the `ParseResult` data type which is the output of the `LogParser` and consists of the parsed input SQL query as well as the SQL and PL/pgSQL UDFs referenced by it. The module provides functions for applying rewrite rules to all of these individual ASTs while maintaining a shared incremental query call site and for printing a `ParseResult` result both as a sequence of SQL statements and as a sequence of ASTs.

The actual rewrite rules for individual ASTs are distributed over several modules which will be discussed in more detail in section 4.4. `InsertLabels` basically implements *site*(), i.e. ensures that unique query call sites are available to every log function call. It does so by wrapping numeric labels around query nodes

in the AST. The rewrite rules for Phase 𝟙 and 𝟚 are implemented in modules `RewriteRules1` and `RewriteRules2`, respectively. Both provide functions `transform[1|2]` and `transform[1|2]PLpg` which take an SQL query or PL/pgSQL UDF AST and return its Phase 𝟙/𝟚 version. Finally, the `WhyProvenance` module is used by `RewriteRules2` to additionally apply the transformations for why-provenance if requested by the user. All of these modules use the AST constructors, accessors and transformation functions provided by `RewriteHelpers`.

4.3 Parsing SQL in Haskell

The `LogParser` package was developed by Hirn (2017, p. 4f) for obtaining a JavaScript Object Notation (JSON) representation of an SQL query. It does not process the raw query itself, but retrieves the parse tree generated by the PostgreSQL backend from a log file (hence the name) and builds a type-annotated AST structure from it. The module was later extended to also support PL/pgSQL constructs.

4.3.1 Haskell's Record Syntax

Haskell makes extensive use of algebraic data types, i.e. types that are the sum or product of other data types. The data type `SQLQuery` representing a very simple SQL query, for instance, could be declared like this:

```
data SQLQuery = SFWQuery [ColRef] [String] BoolExpr
data ColRef = ColRef String String
data BoolExpr = And ColRef ColRef | Or ColRef ColRef
```

The above code snippet creates three data types: `SQLQuery`, `ColRef`, and `BoolExpr`. `SFWQuery` is a *constructor* for `SQLQuery` with three *fields*: One of type `[ColRef]` (a list of `ColRef`s) representing the `SELECT` clause, one of type `[String]` representing the `FROM` clause, and one of type `BoolExpr` representing the `WHERE` clause. Similarly, both `And` and `Or` are constructors for `BoolExpr` having two `ColRef` fields each. `ColRef` is both a type and that type's constructor taking two `String`s for the table and column name.

To access the fields of these types, we need to use pattern matching:

```
getWhereClause :: SQLQuery -> BoolExpr
getWhereClause (SFWQuery select from whereEx) = whereEx
```

There is an alternative way of declaring these types called *record syntax*. Record syntax is syntactic sugar for declaring both an algebraic data type and functions like `getWhereClause` accessing the fields of its constructors. Using record syntax, we can declare `SQLQuery` as:

```
data SQLQuery = SFWQuery{ select   :: [ColRef]
                        , from     :: [String]
                        , whereEx  :: BoolExpr }
```

This declaration also creates the functions `select`, `from`, and `whereEx` that take an `SQLQuery` (constructed via `SFWQuery`) and return the respective fields. In addition, we can match individual fields via pattern matching and return a copy of our query with some fields modified. In the below code snippet, the function `copyWhereClause` matches the `WHERE` clause of query `q1` as `w1` and returns a copy of `q2` with its `WHERE` clause replaced by `w1`:

```
copyWhereClause :: SQLQuery -> SQLQuery -> SQLQuery
copyWhereClause q1@SFWQuery{ whereEx = w1 }
                q2@SFWQuery{} = q2{ whereEx = w1 }
```

Alternatively, we can load the `RecordWildcards` extension and write:

```
copyWhereClause q1@SFWQuery{..}
                q2@SFWQuery{} = q2{ whereEx = whereEx }
```

The two dots `..` bind the expressions `select`, `from`, and `whereEx` directly to the corresponding fields of `q1`, shadowing the accessor functions.

It is already evident from the above examples that record types are a straightforward way to build as well as traverse and manipulate an AST. The next section will move away from toy examples and introduce some of the record types produced by the `LogParser` to represent SQL and PL/pgSQL ASTs.

4.3.2 Structure of the ASTs

The `LogParser` is divided into several modules itself, the ones relevant for this thesis being the `AST` and `PLpgAST` modules containing the constructors for plain SQL queries (and UDFs) and PL/pgSQL UDFs, respectively. The `LogParser`'s main module has a function `parseQueryAndUDFs` that takes the string representation of an SQL query an returns a tuple `(AST.Query, [AST.UserFunction], [PLpgAST.PLpgUDF])` (target query, referenced plain SQL UDFs, referenced PL/pgSQL UDFs) which my `ParseResult` module refers to as the `ParseResult` type. The below subsections briefly introduce the most important `AST` and `PLpgAST` constructors needed to understand the rewriting functions introduced in the following section.[1]

4.3.2.1 Plain SQL Queries

All plain SQL queries or expressions are of type `SQL SQLType`, where `SQLType` encodes whether the SQL structure in question is e.g. a complete query, a subexpression, or declares a relation. There are type synonyms for each `SQL SQLType`, the three most prominent ones being:

```
type Query   = SQL SQLQuery   -- complete query
type Expr    = SQL SQLExpr    -- expression
type RangeEx = SQL SQLRangeEx -- relation
```

The type `Query` refers to complete SQL queries, i.e. `SELECT`, `INSERT`, or `UPDATE` statements. Its constructor is `QBlockGeneric` with 19 fields overall to cover features like `WINDOW`, `GROUP BY`, or `LIMIT`, though the most important ones are probably those encoding the `SELECT`, `FROM`, and `WHERE` clause as well as the query type (`SELECT`, `INSERT`, or `UPDATE`):

```
QBlockGeneric
    { select  :: [Expr]
    , from    :: [Expr]
    , whereEx :: Maybe Expr
    , ...
```

[1] All code examples in this section have been simplified. The actual constructors use several language extensions like `GADTs` that are not touched upon by this thesis for the sake of brevity and because they are not directly relevant to my implementation.

```
    , cmdType :: CmdType }
```

Even though `select` is of type `[Expr]`, it exclusively consists of the `Expr` constructor `ETargetEx` which wraps around another `Expr`, adding information such as the alias and `Type` of the expression's result. `Expr` in general refers to any SQL subexpression that does not form a complete query, from constant literals and column references over function applications to sublink expressions like `EXISTS` or `IN`.

Similarly, `from` mostly contains the `Expr` constructor `ERangeTblRef` which holds a `RangeEx` in its field `table`. `RangeEx` has several constructors encoding e.g. table references, table-returning function calls, nested subqueries, or `VALUES` clauses.

The `WHERE` expression is optional (as encoded by the Haskell construct `Maybe`) and can hold any `Expr` constructor returning a boolean value, like `EBoolExpr` (for `AND` and `OR`) or `ESublink`.

4.3.2.2 PL/pgSQL Functions

The `PLpgAST` types are a bit more complex. Every expression in the AST is of type `PLpgSQL PLpgType b c`. The type variables `b` and `c` are usually set to `AST.Type` and `AST.Query`. The three most important type synonyms here are:

```
type PLpgUDF  = PLpgSQL 'PLpgUDFTy  AST.Type AST.Query
type PLpgStmt = PLpgSQL 'PLpgStmtTy AST.Type AST.Query
type PLpgVar  = PLpgSQL 'PLpgVarTy  AST.Type AST.Query
```

`PLpgUDF` encodes a complete PL/pgSQL function. Its constructor is `PLpgFunction`:

```
PLpgFunction
    { oid       :: Integer
    , pludfname :: String
    , dataArea  :: [PLpgVar]
    , block     :: [PLpgStmt]
    , rettype   :: AST.Type
    , plargs    :: [String]
    , plargnos  :: [Integer] }
```

Here, `pludfname` is the function's name, `plargs` and `plargnos` encode its arguments,

`rettype` is its return type, and `block` contains the function body. Finally, `dataArea` is a list of all variables that are used somewhere in this function, even if they are declared in a block nested more deeply.

The `PLpgVar`s in `dataArea` contain information about the variable's name and type as well as an optional initialization query. Most importantly, they link this information to a unique variable id. In the deeper nodes of the AST, all variables will only be referenced by this id and never by their name. The `dataArea` in the enclosing `PLpgFunction` node is the only place where all of the variable information can be accessed. As we will see in the next section, this has fundamental consequences for the implementation of rewrite rules.

In the function's `block`, the actual AST continues in the form of nested `PLpgStmt`s. The constructors for `PLpgStmt` encode the various PL/pgSQL constructs such as `IF`, `LOOP`, or `FORS` (`FOR-IN`). They may have fields of type `AST.Query` (e.g. in the condition of an `IF` node), but no node from the `AST` module can contain a node from the `PLpgAST` module.

4.4 Implementing Rewrite Rules

4.4.1 The OperSem Monad

The rewrite rules are formulated as implementations of the `OperSem` monad included in the `LogParser` package. The `OperSem` monad represents inference rules in the context of operational semantics and consists of several other monads: The `Except` monad for handling exceptions, the `Reader` monad for carrying an *environment* (e.g. parameters for the rule), the `Writer` monad for creating logs of the rule application, and the `State` monad for supplying a modifiable *state*. This is the definition of the `OperSem` monad:

```
type OperSem s e a l =
    ExceptT String (WriterT l (StateT s (Reader e))) a
```

Wrapping the rewrite rules inside `OperSem` has the advantage that we can use the `transformM` function also supplied by the `LogParser` for recursively applying a rule to all eligible fields of the current AST node. More precisely, `AST.transformM` (and,

```
--             in ->         state,  env, out, Log
type Rule a = a  -> OperSem Integer ()   a    ()

insertLabels :: Rule (SQL a)
insertLabels q@QBlockGeneric{} = wrapInLabel q
insertLabels q@EFuncCall{} = wrapInLabel q
insertLabels q = transformM insertLabels q

wrapInLabel :: Rule (SQL a)
wrapInLabel q =
  do
    l <- get
    put $ l + 1
    q' <- transformM insertLabels q
    return GLabel{ labelG = Label l, labelGArg = q' }
```

Figure 4.3: Slightly simplified excerpt from the `InsertLabels` module showing how SQL AST nodes are annotated with query call sites.

analogously, `PLpgAST.transformM`) is defined as:

```
transformM :: Monad m => (forall a. (SQL a -> m (SQL a)))
    -> SQL a -> m (SQL a)
```

It takes a function `f :: (SQL a -> m (SQL a))` that wraps an AST node into some monad and an actual node `n :: (SQL a)`, and applies `f` to all record fields of `n` which are also of type `SQL a`, finally returning `n` as a monad. Since `OperSem` is a monad, this allows us to wrap an AST into a single `OperSem` applying rewrite rules to all nodes in the tree.

4.4.2 Injecting Query Call Sites

Before the actual rewrite rules can be applied, we need to annotate all nodes that are the target of a rewrite rule with a unique query call site ℓ. The module `InsertLabels` implements this by wrapping them into `GLabel` nodes (or `PLLabel` in case of `PLpgAST`). These nodes have only two fields: An integer label and another AST node. We can use this to label all `QBlockGeneric` and `EFuncCall` nodes with a unique ℓ retrieved from the `OperSem` state, as shown in Figure 4.3.

This code snippet illustrates several techniques that are also employed for imple-

menting the rewrite rules. First, the type `Rule a` is declared as a function wrapping an input of type `a` into an `OperSem` with an integer state. This state represents the next available ℓ. The function `insertLabels` then implements such a `Rule` for our plain SQL AST. If it is applied to a `QBlockGeneric` or `EFuncCall` node, it wraps them into a label node via `wrapInLabel`, else it just applies itself to its child nodes using `transformM`.

The rule `wrapInLabel` first retrieves ℓ from the state using the function `get`. It then increments the state via `put`, so that the next application of `wrapInLabel` to a lower node can retrieve a fresh ℓ from the state. Afterwards, `transformM` is used to apply `insertLabels` to the child nodes, and finally, a `GLabel` node annotating the updated node with ℓ is returned.

4.4.3 Where-Provenance

The actual rewrite rules for where-provenance are defined in the modules `Rewrite-Rules1` and `RewriteRules2`. They again define a `Rule` type based on `OperSem`:

```
data StateR = StateR{
  phiV :: Integer,         -- Id of phi var in PLpg AST
  whyParams :: WhyParams  -- Phase 2: Parameters for why-provenance
}
type EnvR = Bool  -- Whether to derive why-provenance
type LogR = ()

type Rule a = a -> OperSem StateR EnvR a LogR
```

Here, the state is a record with two fields. First, `phiV` stores the current id of the function call site variable φ_v in the `dataArea` of the PL/pgSQL AST. Remember that we cannot reference variables by name in the lower nodes, but must refer to them via their id. Since in our rewrite rules, we use φ_v in log calls and when overwriting its value, we carry its id in the `OperSem` state and must update it when entering a new function definition. Second, the state contains some parameters for why-provenance which will become important in section 4.4.4. In addition, the environment is a boolean that encodes whether or not to apply the why-provenance rewrite rules at all.

```
rewrite1PLpg PLLabel{ plLabel = l,
                      plLabelArg = p@IF{ exprif = Just q } } =
  do
    phi <- gets phiV
    p' <- PLTF.transformM rewrite1PLpg p
    q' <- rewrite1 q
    return p'{ exprif = Just q',
               exprthen = (performLogCall writeFilterFunc l phi [])
                          : (exprthen p') }
```

Figure 4.4: Phase $\mathbb{1}$ implementation of IfThenElse.

The functions `rewrite[1|2]` and `rewrite[1|2]PLpg` are then defined as `Rule`s implementing the rewrite rules for plain SQL and PL/pgSQL ASTs, respectively:

```
-- Module RewriteRules1:
rewrite1     :: Rule (SQL a)
rewrite1PLpg :: Rule (PLpgSQL a Type Query)
-- Module RewriteRules2:
rewrite2     :: Rule (SQL a)
rewrite2PLpg :: Rule (PLpgSQL a Type Query)
```

As an example of what such an implementation looks like, we consider the rewrite rule IfThenElse. Figure 4.4 shows its Phase $\mathbb{1}$ implementation. Since the AST has been pre-processed by `InsertLabels`, the function does not only match the `IF` node, but also the surrounding `PLLabel` to extract ℓ from it. In addition, it matches the plain SQL query `q` contained in `IF`'s `exprif` field.

In the function body, we extract φ_v's id from the state. Then, we apply the PL/pgSQL rewrite rules to the child nodes via `transformM`. In addition, we need to apply `rewrite1` to `q` since it is not a PL/pgSQL AST node, and thus is not matched by the `transformM` call. Then, we inject it back into our rewritten `IF` node. In addition, a call to the $write_{\texttt{FILTER}}$ function is prepended to the `THEN` body contained in the field `exprthen`. This call is constructed by the helper function `performLogCall` from the module `RewriteHelpers`. It takes the name of a function (`writeFilterFunc`, a constant also from `RewriteHelpers`), an ℓ, the φ_v id, and a list of relation names whose ρs should be added to the log function's arguments. In this case, we want to call $write_{\texttt{FILTER}}$ on ℓ and φ_v only, so the list remains

```
rewrite2PLpg PLLabel{ plLabel = l,
                      plLabelArg = p@IF{ exprif = Just q } } =
  do
    phi <- gets phiV
    q' <- rewrite2 q
    p' <- PLTF.transformM rewrite2PLpg p
    let p'' = p'{
      exprif = Just $ (emptyQueryBlock CMD_SELECT){
        select = [ eTargetEx
                     (sublinkOf Exists $
                                selectLogCall readFilterFunc l phi [])
                     boolType
                     "found" ] } }
    return p''
```

Figure 4.5: Phase 2 implementation of IfThenElse, without the code pertaining to why-provenance derivation.

empty.

Phase 2 is implemented in a similar fashion, as shown in Figure 4.5. Here, the `IF` condition needs to be replaced by the sublink `EXISTS` $(read_{\texttt{FILTER}}(\ell, \varphi_v))$. For this, we place the log function call in an `ESublink` node. `ESublink` is an `Expr`, but `exprif` must contain a `Query` (unlike its name suggests), so we additionally insert the sublink in the `SELECT` clause of an empty query block. Since the `select` field of such a `QBlockGeneric` must only contain `ETargetEx` nodes, we additionally wrap the sublink into one, specifying its type as boolean and label the only column `"found"`. For all of these nodes, we again use constructor functions from the module `RewriteHelpers` to enhance readability and avoid code duplication.

4.4.4 Why-Provenance

Since why-provenance derivation is optional and should only be performed when the user requests it, the functions implementing why-provenance rewrites are all located in the separate module `WhyProvenance`, and are called by `RewriteRules2` when the `OperSem` environment is set to `True`. They need the `WhyParams` record stored in the rule state, which is defined as follows:

```
data WhyParams = NoWhy | WhyParams{
  whyV :: Integer,        -- Id of why var currently in scope
  nxtV :: Integer,        -- Free id for the next new variable
  oldVars :: [PLpgVar],   -- Old variables (needed for FOR-IN rewrite)
  newVars :: [PLpgVar] }  -- All the new vars declared in nested blocks
```

It is set to `NoWhy` when no why-provenance was requested or when not in a Phase 2 PL/pgSQL function definition. Otherwise, it contains information needed to declare new y_v variables.

Just like the function call site φ_v, the cumulative why-provenance y_v is declared as a function argument. However, it may be shadowed in inner `DECLARE` blocks. Even though these shadowing variables have the same name as the outer y_v, they are all distinct and require new variable ids. Thus, why-provenance derivation needs to insert new variables into the AST. Since new variables can only be inserted at the outer node of the AST which owns the `dataArea`, we need to collect all newly introduced variables inside the state, so that the transformation for function definitions can inject them into the `dataArea` afterwards. Thus, `WhyParams` does not only store the current y_v id `whyV`, but also the next available variable id `nxtV` and the newly declared variables `newVars`[2].

Figure 4.6 shows the declaration of the function `whyPLpg` from the `WhyProvenance` module and its implementation for `IF` nodes. Note that `whyPLpg` is not a `Rule`, but a regular function, because it is an extension of the functionality of `rewrite2PLpg` rather than a rewrite rule of its own. It takes the `WhyParams`, the `IF` condition query `c` from *before it was rewritten* and the `IF` node `p''` after the regular where-provenance rewrite. We need `c` in addition because it has been removed from `p''` by the rewrite rules. The function applies the why-provenance transformation and returns the transformed `p''` and the newly introduced variables.

As imposed by IfThenElse, the `IF` statement is wrapped into a block with y_v being shadowed in the `DECLARE` block. The new y_v is initialized as $y_v \cup Y(c)$ (as encoded by `unionVal`). There is only one newly declared variable, namely our new y_v, whose full specification is placed in the list `vars` as a `VAR` node with name `whyVarName` (`"why_v"`), id `whyVar` (as extracted from the `WhyParams` field

[2]In addition, the `oldVars` are needed for the `FOR-IN` transformation, but this will not be discussed here.

```
whyPLpg :: WhyParams
        -> Maybe Query
        -> PLpgSQL a Type Query
        -> (PLpgSQL a Type Query, [PLpgVar])
whyPLpg WhyParams{..} (Just c) p''@IF{}
  = (block, vars)
    where
      whyVar = nxtV
      block = BLOCK{ blockname = Nothing,
                     initvarnos = [whyVar],
                     blockstmts = [p''] }
      unionVal = mapTargetEx ((unionWhyVar whyVar) . toY) c
      vars = [plpgVarInit whyVarName
                          whyVar
                          psetType
                          (selectViaSublink unionVal)]
```

Figure 4.6: Why-Provenance transformation of `IF` node.

`nxtV`), type `PSET` and initial value `unionVal`. Just like the other rewrite modules, `WhyProvenance` uses several constructor functions from `RewriteHelpers` to build the new AST nodes.

Since `whyPLpg` is not a `Rule`, it cannot update the `OperSem`'s state. This responsibility lies with the calling `rewrite2PLpg`. Figure 4.7 shows the full implementation of `rewrite2PLpg` for `IF` with why-provenance handling included.

In addition to φ_v, `rewrite2PLpg` also retrieves the `WhyParams` from the state and `ask`s for the environment that encodes if why-provenance transformations should be applied. We know that `IF` will insert one new variable and that the y_v of inner nodes is the one `IF` introduces with id `nxtV`. Thus, before applying `transformM`, the `WhyParams` for that call are updated accordingly.

The actual transformation only takes place after the where-provenance rewriting has been applied. The `newWhyVars` are extracted from the result of `whyPLpg` and the previous (outer) y_v is obtained from the old `WhyParams`. Then, the current `WhyParams` (that also contain the updates from the `transformM` call) are fetched and updated by writing `whyV` back to the old shadowed y_v and adding the newly declared variable to the `newVars` field. Only afterwards the rewritten node is returned.

```
rewrite2PLpg PLLabel{ plLabel = l,
                      plLabelArg = p@IF{ exprif = Just q } } =
  do
    phi <- gets phiV
    why <- gets whyParams
    doWhy <- ask
    q' <- rewrite2 q
    when doWhy (do
      state <- get
      put state{ whyParams = why{ whyV = nxtV why,
                                  nxtV = (nxtV why) + 1 } }
    p' <- PLTF.transformM rewrite2PLpg p
    let p'' = p'{
      exprif = Just $ (emptyQueryBlock CMD_SELECT){
        select = [ eTargetEx
                   (sublinkOf Exists $
                              selectLogCall readFilterFunc l phi [])
                   boolType
                   "found" ] } }
    if doWhy
      then do
        let (p_why, newWhyVars) = whyPLpg why (Just q') p''
        let oldWhyV = whyV why
        state <- get
        let why = whyParams state
        put state{
          whyParams = why{whyV = oldWhyV,
                          newVars = newWhyVars ++ (newVars why)} }
        return p_why
      else do return p''
```

Figure 4.7: Phase 2 implementation of IfThenElse with why-provenance derivation (where-provenance parts grayed out).

5 Evaluation

This chapter seeks to investigate the performance overhead for a query imposed by the provenance computation. For this, I compare the running times of the original query to those of its Phase 𝟙 and 𝟚 rewrites, as produced by `PLSQLProv`, and inspect the additional disk space occupied by the decision logs. Phase 𝟙's overhead should be minor, since it mostly preserves the shape of the original query and only adds log calls on top of it. In Phase 𝟚, however, the assembly of the dependency sets could become rather costly, especially with the integer array implementation of dependency sets.

5.1 Setup

For measuring the performance of provenance derivation, I use the A* implementation that was already mentioned in the introduction on different input sizes. Here, the A* algorithm is implemented as a complex PL/pgSQL function `astar` that takes a `start` and `target` node as input and computes the length of the shortest path from `start` to `target`. It operates on a graph whose edges are stored in the relation `edges` and utilizes an auxiliary table `reached` to store candidate nodes and nodes that were already processed[1].

The rewriter inserts 14 log calls into this function based on 2 Join, 1 OrderBy, 2

[1] Originally, `reached` was supposed to be a temporary table declared inside the function, but this was not supported by the `LogParser`, so `reached` is now a persistent relation that is cleared at the beginning of the function.

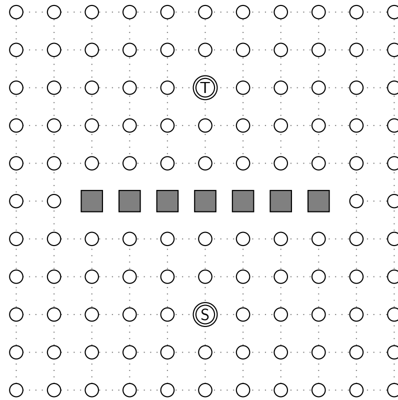


Figure 5.1: A* evaluation graph with grid size 11.

Insert, 2 Update, 1 FuncCall, 3 IfThenElse, 1 Loop, and 1 Foreach (inserts 2 log calls) rule applications. Thus, the `astar` function covers all PL/pgSQL constructs discussed in chapter 3.

For the evaluation, `astar` and its provenance rewrites are run on a two-dimensional grid of width and height n, with bidirectional edges between vertically and horizontally neighboring nodes. The nodes $(3, \lceil \frac{n}{2} \rceil)$ to $(n-2, \lceil \frac{n}{2} \rceil)$ are inaccessible to introduce an obstacle for the algorithm. Its task is to find the shortest path from $(\lceil \frac{n}{2} \rceil, 3)$ to $(\lceil \frac{n}{2} \rceil, n-2)$. Figure 5.1 shows the grid setup for $n = 11$.

The experiments were run in PostgreSQL 11.11 on a virtual Ubuntu 18.04 machine hosted by Oracle VirtualBox 6.1 with 8 GB RAM and 6 threads of an Intel Core i7-8750H CPU. Each data point presented in the next section is the average result of ten individual runs.

5.2 Results and Discussion

Figure 5.2 shows the absolute running times for the different queries. As expected, Phase $\mathbb{1}$ is only slightly slower than the original query for all grid sizes, taking up to 54 ms for grid size 39. The combination of Phase $\mathbb{1}$ and $\mathbb{2}$ without why-provenance takes much longer, its running times growing exponentially up to 2.88 s for grid size 39. The by far greatest overhead is imposed by switching on why-provenance derivation in Phase $\mathbb{2}$, which leads to a steep exponential growth of running time up to almost 7 minutes for grid size 39.

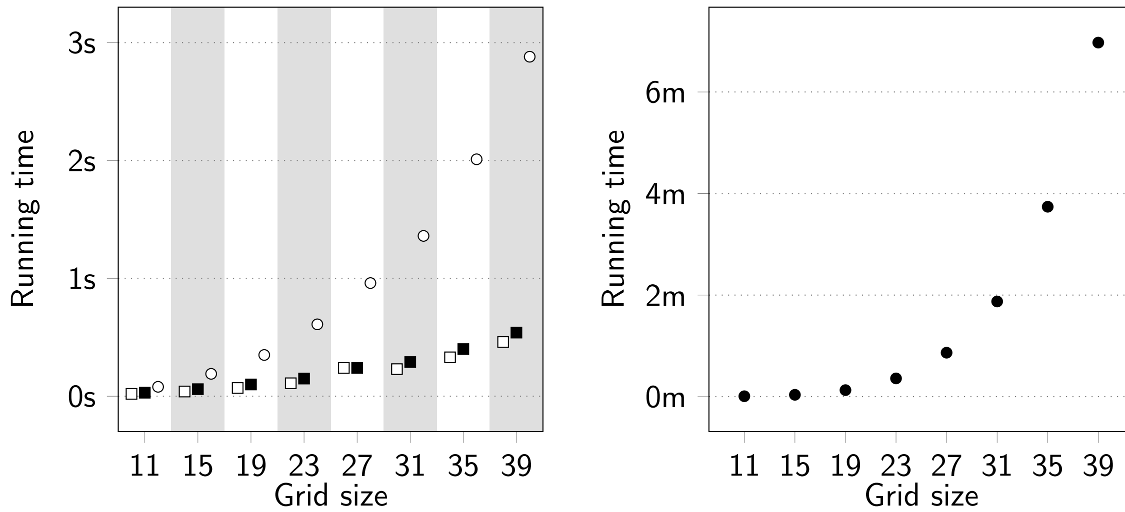
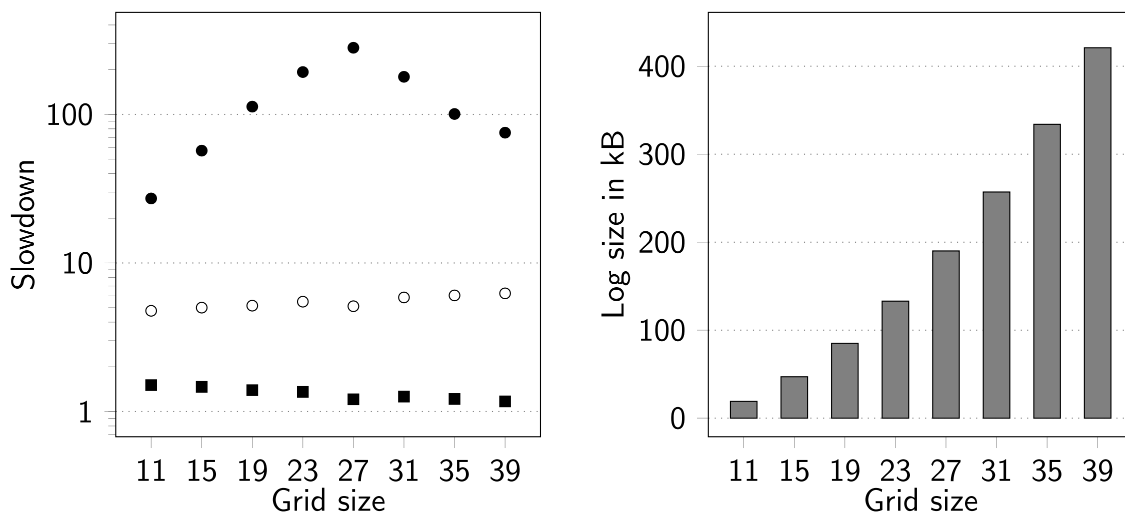


Figure 5.2: Absolute running times of the original query without provenance derivation (□), Phase 𝟙 (▪), Phase 𝟙 and 𝟚 without why-provenance (◦), and with why-provenance (•).



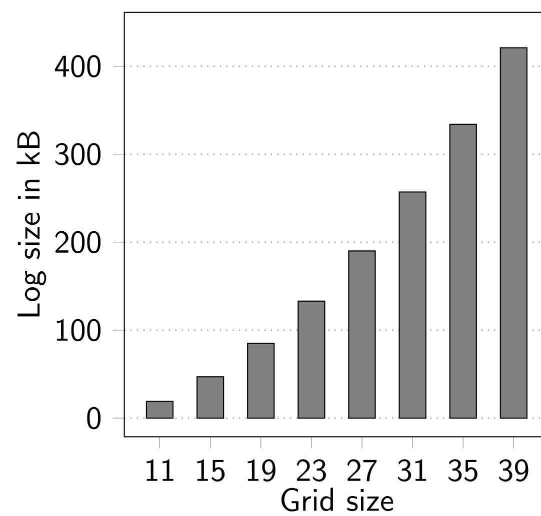


Figure 5.3: Slowdown compared to the original query of Phase 𝟙 (▪), Phase 𝟙 and 𝟚 without why-provenance (◦), and with why-provenance (•).

Figure 5.4: Disk space occupied by the log relations.

This is further illustrated by Figure 5.3, which shows the slowdown that provenance derivation causes in contrast to the original query. The impact of Phase $\mathbb{1}$ is minor and decreases with increasing input size from a factor of 1.5 for grid size 11 to a factor of just 1.17 for grid size 39. This shows that the slowdown imposed by the function itself is greater than that caused by logging.

The negative effect of Phase $\mathbb{2}$, on the other hand, increases with the input size, which makes sense, considering that a larger grid leads to larger dependency sets. When only where-provenance is assembled, however, the slowdown is moderate, going from a factor of 4.8 for grid size 11 rather linearly up to only 6.2. This is due to the low cardinality of the dependency set, which is equal to the shortest path length and is linearly incremented by 8 up to only 68 for grid size 39.

Why-provenance computation, however, leads to an exponential increase in slowdown up to grid size 27, where it peaks at a factor of 281. This is likely due to the size of the dependency sets, which ranges from 141 for grid size 11 to 1859 for grid size 39, and the higher number of $\cup$ and $\bigcup$ operations needed to build them. Curiously, the slowdown drops after grid size 27 and is down to 75.3 again for grid size 39. While I have no clear explanation for this, it is possible that PostgreSQL switched to a more efficient query plan for higher grid sizes. The peak at grid size 27 is not random; it was consistently measured during multiple runs of the evaluation script.

The average slowdown of where-provenance derivation (5.5) is comparable to the slowdown observed by Müller et al. (2018, p. 9) for the plain SQL rewrite rules (4.6). Why-provenance derivation, on the other hand, only imposes an average slowdown of 9.0 in their evaluation, while it is 128.2 for the A* function. However, I only investigate a single function here, whereas Müller et al. (2018) used all 22 queries from the TPC-H benchmark, which also showed a relatively diverse behavior with respect to performance: One of them created a slowdown of 1,039, while others showed almost no impact on performance at all. Thus, my results can be said to be similar to theirs overall.

Finally, Figure 5.4 displays the size of the logs, computed by summing up the `pg_column_size` of all rows in the log relations. They store between 19 kB (or 533 rows) for grid size 11 and 421 kB (or 11,447 rows) for grid size 39.

6 Conclusion

In this thesis, I have presented `PLSQLProv`, a system for rewriting PL/pgSQL functions such that they can derive their own provenance. The derivation process proceeds in two phases, following the approach of Müller et al. (2018) for plain SQL queries: In Phase 𝟙, the function is instrumented to log its value-based decisions while still performing the same computations and returning the same result as the original function. In Phase 𝟚, the rewritten function then interprets these logs to assemble attribute-level where- and, optionally, why-provenance.

In order to adapt the idea of Müller et al. (2018) for PL/pgSQL functions, it was necessary to introduce new variables to the rewriting process for dealing with repeated log calls from the same query inside loops or several distinct calls to the same function, and for handling the cumulative why-provenance contributed by all of the control structures that have scope over a PL/pgSQL statement. These variables had to be incorporated into the already existing rewrite rules for SQL queries from Müller et al. (2018). Also, their OrderBy rule was modified to better capture the semantics of `ORDER BY` clauses. In addition, I introduced new rewrite rules for data-modifying SQL queries and PL/pgSQL control structures.

The slowdown these rewritten functions yield compared to the original function is moderate for where-provenance derivation with an average factor of 5.5, but rather high when why-provenance is assembled as well (128.2). Whether this is specific to the A* example used for testing or a general issue with PL/pgSQL provenance derivation needs to be investigated further. However, it appears to be

a good decision to make why-provenance derivation an optional feature that can be switched on for simpler functions or when really needed for debugging.

6.1 Future Work

The main bottleneck of the provenance computation presented here is the implementation of dependency sets, which was already noted by Müller et al. (2018). Integer arrays are not duplicate-free by design, so the set operations $\cup$ and $\bigcup$ require expensive explicit duplicate elimination every time they are called. Running Phase 2 queries, especially with why-provenance derivation enabled, should become much faster using a more efficient dependency set implementation. Müller et al. (2018, p. 18f) noted significant improvements in running time when using a bit set representation based on *roaring bitmaps*. Slight performance gains should also be possible when filtering out obvious duplicates in long union chains already inside the rewriter, as opposed to simply replacing all operators by $\cup$ without inspecting the operands.

Another issue that does not impact running times but disk space usage is the duplication of input relations (and replacement of their values by dependency sets) in Phase 2. While the tables used in the examples in this thesis and also for the A* algorithm are relatively small, SQL tables can easily contain gigabytes of data. It could be worthwhile to experiment with the use of `VIEW`s instead, though these might then have negative effects on running time again.

Finally, the rewrite rules presented in chapter 3 did not cover `DELETE` statements. While deletion does not produce any provenance itself (its targets are deleted, after all), it may remove tuples referenced by dependency sets created in previous statements, rendering these dependency sets meaningless. On the other hand, new tuples may be `INSERT`ed and constitute provenance during the same function. This is why simply creating a copy or view of all input relations before function execution and referencing these copies during Phase 2 is not feasible either. To complicate things even further, `UPDATE` statements may alter tuples at any time, erasing the value that contributed to the provenance while preserving its tuple identifier.

Thus, the cell identifiers used in dependency sets should actually refer not only to

a specific cell in the input, but to a specific cell at a specific time. Implementing this would require us to keep deleted or updated entries and annotate them as 'obsolete', while inserting the updated rows with fresh identifiers. However, this considerably changes the semantics of the original function and distances us further from the goal of Phase 𝟙 being equivalent to the original with just logging added as a side effect. Also, we would then need to remove the 'obsolete' rows after Phase 𝟚, since other non-rewritten queries or functions may need to access the affected relations, unable to recognize the 'obsolete' rows as such. However, the two-phase approach does not require the two phases to be executed directly one after the other. Since the logging information is persisted, it is well possible to execute other unrelated queries in between. Handling deletions and updates in PL/pgSQL provenance derivation is thus still an open problem.

Bibliography

Arab, Bahareh Sadat, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng (2018). "GProM - A Swiss Army Knife for Your Provenance Needs." In: *Bulletin of the Technical Committee on Data Engineering* 41.1, pp. 51–62.

Benjelloun, Omar, Anish Das Sarma, Chris Hayworth, and Jennifer Widom (2006). "An Introduction to ULDBs and the Trio System." In: *Bulletin of the Technical Committee on Data Engineering* 29.1, pp. 5–16.

Buneman, Peter, Sanjeev Khanna, and Wang-Chiew Tan (2001). "Why and Where: A Characterization of Data Provenance." In: *Lecture Notes in Computer Science, International Conference on Database Theory*. Vol. 1973. Springer, pp. 316–330.

Cheney, James, Laura Chiticariu, and Wang-Chiew Tan (2009). "Provenance in databases: Why, How, and Where." In: *Foundations and Trends® in Databases* 1.4, pp. 379–474.

Chiticariu, Laura, Wang-Chiew Tan, and Gaurav Vijayvargiya (2004). *DBNotes: A Post-It System for Relational Databases based on Provenance*. Tech. rep.

Cui, Yingwei, Jennifer Widom, and Janet L. Wiener (2000). "Tracing the Lineage of View Data in a Warehousing Environment." In: *ACM Transactions on Database Systems* 25.2, pp. 179–227.

Glavic, Boris and Gustavo Alonso (2009). "Perm: Processing provenance and data on the same data model through query rewriting." In: *Proceedings of the 25th International Conference on Data Engineering*. Shanghai, China, pp. 174–185.

Herschel, Melanie, Ralf Diestelkämper, and Houssem Ben Lahmar (2017). "A survey on provenance: What for? What form? What from?" In: *The VLDB Journal* 26.6, pp. 881–906.

Hirn, Denis (2017). "Compilation of SQL into KL." Bachelor's thesis. University of Tübingen.

Müller, Tobias, Benjamin Dietrich, and Torsten Grust (2018). "You Say 'What', I Hear 'Where' and 'Why' – (Mis-)Interpreting SQL to Derive Fine-Grained Provenance." In: *arXiv preprint, arXiv:1805.11517*.

Paradzik, Gabriel (2018). "Language-Level Provenance Analysis of SQL." Bachelor's thesis. University of Tübingen.

The PostgreSQL Global Development Group (2021). *PostgreSQL 11.11 Documentation*. URL: `https://www.postgresql.org/docs/11/` (visited on 02/17/2021).

Senellart, Pierre, Louis Jachiet, Silviu Maniu, and Yann Ramusat (2018). "ProvSQL: Provenance and Probability Management in PostgreSQL." In: *Proceedings of the VLDB Endowment (PVLDB)*. Vol. 11. 12. VLDB Endowment, pp. 2034–2037.

Stack contributors (2020). *The Haskell Tool Stack*. URL: `https://haskellstack.org` (visited on 02/22/2021).