Detached Provenance Analysis

Dissertation
der Mathematisch–Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Tobias Müller
aus Tübingen

Tübingen
2019

Gedruckt mit Genehmigung der Mathematisch–Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	5. März 2020
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Torsten Grust
2. Berichterstatter:	Dr. Boris Glavic, Assoc. Prof.

Abstract

Data provenance is the research field of the algorithmic derivation of the source and processing history of data. In this work, the derivation of *Where– and Why–provenance* in *sub–cell–level granularity* is pursued for a *rich SQL dialect*. For example, we support the provenance analysis for individual elements of nested rows and/or arrays. The SQL dialect incorporates window functions and correlated subqueries.

We accomplish this goal using a novel method called *detached provenance analysis*. This method carries out a SQL–level rewrite of any user query Q, yielding $(Q^{\mathbb{1}}, Q^{\mathbb{2}})$. Employing two queries facilitates a *low–invasive* provenance analysis, i.e. both queries can be evaluated using an unmodified DBMS as backend. The queries implement a *split of responsibilities*: $Q^{\mathbb{1}}$ carries out a runtime analysis and $Q^{\mathbb{2}}$ derives the actual data provenance. One drawback of this method is that a synchronization overhead between $Q^{\mathbb{1}}$ and $Q^{\mathbb{2}}$ is induced. Experiments quantify the overheads based on the TPC-H benchmark and the PostgreSQL DBMS.

A second set of experiments carried out in row–level granularity compares our approach with the *PERM* approach (as described by B. Glavic et al.). The aggregated results show that basic queries (typically, a single SFW expression with aggregations) perform slightly better in the *PERM* approach while complex queries (nested SFW expressions and correlated subqueries) perform considerably better in our approach.

Zusammenfassung

Data Provenance ist das Forschungsgebiet, auf dem man sich mit der Berechnung von Herkunft und Verarbeitungshistorie von Daten beschäftigt. In dieser Arbeit verfolgen wir das Ziel, die Where– und Why–Provenance in feiner Granularität und für einen reichen SQL Dialekt zu berechnen. Der SQL Dialekt beinhaltet Window Functions und korrelierte Subqueries.

Wir erreichen dieses Ziel mittels eines neuen Ansatzes, genannt *Detached Provenance Analysis*. Dieser Ansatz basiert darauf, eine gegebene Query Q umzuschreiben in $Q^{\mathbb{1}}$ und $Q^{\mathbb{2}}$. Durch die Verwendung zweier Queries ist es möglich, eine minimalinvasive Provenance–Analyse durchzuführen. Mit anderen Worten, beide Queries können von einem unmodifizierten DBMS berechnet werden. Dies ist möglich dank einer klaren Aufgabentrennung. $Q^{\mathbb{1}}$ führt eine leichtgewichtige Laufzeitanalyse durch und $Q^{\mathbb{2}}$ berechnet die eigentliche Data Provenance. Ein Nachteil dieses Ansatzes ist ein zusätzlicher Synchronisierungsaufwand zwischen $Q^{\mathbb{1}}$ und $Q^{\mathbb{2}}$. Wir quantifizieren diesen Aufwand experimentell, basierend auf dem TPC-H Benchmark und dem PostgreSQL DBMS.

Mit weiteren Experimenten in Zeilengranularität vergleichen wir unsere Ergebnisse mit dem *PERM* Ansatz (von B. Glavic et al.). Die aggregierten Ergebnisse zeigen, dass einfache Queries etwas schneller von *PERM* ausgewertet werden. Komplexe Queries (geschachtelte SFW Ausdrücke und korrelierte Subqueries) werden mit unserem Ansatz erheblich schneller ausgewertet.

Contents

I Introduction 13

1 Data Provenance 15

1.1 Data Provenance for SQL 16

2 Research Focus and Contribution 17

2.1 Publications 21

II Fundamentals 23

3 Notation 25

4 Relational Model 29

4.1 Introductory Example 29

4.2 RM Types 29

4.3 RM Values 30

4.3.1 Base Domains 31

4.3.2 Array Domains 31

4.3.3 Row Domains 32

4.3.4 Table Domains 32

4.3.5 Database Domains 33

5 Provenance Representation 35

5.1 Motivating Example 35

5.2 Provenance Annotations 37

5.2.1 Union Operators $\cup$ and $\bigcup$ 37

5.2.2 Where–Provenance 37

5.2.3 Why–Provenance 39

5.3 Lifted Relational Model (LRM) 40

5.3.1 Integration of Provenance Annotations 40

5.3.2 LRM Types 40

5.3.3 LRM Values 41

5.3.4 $\mathsf{liftrm}(\cdot)$ 43

5.3.5 Push $\Psi(\cdot,\cdot)$ 44
5.3.6 Collect $\curlyvee(\cdot)$ 45

6 SQL and its Provenance Semantics 47

6.1 Cell Expressions 49
6.1.1 Literals ℓ 51
6.1.2 Row Constructors `ROW` 52
6.1.3 Column References $.col$ 52
6.1.4 Comparison of Row Values 52
6.1.5 Generic Operators 53
6.1.6 `ISNULL` 54
6.1.7 Variable References *var* 54
6.1.8 Array Constructors `ARRAY` 55
6.1.9 `CASE` 56
6.1.10 `EQCASE` 57
6.1.11 `TOROW` 57
6.1.12 `EXISTS` 59
6.2 Table Expressions 60
6.2.1 Table References *tab* 61
6.2.2 `VALUES` 62
6.2.3 `WITH` 62
6.2.4 `UNION ALL` 63
6.2.5 Table–Valued UDFs 63
6.3 SFW Expression 63
6.3.1 Decomposition 65
6.3.2 Joined Tables 67
6.3.3 Formalized Decomposition 69
6.3.4 e^{clause} Expressions 69
6.4 Aggregations and Window Functions 76
6.4.1 Aggregations in the Backend Dialect 76
6.4.2 `groupagg` 77
6.5 Aggregation Functions 80
6.5.1 `SUM` 81
6.5.2 `SUM`–Like Provenance Semantics 81
6.5.3 `THE` 81
6.5.4 `COUNTSTAR()` 82
6.6 Window Functions 82
6.6.1 Introductory Example 82
6.6.2 `windows` 84
6.6.3 Window Functions e^{win} 84

6.7 Data Provenance for SQL 86

III Provenance Analysis 89

7 Normalization 91

7.1 Definition 91
7.2 UDF Inlining 91
7.3 Correlation Normalization 92
7.4 SFW Normalization 95
7.4.1 Motivation and Example 95
7.4.2 Overview 96
7.4.3 `FROM` and `WHERE` 98
7.4.4 `GROUP BY`, `AGGREGATES` and `HAVING` 99
7.4.5 `WINDOWS` 100
7.4.6 `ORDER BY`, `DISTINCT ON` and `OFFMIT` 101
7.4.7 The Fully Normalized SFW Expression 102

8 Detached Provenance Analysis 103

8.1 Overview 103
8.2 Introductory Example 104
8.2.1 *One* Table Detached 106
8.2.2 *All* Tables Detached 106
8.2.3 Phase $\mathbb{1}$ 108
8.2.4 Phase $\mathbb{2}$ 110
8.3 Formalization 112
8.3.1 Row Identifiers ρ 112
8.3.2 Logging and Nested Loops 113
8.3.3 Logging Locations $\boldsymbol{\ell}$ and UDFs 113
8.3.4 RM$\mathbb{1}$ 114
8.3.5 RM$\mathbb{2}$ 115
8.3.6 SQL 115
8.3.7 Detached Provenance Analysis 117
8.4 Provenance Annotation Operators 118
8.4.1 Definitions 119
8.5 Rewrite of Cell Expressions 121
8.5.1 Literals ℓ 122
8.5.2 Row Constructors `ROW` 122
8.5.3 Column References $.col$ 123
8.5.4 Comparison of Row Values 123

8.5.5 Generic Operators . . . 124
8.5.6 ISNULL . . . 124
8.5.7 Variable Reference *var* . . . 124
8.5.8 Array Constructor ARRAY . . . 124
8.5.9 Array Length . . . 125
8.5.10 CASE . . . 125
8.5.11 TOROW . . . 128
8.5.12 EXISTS . . . 128
8.6 Rewrite of Table Expressions . . . 129
8.6.1 Table References *tab* . . . 129
8.6.2 VALUES . . . 129
8.6.3 WITH . . . 130
8.6.4 UNION ALL . . . 130
8.7 Rewrite of the SFW Expression . . . 132
8.7.1 Normalized SELECT . . . 132
8.7.2 Normalized Join . . . 132
8.7.3 Normalized ORDER BY . . . 136
8.7.4 Normalized AGGREGATES . . . 136
8.7.5 Normalized WINDOWS . . . 141

9 Experimental Evaluation 147
9.1 Implementation in PostgreSQL . . . 147
9.1.1 Query Rewrites . . . 147
9.1.2 Implementation of Provenance Annotations . . . 148
9.1.3 Implementation of Logging: Overview . . . 151
9.1.4 Implementation of Logging: Example . . . 151
9.2 Setup . . . 156
9.2.1 Hard– and Software . . . 156
9.2.2 Database . . . 156
9.3 Normalization Overhead . . . 159
9.3.1 Speedup of Q9 . . . 159
9.3.2 Speedup of Q21 . . . 160
9.3.3 Slowdown of Q16 . . . 161
9.4 Phase 𝟙 Overhead . . . 161
9.4.1 Results . . . 163
9.4.2 Speedup of Q2 . . . 164
9.4.3 Slowdown of Q18 . . . 164
9.4.4 Slowdown of Q17 . . . 167
9.4.5 Optimization . . . 168

9.5 Phase 𝟚 Overhead 169
9.5.1 Provenance Size 170
9.5.2 Absolute Performance 171
9.5.3 Performance Relative to Provenance Size 172
9.5.4 Speedup of Q19 175
9.5.5 Slowdown of Q10 175
9.6 Optimization of Phase 𝟚 176
9.6.1 $\bigcup$ Optimization 176

IV Conclusion 179

10 Related Work 181
10.1 Fine–Grained Data Lineage 182
10.2 Data Lineage 183
10.2.1 *WHIPS* Implementation 184
10.3 Why– and Where–Provenance 186
10.3.1 Why–Provenance 186
10.3.2 Where–Provenance 187
10.3.3 Comparison 187
10.3.4 *DBNotes* Implementation 189
10.4 Semirings Approach 190
10.4.1 Comparison of Data Provenance 191
10.4.2 ProvSQL 194
10.5 Traces Approach 194
10.6 Smoke Approach 194
10.7 The Detached Approach and Imperative Languages 195
10.8 *GProM* 197

11 Comparison with *PERM* 199
11.1 Implementation of Row–Level Granularity 199
11.1.1 Tables of Phase 𝟚 199
11.1.2 Queries of Phase 𝟚 199
11.1.3 Conclusion 200
11.2 PERM 200
11.2.1 Introductory Example 200
11.2.2 Provenance Semantics 203
11.3 Experiments 204
11.3.1 Setup 204
11.3.2 Comparison of Provenance Representation Sizes 204

11.3.3 Comparison of Performance 207
11.3.4 Basic Queries 208
11.3.5 Nesting–Heavy Queries 209

12 Summary 211

13 Future Work 213
13.1 `WITH RECURSIVE` 213
13.2 Set Operations 215
13.3 Slicing–Based How–Provenance 216
13.4 Parallel Query Execution 217

Bibliography 223

Acknowledgements 225

Part I

Introduction

1 Data Provenance

Data is an inevitable part of the modern life. Maybe the most critical kind is financial data which keeps track of the funds of countries, companies and individuals. Other examples are books, news, health data, shipment tracking or *meta data*. The last one is nothing else but data about data. Example meta data can be as simple as the author of a book or as complex (and important for retirement planning) as the credit rating of a country or company.

This very work contributes towards the algorithmic derivation of meta data. More exactly, we focus on the derivation of *data provenance* for database queries. In their influential work, P. Buneman, S. Khanna and W. Tan [BKT01, Sec. 1] specify:

> Data provenance — sometimes called "lineage" or "pedigree" is the description of the origins of a piece of data and the process by which it arrived in a database.

An example is provided in Figure 1.1. For the concrete piece of output data ⟨John Doe⟩, which input data has contributed to its existence? Data provenance provides an answer to this question and identifies corresponding cells (denoted↶⋯↷), ignoring everything not important (denoted □).

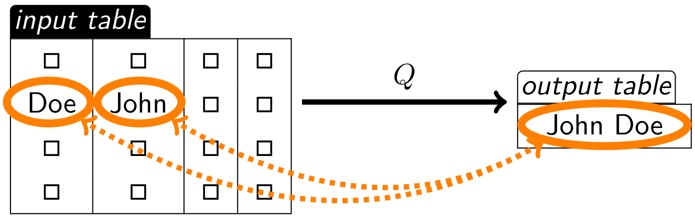


Figure 1.1: **Example: query Q yields an output table. Data provenance (↶⋯↷) identifies the input/output relationships.**

planets

name	*density*	*volume*	
Earth	⟨rocky, $5.5 \cdot 10^3$⟩	$1.1 \cdot 10^{21}$	t_1
Jupiter	⟨gaseous, $1.3 \cdot 10^3$⟩	$1.4 \cdot 10^{24}$	t_2

(a) **Example database with planetary data.**

output

planet	*mass*
Earth	$6.1 \cdot 10^{24}$
Jupiter	$1.8 \cdot 10^{27}$

(b) **Output of query Q.**

```
  FROM planets AS p
   ...
SELECT p.name AS planet,
       (p.density).numerical * p.volume AS mass
   ...
```

(c) **Query Q: what is the mass of a planet?**

Figure 1.2: **Example: analysis of output ◌, yielding data provenance ●. The units for density, volume and mass are: $[kg/m^3]$, $[m^3]$ and $[kg]$.**

1.1 Data Provenance for SQL

Next, we discuss a less obvious example of data provenance for a SQL query. Figure 1.2(a) lists the input table *planets*. The *density* of a planet is a nested row value. For example, t_1 can be denoted

$$\langle \text{Earth}, \langle \text{rocky}, 5.5 \cdot 10^3 \rangle, 1.1 \cdot 10^{21} \rangle$$

(with implicit attribute names). Query Q (see Figure 1.2(c)) is denoted in the SQL dialect from the body of this work. That dialect features an alternative order of SFW clauses (`FROM` before `SELECT` — details omitted for now). The result table is shown in Figure 1.2(b).

What data provenance has to offer is to support the database user in query understanding and debugging. Inspecting the output table (see Figure 1.2(b)) and value $6.1 \cdot 10^{24}$, our provenance analysis finds that $5.5 \cdot 10^3$ and $1.1 \cdot 10^{21}$ have contributed to its computation. Evidently, both of the input values have been transformed during query evaluation (i.e., cannot be recognized in the output table) but the methods of provenance analysis still unveil the relationships between output and the original inputs. Looking at this provenance result, the database user can gain confidence in the correctness of Q and its computation of the mass (implementing $m = \rho \cdot V$). We continue this example in the next chapter and sketch how its provenance derivation works.

2 Research Focus and Contribution

Feature-Rich SQL Dialect One main focus of our work is the provenance analysis for a feature–rich (read–only) SQL dialect. Among other features, this dialect covers bag semantics, aggregations, correlated subqueries, nested row/array expressions and window functions.

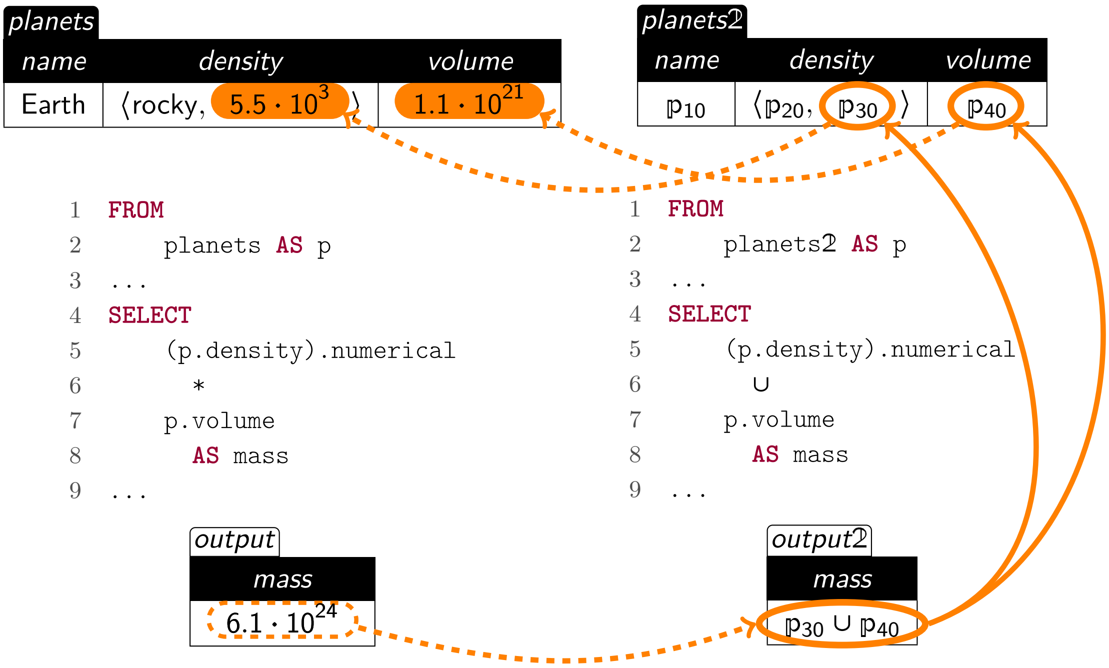
We consider the *PERM* approach by B. Glavic and G. Alonso [GA09a] the closest related work in terms of the supported SQL dialect. The implementation (also called *PERM*) supports the provenance analysis of all queries in the TPC-H benchmark. It integrates the support for correlated subqueries [GA09b] (same authors as above). *PERM*'s implementation is freely available and we carried out an experimental comparison with our approach (subject of Chapter 11). The more recent *GProM* project by B. Arab, S. Feng et al. [AFG$^+$18] may be considered a successor of *PERM* but does not support correlated subqueries.

Fine–Grained Data Provenance A well–known classification criterion for data provenance is called the provenance granularity. Our provenance analysis implements a sub–cell granularity of data provenance (already exemplified in Section 1.1). An early work by A. Woodruff and M. Stonebraker [WS97] shares this focus. A considerable share of contemporary works (more discussion carried out in Chapter 10) derives data provenance in (coarser) row granularity.

On top of that, our analysis approach allows for a switch to row granularity with little effort. We exploited this feature in the experimental comparison with *PERM*.

Provenance Notions Different types of data provenance can be distinguished. P. Buneman, S. Khanna and W. Tan [BKT01] characterize the two notions of so–called *Where*– and *Why*–provenance. In our work, we use their very intuitive terminology but with an adapted provenance semantics.

- *Where* does an output value v come from? This provenance notion encompasses all input values which have been copied or transformed in order to produce v. We call this relationship Where–provenance. Both examples from Chapter 1 yield Where–provenance.



planets

name	density	volume
Earth	⟨rocky, $5.5 \cdot 10^3$ ⟩	$1.1 \cdot 10^{21}$

planets𝟚

name	density	volume
𝕡10	⟨𝕡20, 𝕡30 ⟩	𝕡40

```
FROM
    planets AS p
...
SELECT
    (p.density).numerical
      *
    p.volume
      AS mass
...
```

```
FROM
    planets𝟚 AS p
...
SELECT
    (p.density).numerical
      ∪
    p.volume
      AS mass
...
```

output

mass
$6.1 \cdot 10^{24}$

output𝟚

mass
𝕡30 ∪ 𝕡40

(a) **Original query Q and data.**

(b) $Q^{\mathbb{2}}$ **and provenance annotations.**

Figure 2.1: **Example: values and provenance each computed separately (simplified).**

- *Why* is v an output value? This provenance notion encompasses all input values which have been inspected through predicates in order to yield v. We call this relationship Why–provenance. Our understanding of Why–provenance strongly deviates from [BKT01]. A comparison is carried out in Chapter 10.

Put in a single sentence, Where–provenance tracks data while Why–provenance tracks predicate decisions. Both notions share the same granularity and get computed in parallel. We provide definitions for a SQL dialect and its Where– and Why–provenance.

Detached Provenance Analysis (Example) The main contribution of this work is the approach called *detached provenance analysis*. We continue the example from Section 1.1 which involves nested rows. Figure 2.1(a) lists query Q and data, supposedly specified by the database user.

What we basically do in our approach is to rewrite Q into $Q^{\mathbb{2}}$ (see Figure 2.1(b)), replacing data values with provenance annotations $\mathbb{p}_\square$. We point out some important properties of Q and $Q^{\mathbb{2}}$.

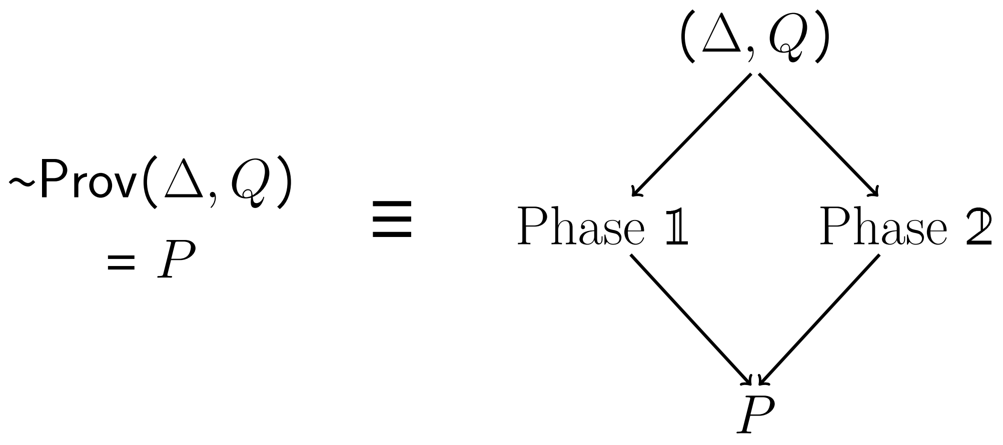


Figure 2.2: **Overview: detached provenance analysis.**

- The base tables *planets* and *planets*𝟚 share the same column names and row counts. However, *planets*𝟚 contains provenance annotations instead of data. Any $\mathbb{p}_{\square}$ is a flat set.
- The arithmetic operator $*$ (see line 6) is replaced with set union, denoted ∪. Basically, this substitution turns $*$ into the derivation of Where–provenance.
- The two result tables (*output* and *output*𝟚) share the same column name(s) and row count(s). *output*𝟚 contains the resulting data provenance.
- Both *output* and *output*𝟚 may become the input tables for subsequent database queries.

Due to symmetry in data structures, corresponding provenance annotations and data values can be associated with each other. We use dashed edges (⇢) in Figure 2.1 to connect the corresponding pairs. The solid edges (→) denote relationships between provenance annotations. Following all edges, it can be concluded that the data provenance of $6.1 \cdot 10^{24}$ is $5.5 \cdot 10^{3}$ and $1.1 \cdot 10^{21}$. This very result is the same as in Section 1.1.

The example just discussed is kept very basic for presentation reasons. In fact, the example breaks for table cardinalities > 1 (details omitted). In the body of this work, additional means are introduced which support the provenance analysis of a feature–rich SQL dialect.

Detached Provenance Analysis (Generalization) A high–level overview of our approach is provided in Figure 2.2. Let ~Prov(Δ, Q) = P be the problem of provenance analysis with Δ a database instance, Q a user query and P the resulting data provenance. The detached provenance analysis turns this task into two steps.

- Phase $\mathbb{1}$ carries out a dynamic analysis of the user query (elaborated later) and
- phase $\mathbb{2}$ carries out the provenance analysis (as exemplified in Figure 2.1(b)).

In the body of this work, we provide a definitional interpreter $\sim\mathsf{Prov}(\cdot,\cdot)$, formalizing SQL, its semantics and its provenance semantics. On that basis, SQL rewrite rules ($Q \rightsquigarrow (Q^{\mathbb{1}}, Q^{\mathbb{2}})$) are defined. The phases $\mathbb{1}$ and $\mathbb{2}$ (right–hand side of Figure 2.2) denote the evaluations of the rewritten queries. We call this method *detached provenance analysis* because the provenance derivation is carried out *detached* from the evaluation of (regular) data. A proof of equality (according to Figure 2.2) is not part of this work.

One major advantage of the detached approach (over a definitional interpreter) is that $Q^{\mathbb{1}}$ and $Q^{\mathbb{2}}$ can be evaluated using an off–the–shelf DBMS and benefit from query optimization (continued below).

Low–Invasive Provenance Analysis The detached provenance analysis produces rewritten queries ($Q^{\mathbb{1}}$ and $Q^{\mathbb{2}}$) which can be be evaluated by a DBMS backend. The DBMS core does not require any modifications which is why we call our approach *low–invasive.* In fact, the experimental evaluation provided in this work (described in Chapter 9) was carried out using an off–the–shelf PostgreSQL 9.5 DBMS. Exploiting existing DBMS infrastructure has a couple of advantages (compared to a provenance implementation from scratch).

- A modern DBMS is the result of decades of research, e.g. query optimization, indexing and caching. The detached provenance analysis leverages that research.
- The query rewrite ($Q \rightsquigarrow (Q^{\mathbb{1}}, Q^{\mathbb{2}})$) sits on a comparably high abstraction level, i.e. SQL. This makes the technology stack below SQL exchangeable (e.g., DBMS, operating system, hardware). Moreover, the detached provenance analysis will benefit from future improvements in that stack.

2.1 Publications

This thesis continues the thread of research we have started in 2013. Our findings thus far have been published in the following papers.

- T. Müller, B. Dietrich and T. Grust: *You Say 'What', I Hear 'Where' and 'Why' — (Mis-)Interpreting SQL to Derive Fine-Grained Provenance*, PVLDB 2018, [MDG18a]
- D. O'Grady, T. Müller and T. Grust: *How "How" Explains What "What" Computes — How-Provenance for SQL and Query Compilers*, TaPP 2018, [OMG18]
- B. Dietrich, T. Müller and T. Grust: *The Best Bang for Your Bu(ck)g*, EDBT 2016, [DMG16]
- T. Müller: *Have Your Cake and Eat it, Too: Data Provenance for Turing-Complete SQL Queries*, VLDB PhD Workshop 2016, [Mül16]
- T. Müller and T. Grust: *Provenance for SQL through Abstract Interpretation: Value-less, but Worthwhile*, PVLDB 2015, [MG15]
- T. Müller: *Where- und Why-Provenance für syntaktisch reiches SQL durch Kombination von Programmanalysetechniken*, GvDB 2015, [Mül15]

The full bibliographic information is provided at the end of the document. In context of this thesis, the most relevant publication is [MDG18a]. We thank Benjamin Dietrich, Daniel O'Grady and Torsten Grust for the productive and pleasant cooperation in the research and publication process.

Part II

Fundamentals

3 Notation

This chapter provides an overview of the main notations used in the document.

Meta Variables Meta variables are denoted in italic font, e.g. x or foo. Typically, subscripts x_1, x_2 are used for counters while superscripts e^{cell}, e^{table} are used to distinguish types. The square symbol $\square$ is a context–specific placeholder. For example, $x_\square$ where $\square$ is a counter variable.

Sets The elements of a set are unique and unordered.

Symbol	Example	Description
$\{\cdot\}$	$\{x_1, x_2, \ldots\}$	A set. The empty set is $\{\}$ or $\varnothing$.
$\cup$	$\{x_1\} \cup \{x_2\} = \{x_1, x_2\}$	Binary set union.
$\bigcup$	$\bigcup_{i=1}^{n} x_i$	n–ary set union.
$\mathcal{P}(\cdot)$	$\mathcal{P}(\{x1, x2\}) = \{\varnothing, \{x1\}, \ldots\}$	Power set.

Dictionaries The keys of a dictionary are unique and unordered.

Symbol	Example	Description
$\{\!\{\cdot\}\!\}$	$\{\!\{k_1 \mapsto v_1, \ldots\}\!\}$	A dictionary associating keys $k_\square$ with values $v_\square$.
$\uplus$	$\{\!\{k \mapsto v_1\}\!\} \uplus \{\!\{k \mapsto v_2\}\!\}$ $= \{\!\{k \mapsto v_2\}\!\}$	Binary union of dictionaries. For duplicate keys, the *right* dictionary dominates.

Data Provenance Data provenance is central to this work. Here, an overview of the notation is provided. The formal introduction is carried out in Chapter 5.

Symbol	Example	Description
$\square^{e}$	10^{e}	A single identifier of Wh**e**re–provenance with $\square \in \mathbb{N}$.
$\square^{y}$	20^{y}	A single identifier of Wh**y**–provenance with $\square \in \mathbb{N}$.
$\mathbb{p}$	$\mathbb{p} = \{10^{e}, 20^{y}\}$	A provenance annotation.
$\mathsf{Y}(\cdot)$	$\mathsf{Y}(\{10^{e}, 20^{y}\})$ $= \{10^{y}, 20^{y}\}$	Transform Where–provenance into Why–provenance.
$\blacktriangleleft \cdot, \cdot \blacktriangleright$	$\blacktriangleleft 42, \{10^{e}\} \blacktriangleright$	Value 42 has the provenance annotation (i.e., data provenance) $\{10^{e}\}$.

Any provenance annotation $\mathbb{p}$ is a flat set. The set operations (see above) are applicable. The union of provenance annotations (e.g. $\mathbb{p}_1 \cup \mathbb{p}_2$) is of high relevance.

Due to their central role in this work, custom formatting is employed for

- provenance annotations $\mathbb{p}$ (= a meta variable in blackboard font) and
- value/provenance pairs $\blacktriangleleft v, \mathbb{p} \blacktriangleright$ (= an ordered pair with triangular delimiters).

Types Types are denoted in small capitals, e.g. INT. The associated meta variable is τ. The type τ of a value v is denoted $\mathsf{typeof}(v) = \tau$ or $v :: \tau$. For example, $42 :: \text{INT}$.

Values and types are nested in analogous manner. For example, the type of an array value would be denoted $[42] :: \text{ARRAY}(\text{INT})$.

For convenience, we add so–called *super types* which include multiple types. Super types are decorated with an overbar. For example, $\overline{\text{ARRAY}}$ denotes all $\text{ARRAY}(\tau)$ with τ only restricted regarding to legal array types (elaborated later). For succinct notation, :: is overloaded, e.g. $[42] :: \overline{\text{ARRAY}}$. Inclusion check is denoted $\text{ARRAY}(\text{INT}) <: \overline{\text{ARRAY}}$.

Expressions have their dedicated types, e.g., $e^{table} :: \text{ETABLE}$. Inferring value types from expressions is not part of this work.

SQL Listings We employ two dialects of SQL.

- The so–called *backend dialect* is exemplified in Figure 3.1(a). This dialect (and its data provenance) is defined in Chapter 6 and is the primary dialect of this document. It is easily distinguishable through the highlighted keywords and the uncommon (but beneficial) order of SFW clauses.
- For the experimental parts of this work, we employed the PostgreSQL DBMS and discuss some concrete queries. Their formatting is exemplified in Figure 3.1(b).

```
  FROM table AS t
 WHERE ...
   ...
SELECT t.column AS result
   ...
```

(a) **Backend dialect.**

```
SELECT t.column AS result
  FROM table AS t
 WHERE ...
   ...
```

(b) **PostgreSQL dialect.**

Figure 3.1: **Examples: formatting of SQL listings.**

$$\textsc{example}\quad \frac{\mathit{premise1} \qquad \mathit{premise2}}{\mathit{env} \vdash e \Mapsto v}$$

Figure 3.2: **Notation example: natural semantics.**

Natural Semantics Natural semantics by G. Kahn [Kah87] is a formalization tool for semantics specification. It is also known as big–step semantics (or operational semantics). Natural semantics employs a set of rules. An example rule (EXAMPLE) is provided in Figure 3.2. The wide bar separates premises (above) from a single conclusion (below).

The *conclusion* can be read as: *under the environment env, expression e evaluates to value v*. In context of SQL,

- *env* would denote the visible tuple variables,
- *e* would represent the SQL expression currently being defined and
- *v* would define the result table.

A *premise* can be a predicate (hence the name). If $\neg \mathit{premise1}$, the conclusion does not hold. Alternatively, auxiliary variables can be defined (for example, the single rows constituting v). Premises are unordered.

Other The operations listed below are used to inspect / access container structures (like sets, dictionaries, rows or arrays).

Symbol	Example	Description
$\lvert\cdot\rvert$	$\lvert\varnothing\rvert = 0$	Cardinality, i.e. the count of elements.
$\cdot[\![\cdot]\!]$	$\{\!\{x \mapsto 42\}\!\}[\![x]\!] = 42$	Element retrieval (not for sets).

4 Relational Model

In this chapter, we formally introduce a *relational model* (RM) tailored for the context of this work. It is based on the pioneering works of E. Codd, starting with [Cod70]. Our flavor of the RM is designed for compatibility with the SQL dialect (subject of Chapter 6) and has bag semantics.

4.1 Introductory Example

Figure 4.1 shows an example *table* of the RM. The table goes by the name *planets* and it consists of the four *rows* (=tuples) $t_1, \ldots t_4$. Each row has the two *attributes* *name* and *density* and all rows together constitute two accordingly named *columns*.

4.2 RM Types

Central to the formalization of the relational model are the types of Definition 4.1.

planets

name	*density*	
Earth	rocky	t_1
Mars	rocky	t_2
Jupiter	gaseous	t_3
Neptune	icy	t_4

Figure 4.1: **A *table* in the relational model. The row numbers $t_\square$ are an optional meta description.**

Definition 4.1: RM Types

An **RM type** is a substitution of τ^{BASE}, τ^{CELL}, τ^{ROW}, τ^{TABLE} or τ^{DB}. The substitution rules (listed below) are applied recursively until all non–terminals have been replaced.

$$
\begin{aligned}
\tau^{\text{BASE}} &::= \text{BOOL} \\
&\mid \text{INT} \\
&\mid \text{DEC} \\
&\mid \text{TEXT} \\
\tau^{\text{CELL}} &::= \tau^{\text{BASE}} \\
&\mid \tau^{\text{ROW}} \\
&\mid \text{ARRAY}(\tau^{\text{CELL}}) \\
\tau^{\text{ROW}} &::= \text{ROW}(col_1 \mapsto \tau_1^{\text{CELL}}, \ldots col_n \mapsto \tau_n^{\text{CELL}}) \\
\tau^{\text{TABLE}} &::= \text{TABLE}(\tau^{\text{ROW}}) \\
\tau^{\text{DB}} &::= \{\!\{ tab_1 \mapsto \tau_1^{\text{TABLE}}, \ldots tab_m \mapsto \tau_m^{\text{TABLE}} \}\!\}
\end{aligned}
$$

The column identifiers col_i in $\text{ROW}(\cdot)$ are pairwise different and ordered.
The super types $\overline{\text{BASE}}$, $\overline{\text{CELL}}$, $\overline{\text{ROW}}$, $\overline{\text{TABLE}}$ and $\overline{\text{DB}}$ consist of all types which can be substituted starting from τ^{BASE}, τ^{CELL}, τ^{ROW}, τ^{TABLE} and τ^{DB}, respectively.

For example, the table from Figure 4.1 has the type

$$\text{TABLE}(\text{ROW}(\mathit{name} \mapsto \text{TEXT}, \mathit{density} \mapsto \text{TEXT}))$$

.

4.3 RM Values

The RM types according to Definition 4.1 provide the rules to construct RM values.

4.3.1 Base Domains

Definition 4.2: Base Domains

The four **base domains** are

$$\begin{aligned}\mathsf{dom}(\textsc{bool}) &\coloneqq \left\{\mathsf{true}, \mathsf{false}, \mathsf{null}^{\textsc{bool}}\right\}\\ \mathsf{dom}(\textsc{int}\ ) &\coloneqq \left\{0, +1, -1, \ldots \mathsf{null}^{\textsc{int}}\right\}\\ \mathsf{dom}(\textsc{dec}\ ) &\coloneqq \left\{0, +0.1, -0.1, \ldots \mathsf{null}^{\textsc{dec}}\right\}\\ \mathsf{dom}(\textsc{text}) &\coloneqq \left\{, \mathsf{a}, \mathsf{b},\ \ldots\ \mathsf{null}^{\textsc{text}}\right\}\end{aligned}$$

.

For example, $42 :: \textsc{int}$. The leftmost entry in the definition of $\mathsf{dom}(\textsc{text})$ is the empty text with length of zero characters (different from $\mathsf{null}^{\textsc{text}}$). The implementation details of the base domains (i.e. maximum length and precision) are kept abstract. Values of the base domains are called *base values* and considered atomic.

null Values The RM supports null values (with limitations). To avoid ambiguity, null can be annotated with type information. In total, there are $\mathsf{null}^{\textsc{bool}}$, $\mathsf{null}^{\textsc{int}}$, $\mathsf{null}^{\textsc{dec}}$ and $\mathsf{null}^{\textsc{text}}$. Put in other words, null is a (special) base value and nothing else.

4.3.2 Array Domains

Definition 4.3: Array Domains

Let $\textsc{array}(\tau)$ be a type according to Definition 4.1.
Its associated **array domain** is

$$\begin{aligned}&\mathsf{dom}(\textsc{array}(\tau)) \coloneqq\\ &\qquad \{[] :: \textsc{array}(\tau)\} \cup \{[v_1, \ldots v_n] | n \in \mathbb{N}_{>0}, v_1 \in \mathsf{dom}(\tau), \ldots v_n \in \mathsf{dom}(\tau)\}\end{aligned}$$

. The elements of an array are ordered.

For example, $[10, 20] :: \textsc{array}(\textsc{int})$ is an array value with two elements.

Arrays can be empty (i.e., $[]$) but they cannot be null. Different empty arrays may have different types.

4.3.3 Row Domains

Definition 4.4: Row Domains

Let $\textsc{row}(col_1 \mapsto \tau_1,\ \dots\ col_n \mapsto \tau_n)$ be a type according to Definition 4.1.
Its associated **row domain** is

$$\mathsf{dom}(\textsc{row}(col_1 \mapsto \tau_1, \dots col_n \mapsto \tau_n)) \coloneqq \{\langle v_1, \dots v_n\rangle \mid v_1 \in \mathsf{dom}(\tau_1), \dots v_n \in \mathsf{dom}(\tau_n)\}$$

.

As an example, $\langle \mathsf{null}^{\textsc{text}},\ 42\rangle :: \textsc{row}(\mathit{question} \mapsto \textsc{text},\ \mathit{answer} \mapsto \textsc{int})$ is a row value. Column names (i.e., attribute names) *col* are denoted in the row type and not in the value constructor. As a reminder of Definition 4.1, column names are unique and ordered.

4.3.4 Table Domains

Definition 4.5: Table Domains

Let $\textsc{table}(\tau)$ be a type according to Definition 4.1. The associated **table domain** is

$$\mathsf{dom}(\textsc{table}(\tau)) \coloneqq \{() :: \textsc{table}(\tau)\} \cup \{(v_1, \dots v_n) \mid n \in \mathbb{N}_{>0}, v_1 \in \mathsf{dom}(\tau), \dots v_n \in \mathsf{dom}(\tau)\}$$

. The elements of a table are ordered.

Due to the type constraints of Definition 4.1, table elements are exclusively rows. For example, $(\langle 42\rangle) :: \textsc{table}(\textsc{row}(\mathit{answer} \mapsto \textsc{int}))$ is a table containing a single row. Duplicate rows are possible, for example $(\langle 42\rangle, \langle 42\rangle)$ (known as bag semantics).

According to Definition 4.5, tables have ordered rows. This restriction only applies to materialized tables. We generally consider the physical operators of a DBMS to produce tables with pseudo–random row order. The performance experiments employed such physical operators.

$$\begin{aligned}(&\langle \mathsf{Earth},\ \mathsf{rocky}\rangle,\\ &\langle \mathsf{Mars},\ \mathsf{rocky}\rangle,\\ &\langle \mathsf{Jupiter},\ \mathsf{gaseous}\rangle,\\ &\langle \mathsf{Neptune},\ \mathsf{icy}\rangle)\end{aligned}$$

(a) **A table according to Definition 4.5.**

planets

name	*density*
Earth	rocky
Mars	rocky
Jupiter	gaseous
Neptune	icy

(b) **Pretty–printed table, including a table name.**

Figure 4.2: **Example: pretty–printing of tables.**

4.3.4.1 Pretty–Printed Tables

Pretty–printed tables are generally preferred. As an example for the two representations, see Figure 4.2. The pretty–printed table (on the right) integrates column names and the table name.

4.3.5 Database Domains

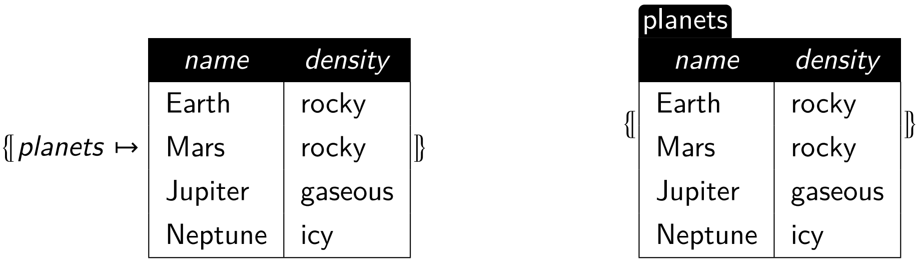
Definition 4.6: Database Domains

Let $\tau := \{\!\!\{ tab_1 \mapsto \tau_1, \dots tab_n \mapsto \tau_n \}\!\!\}$ be a type with $\tau <: \overline{\text{DB}}$ according to Definition 4.1.
Its associated **database domain** is

$$\mathsf{dom}(\tau) := \{\{\!\!\{ tab_1 \mapsto v_1, \dots tab_n \mapsto v_n \}\!\!\} \,|\, v_1 \in \mathsf{dom}(\tau_1), \dots v_n \in \mathsf{dom}(\tau_n)\}$$

.

The meta variable Δ denotes a *database instance*, i.e. $\Delta :: \overline{\text{DB}}$. An example database with a single table is provided in Figure 4.3. The tables of a database have unique names (implemented as the unique keys of a dictionary). The more compact notation on the right–hand side is preferred.



⦃*planets* ↦

name	*density*
Earth	rocky
Mars	rocky
Jupiter	gaseous
Neptune	icy

⦄

(a) **Notation with explicit ↦.**

⦃ planets

name	*density*
Earth	rocky
Mars	rocky
Jupiter	gaseous
Neptune	icy

⦄

(b) **Compact notation.**

Figure 4.3: **An example database with a single table.**

5 Provenance Representation

In this chapter, we formally define the (set–based) provenance representation. This representation is the basis for the lifted relational model (short: LRM) which is a relational model with integrated provenance annotations.

5.1 Motivating Example

planets

name	*density*	
Earth	rocky	t_1
Mars	rocky	t_2
Jupiter	gaseous	t_3
Neptune	icy	t_4

(a) **Example table.**

```
SELECT p.name AS name
  FROM planets AS p
 WHERE p.density='rocky'
```

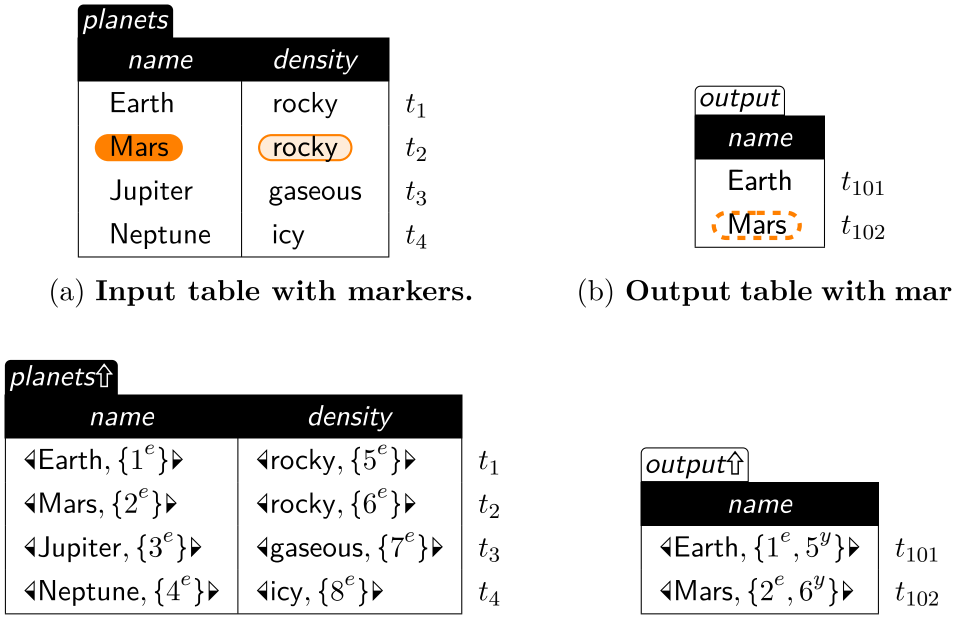
(b) **Example query Q.**

Figure 5.1: **Running example.**

Figure 5.1 lists an example table and query. For presentation reasons, the SQL query is provided in a non–formal SQL dialect. Planets Earth and Mars qualify against the `WHERE`–predicate. We are going to analyze this query and input table for two provenance notions: Where–provenance (visualized ●) and Why–provenance (visualized ○).

The visualized provenance result is listed in Figures 5.2(a) and 5.2(b). The output table contains a marker Mars which denotes a provenance inquiry for the according cell. In the input table, its data provenance is highlighted. The data provenance states that Mars and rocky contributed to the result cell in two different ways.

- The value Mars has been copied to the output Mars (modeled as Where–provenance).
- The value rocky has been inspected by a predicate in order to produce the output Mars (modeled as Why–provenance).



planets

name	*density*	
Earth	rocky	t_1
Mars	rocky	t_2
Jupiter	gaseous	t_3
Neptune	icy	t_4

(a) **Input table with markers.**

output

name	
Earth	t_{101}
Mars	t_{102}

(b) **Output table with markers.**

planets⇧

name	*density*	
$\triangleleft$Earth, $\{1^e\}\triangleright$	$\triangleleft$rocky, $\{5^e\}\triangleright$	t_1
$\triangleleft$Mars, $\{2^e\}\triangleright$	$\triangleleft$rocky, $\{6^e\}\triangleright$	t_2
$\triangleleft$Jupiter, $\{3^e\}\triangleright$	$\triangleleft$gaseous, $\{7^e\}\triangleright$	t_3
$\triangleleft$Neptune, $\{4^e\}\triangleright$	$\triangleleft$icy, $\{8^e\}\triangleright$	t_4

(c) **Lifted input table.**

output⇧

name	
$\triangleleft$Earth, $\{1^e, 5^y\}\triangleright$	t_{101}
$\triangleleft$Mars, $\{2^e, 6^y\}\triangleright$	t_{102}

(d) **Lifted output table.**

Figure 5.2: **Provenance visualization (above) and provenance annotations (below).**

We are going to switch from the visualized data provenance to the more technical perspective of provenance representation. Provenance annotations are supposed to

- reference multiple input values,
- distinguish between two different provenance notions and
- accompany every single value.

Figures 5.2(c) and 5.2(d) continue the example with all provenance annotations made explicit. The fundamental idea is to turn any value v into a pair $\triangleleft v, \mathbb{p} \triangleright$ where $\mathbb{p}$ is a provenance annotation. Tables *planets*/*planets⇧* and *output*/*output⇧* directly correspond to each other. The arrow (⇧) indicates that the table has been *lifted* and integrates provenance annotations.

- The concrete Where–provenance from above (Mars) is now represented as 2^e. Its appearance in tuples t_2 *and* t_{102} denotes the provenance relationship between the two corresponding data values. The visualization renders their relationship in color 🟧.
- The concrete Why–provenance from above (rocky) is now represented as 6^y. Why–provenance uses a conversion step, i.e. 6^y is derived from the 6^e (tuple t_2).

The visualization is aware of this, detects the correspondence between $6^y/6^e$ and renders the according marker in the color 🟧.

As a reminder, the example just discussed is only for introduction and motivation of provenance annotations (not for provenance semantics or computation). In the body of this chapter, provenance annotations and the lifted relational model get introduced formally.

5.2 Provenance Annotations

Definition 5.1: Provenance Annotation

Let $\mathbb{P} \coloneqq \{\alpha^\beta | \alpha \in \mathbb{N}_{\geq 1}, \beta \in \{e, y\}\}$ be the set of all provenance identifiers.
A **provenance annotation** is a set $\mathbb{p} \in \mathcal{P}(\mathbb{P})$.

For example, $\varnothing$ and $\{1^e, 2^y\}$ are provenance annotations.

In the definition, $\mathcal{P}(\cdot)$ denotes the powerset. The values α are restricted to natural numbers due to a design decision in the context of this work. It is not a limitation of the approach. For example, string–based identifiers $\{\mathsf{foo}^e, \mathsf{bar}^y\}$ could easily be created. Throughout the document, the meta–variable $\mathbb{p}$ is used to denote provenance annotations. The superscript letters ^{e} and ^{y} denote the provenance notion, i.e. Where–provenance and Why–provenance, respectively. More superscripts can be added if more provenance notions are to be represented. (We consider the integration of How–provenance as an item of future work, see Section 13.3.)

5.2.1 Union Operators $\cup$ and $\bigcup$

Set union is heavily utilized in our approach. For example,

$$\{1^e, 2^e\} \cup \{2^e, 3^e\} = \{1^e, 2^e, 3^e\}$$

with implicit duplicate elimination of the surplus 2^e. Different notions of provenance do not get eliminated, for example $\{1^e\} \cup \{1^y\} = \{1^e, 1^y\}$. The n–ary union operator $\bigcup$ is analogous.

5.2.2 Where–Provenance

The term *Where–provenance* originates in the work of P. Buneman, S. Khanna and W. Tan [BKT01]. Intuitively, they model copy operations of table cells (=cell granularity) as Where–provenance. An example satisfying these criteria is the query discussed in Section 5.1, which copies the value Mars from the database to the output table. However, we deviate from the understanding of P. Buneman et al. in two main aspects.

- The granularity of Where–provenance is not fixed. We use sub–cell granularity in our formalization (happening now) and row granularity for the second set of experiments (Chapter 11).
- The semantics of Where–provenance is not restricted to copy operations. Instead, value transformations (like arithmetics) are considered as Where–provenance as well. An according example can be found in Chapter 2.

Overview The formalization of Where–provenance consists of two parts.

- Generation of the initial $\square^e$ in the preparation phase of a provenance analysis. This is discussed next.
- The rules for forwarding them from the database towards the result table. This is part of Chapter 6 and happens on a per–expression basis for the SQL language.

5.2.2.1 Initial Annotations

Before a provenance analysis is carried out, the input tables get extended with *initial annotations*. As an example, see table *planets*⇧ in Figure 5.2(c). Planet Earth is annotated with a singleton set $\{1^e\}$. Creating these initial annotations is delegated to the auxiliary function $\mathsf{fresh}^{\mathbb{P}}()$. Each evaluation of $\mathsf{fresh}^{\mathbb{P}}()$ yields a singleton set $\{\square^e\}$ which is globally unique. For example,

- $\mathsf{fresh}^{\mathbb{P}}() = \{1^e\}$,
- $\mathsf{fresh}^{\mathbb{P}}() = \{2^e\}$,
- $\mathsf{fresh}^{\mathbb{P}}() = \{5^e\}$,
- $\mathsf{fresh}^{\mathbb{P}}() = \{4^e\}$,
- $\dots$.

The implementation details of $\mathsf{fresh}^{\mathbb{P}}()$ are kept abstract. It is considered to have an integer counter which is incremented on each evaluation. The provenance annotations being produced are not necessarily in ascending order (exemplified above) but unique. In this work, the management of the mandatory side–effects is kept implicit.

5.2.3 Why–Provenance

The term *Why–provenance* also originates in the work of P. Buneman, S. Khanna and W. Tan [BKT01]. A feature of Why–provenance is that it provides multiple explanations for a output values. Also, it is defined to be row–level granularity. In our work, we deviate considerably from that understanding however stick to the name. A more elaborate comparison is deferred until the discussion of related work (in Chapter 10). A short characterization of Why–provenance in the context of our work is provided below.

- Why–provenance and Where–provenance share the same granularity.
- Any Why–provenance is created through transformation of Where–provenance into Why–provenance.
- Semantically, the inspection of predicates is modeled as Why–provenance.

Overview The formalization of Why–provenance consists of two parts.

- Definition of the $\mathsf{Y}(\cdot)$ operator which turns Where–provenance in Why–provenance. This is discussed next.
- The concrete rules when $\mathsf{Y}(\cdot)$ is to be applied. This is part of Chapter 6 and happens on a per–expression basis for the SQL language.

5.2.3.1 Why Operator

Definition 5.2: Why Operator

Let $\mathbb{p}$ be a provenance annotation.
The **Why operator** is $\mathsf{Y}(\mathbb{p}) \coloneqq \{\alpha^{y} | \alpha^{\square} \in \mathbb{p}\}$.

As an example, $\mathsf{Y}(\{1^{e}, 1^{y}, 2^{y}\}) = \{1^{y}, 2^{y}\}$. Due to duplicate elimination, the cardinalities generally are $|\mathsf{Y}(\mathbb{p})| \leq |\mathbb{p}|$ for any provenance annotation $\mathbb{p}$. A reverse operation (turning Why–provenance into Where–provenance) does not exist.

The example provenance analysis from Section 5.1 already involves an application of $\mathsf{Y}(\cdot)$ for the `WHERE` predicate. In that step, $\{6^{y}\} = \mathsf{Y}(\{6^{e}\})$ (visualized: rocky $= \mathsf{Y}($rocky$)$) is computed.

planets

name	*density*
Earth	rocky
Mars	rocky
Jupiter	gaseous
Neptune	icy

(a) **Table in the RM.**

planets⇧

name	*density*
$\blacktriangleleft$Earth, $\mathbb{p}_1\blacktriangleright$	$\blacktriangleleft$rocky, $\mathbb{p}_5\blacktriangleright$
$\blacktriangleleft$Mars, $\mathbb{p}_2\blacktriangleright$	$\blacktriangleleft$rocky, $\mathbb{p}_6\blacktriangleright$
$\blacktriangleleft$Jupiter, $\mathbb{p}_3\blacktriangleright$	$\blacktriangleleft$gaseous, $\mathbb{p}_7\blacktriangleright$
$\blacktriangleleft$Neptune, $\mathbb{p}_4\blacktriangleright$	$\blacktriangleleft$icy, $\mathbb{p}_8\blacktriangleright$

(b) **Table in the LRM.**

Figure 5.3: **Example: a table with and without provenance annotations.**

5.3 Lifted Relational Model (LRM)

The *lifted relational model* (short: LRM) is an extended RM with the ability to integrate provenance annotations. As an example, see Figure 5.3. The main idea of the LRM is to turn values $v :: \overline{\textsc{base}}$ into pairs of value and an associated provenance annotation $\mathbb{p}$, denoted $\blacktriangleleft v, \mathbb{p}\blacktriangleright$. A formal introduction of the LRM is carried out in this section.

5.3.1 Integration of Provenance Annotations

Provenance annotations (defined in the previous section) are to be integrated into the relational model. For consistency, a dedicated type and domain is specified.

Definition 5.3: Provenance Annotation Type and Domain

The type of provenance annotations is PSET. The associated value domain is

$$\mathsf{dom}(\textsc{pset}) \coloneqq \mathcal{P}(\mathbb{P})$$

with $\mathbb{P}$ as in Definition 5.1.

For example, $\{1^e\} \in \mathsf{dom}(\textsc{pset})$ and $\{1^e\} :: \textsc{pset}$.

5.3.2 LRM Types

The LRM types are formalized in Definition 5.4. The (very) high resemblance with Definition 4.1 is intended. By design, the only difference between RM and LRM is the integration of provenance annotations.

Definition 5.4: LRM Types

An **LRM type** is a substitution of $\tau^{\text{BASE⇧}}$, $\tau^{\text{CELL⇧}}$, $\tau^{\text{ROW⇧}}$, $\tau^{\text{TABLE⇧}}$ or $\tau^{\text{DB⇧}}$. The substitution rules (listed below) are applied recursively until all non–terminals are replaced.

$$
\begin{array}{ll}
\tau^{\text{BASE⇧}} & ::= \blacktriangleleft \text{BOOL}, \text{PSET} \blacktriangleright \\
 & \mid\ \blacktriangleleft \text{INT}, \text{PSET} \blacktriangleright \\
 & \mid\ \blacktriangleleft \text{DEC}, \text{PSET} \blacktriangleright \\
 & \mid\ \blacktriangleleft \text{TEXT}, \text{PSET} \blacktriangleright \\
\tau^{\text{CELL⇧}} & ::= \tau^{\text{BASE⇧}} \\
 & \mid\ \tau^{\text{ROW⇧}} \\
 & \mid\ \text{ARRAY⇧}(\tau^{\text{CELL⇧}}) \\
\tau^{\text{ROW⇧}} & ::= \text{ROW⇧}(col_1 \mapsto \tau_1^{\text{CELL⇧}}, \ldots\, col_n \mapsto \tau_n^{\text{CELL⇧}}) \\
\tau^{\text{TABLE⇧}} & ::= \text{TABLE⇧}(\tau^{\text{ROW⇧}}) \\
\tau^{\text{DB⇧}} & ::= \{\!|\, tab_1 \mapsto \tau_1^{\text{TABLE⇧}}, \ldots\, tab_m \mapsto \tau_m^{\text{TABLE⇧}} \,|\!\}
\end{array}
$$

The column identifiers col_i in $\text{ROW}(\cdot)$ are pairwise different and ordered. The super types $\overline{\text{BASE⇧}}$, $\overline{\text{CELL⇧}}$, $\overline{\text{ROW⇧}}$, $\overline{\text{TABLE⇧}}$ and $\overline{\text{DB⇧}}$ consist of all types which can be substituted starting from $\tau^{\text{BASE⇧}}$, $\tau^{\text{CELL⇧}}$, $\tau^{\text{ROW⇧}}$, $\tau^{\text{TABLE⇧}}$ and $\tau^{\text{DB⇧}}$, respectively.

LRM types consistently are suffixed with an arrow (⇧) in order to distinguish them from their corresponding RM types. For example, the tables in Figure 5.3 have the types

- $\mathit{planets} :: \text{TABLE}(\text{ROW}(\mathit{name} \mapsto \text{TEXT}, \mathit{density} \mapsto \text{TEXT}))$ and
- $\mathit{planets}⇧ :: \text{TABLE⇧}(\text{ROW⇧}(\mathit{name} \mapsto \blacktriangleleft \text{TEXT}, \text{PSET} \blacktriangleright, \mathit{density} \mapsto \blacktriangleleft \text{TEXT}, \text{PSET} \blacktriangleright))$.

The triangles $\blacktriangleleft \cdot, \cdot \blacktriangleright$ are nothing but 2–tuples with custom delimiters (due to the high importance of annotations for this work).

5.3.3 LRM Values

The value domains for the LRM are composed from the value domains of the RM (defined in Section 4.3) and $\mathsf{dom}(\text{PSET})$.

Definition 5.5: LRM Base Domains

The LRM base domains are:

$$\begin{aligned}
\mathsf{dom}(\blacktriangleleft\textsc{bool}, \textsc{pset}\blacktriangleright) &:= \{\blacktriangleleft v, \mathbb{p}\blacktriangleright | v \in \mathsf{dom}(\textsc{bool}), \mathbb{p} \in \mathsf{dom}(\textsc{pset})\} \\
\mathsf{dom}(\blacktriangleleft\textsc{int}\ , \textsc{pset}\blacktriangleright) &:= \{\blacktriangleleft v, \mathbb{p}\blacktriangleright | v \in \mathsf{dom}(\textsc{int}), \mathbb{p} \in \mathsf{dom}(\textsc{pset})\} \\
\mathsf{dom}(\blacktriangleleft\textsc{dec}\ , \textsc{pset}\blacktriangleright) &:= \{\blacktriangleleft v, \mathbb{p}\blacktriangleright | v \in \mathsf{dom}(\textsc{dec}), \mathbb{p} \in \mathsf{dom}(\textsc{pset})\} \\
\mathsf{dom}(\blacktriangleleft\textsc{text}, \textsc{pset}\blacktriangleright) &:= \{\blacktriangleleft v, \mathbb{p}\blacktriangleright | v \in \mathsf{dom}(\textsc{text}), \mathbb{p} \in \mathsf{dom}(\textsc{pset})\}
\end{aligned}$$

.

For example, $\blacktriangleleft\mathsf{true}, \{1^e\}\blacktriangleright \in \mathsf{dom}(\blacktriangleleft\textsc{bool}, \textsc{pset}\blacktriangleright)$. The triangles $\blacktriangleleft\cdot, \cdot\blacktriangleright$ denote types and values as well.

Definition 5.6: LRM Array, Row, Table and DB Domains

The value domains for LRM arrays, rows, tables and databases are

$$\begin{array}{rcl}
\mathsf{dom}(\textsc{array}\Uparrow(\tau)) & := & \begin{array}{l} \{[\,] :: \textsc{array}\Uparrow(\tau)\} \cup \\ \{[\overset{\Uparrow}{v}_1, \ldots \overset{\Uparrow}{v}_n] | n \in \mathbb{N}^*, \overset{\Uparrow}{v}_1 \in \mathsf{dom}(\tau), \ldots \overset{\Uparrow}{v}_n \in \mathsf{dom}(\tau)\} \end{array} \\
\begin{array}{r} \mathsf{dom}(\textsc{row}\Uparrow(col_1 \mapsto \tau_1, \\ \ldots col_n \mapsto \tau_n)) \end{array} & := & \begin{array}{l} \{\langle col_1 \mapsto \overset{\Uparrow}{v}_1, \ldots col_n \mapsto \overset{\Uparrow}{v}_n\rangle | \\ \ \overset{\Uparrow}{v}_1 \in \mathsf{dom}(\tau_1), \ldots \overset{\Uparrow}{v}_n \in \mathsf{dom}(\tau_n)\} \end{array} \\
\mathsf{dom}(\textsc{table}\Uparrow(\tau)) & := & \begin{array}{l} \{(\!) :: \textsc{table}\Uparrow(\tau)\} \cup \\ \{(\overset{\Uparrow}{v}_1, \ldots \overset{\Uparrow}{v}_n) | n \in \mathbb{N}^*, \overset{\Uparrow}{v}_1 \in \mathsf{dom}(\tau), \ldots \overset{\Uparrow}{v}_n \in \mathsf{dom}(\tau)\} \end{array} \\
\begin{array}{r} \mathsf{dom}(\textsc{db}\Uparrow(tab_1 \mapsto \tau_1, \\ \ldots tab_n \mapsto \tau_n)) \end{array} & := & \begin{array}{l} \{\{\!\!\{ tab_1 \mapsto \overset{\Uparrow}{v}_1, \ldots tab_n \mapsto \overset{\Uparrow}{v}_n \}\!\!\} | \\ \ \overset{\Uparrow}{v}_1 \in \mathsf{dom}(\tau_1), \ldots \overset{\Uparrow}{v}_n \in \mathsf{dom}(\tau_n)\} \end{array}
\end{array}$$

. The elements of an array and of a table are ordered.

As an example, see table *planets*$\Uparrow$ in Figure 5.3(b). For better readability, table names are suffixed with an arrow ($\Uparrow$).

It is noteworthy that only the LRM base domains (see Definition 5.5) have provenance annotations associated directly. The domains in Definition 5.6 do not have their dedicated annotations but embed those from the base domains.

5.3.4 $\mathsf{liftrm}(\cdot)$

An existing RM database is turned into its LRM equivalent before a provenance analysis can be carried out. The auxiliary function $\mathsf{liftrm}(\cdot)$ takes care of this task.

Definition 5.7: Lift Operator

Let $v :: \tau$ be an RM value (according to Section 4.3) with τ one of the types $\overline{\text{BASE}}$, $\overline{\text{ARRAY}}$, $\overline{\text{ROW}}$, $\overline{\text{TABLE}}$ or $\overline{\text{DB}}$.
The **lift operator** is

$$\mathsf{liftrm}(v) \coloneqq \begin{cases} \blacktriangleleft v, \mathsf{fresh}^{\mathbb{P}}() \blacktriangleright & , v :: \overline{\text{BASE}} \\ \textbf{let } [v_1, \ldots] \coloneqq v \\ \ \textbf{in } [\mathsf{liftrm}(v_1), \ldots] & , v :: \overline{\text{ARRAY}} \\ \textbf{let } \langle col_1 \mapsto v_1, \ldots \rangle \coloneqq v \\ \ \textbf{in } \langle col_1 \mapsto \mathsf{liftrm}(v_1), \ldots \rangle & , v :: \overline{\text{ROW}} \\ \textbf{let } \llparenthesis v_1, \ldots \rrparenthesis \coloneqq v \\ \ \textbf{in } \llparenthesis \mathsf{liftrm}(v_1), \ldots \rrparenthesis & , v :: \overline{\text{TABLE}} \\ \textbf{let } \{\!\!|\, tab_1 \mapsto v_1, \ldots |\!\!\} \coloneqq v \\ \ \textbf{in } \{\!\!|\, tab_1{\Uparrow} \mapsto \mathsf{liftrm}(v_1), \ldots |\!\!\} & , v :: \overline{\text{DB}} \end{cases}$$

.

For example, $\mathsf{liftrm}(\cdot)$ applied to a database may yield

$$\mathsf{liftrm}\left(\left\{\!\!\left| \begin{array}{|c|c|} \hline \multicolumn{2}{|l|}{\textit{planets}} \\ \hline \textit{name} & \textit{density} \\ \hline \text{Earth} & \text{rocky} \\ \vdots & \vdots \\ \hline \end{array} \right|\!\!\right\}\right) = \left\{\!\!\left| \begin{array}{|c|c|} \hline \multicolumn{2}{|l|}{\textit{planets}{\Uparrow}} \\ \hline \textit{name} & \textit{density} \\ \hline \blacktriangleleft \text{Earth}, \{1^e\} \blacktriangleright & \blacktriangleleft \text{rocky}, \{5^e\} \blacktriangleright \\ \vdots & \vdots \\ \hline \end{array} \right|\!\!\right\}$$

. The order of the generated provenance annotations ($\{1^e\}, \ldots \{n^e\}$) is non–deterministic. This is not an issue. In our analysis context, the lifting step is carried out only once per database.

RM types get lifted to their corresponding LRM types. The example from directly above yields

- *planets* $::$ TABLE(ROW(*name* $\mapsto$ TEXT, *density* $\mapsto$ TEXT)) and

- $\mathit{planets}\Uparrow :: \text{TABLE}\Uparrow(\text{ROW}\Uparrow(\mathit{name} \mapsto \blacktriangleleft\text{TEXT}, \text{PSET}\blacktriangleright, \mathit{density} \mapsto \blacktriangleleft\text{TEXT}, \text{PSET}\blacktriangleright))$.

Converting the types from Definition 4.1 (RM) into Definition 5.4 (LRM) is a straight–forward task by design. Compared to the lifting of values, there is no non–deterministic component. The RM/LRM types $\text{ROW}(\cdot)$ and $\text{ROW}\Uparrow(\cdot)$ are supposed to keep the order of their attributes. We skip the formalization.

5.3.5 Push $\Psi(\cdot,\cdot)$

The LRM supports nested data structures. For example, rows can be nested arbitrarily deep. For provenance analysis, this makes additional recursive operators necessary.

Definition 5.8: Push Operator

Let $v^{\Uparrow} :: \tau$ with τ one of the types $\overline{\text{BASE}\Uparrow}$, $\overline{\text{ARRAY}\Uparrow}$, $\overline{\text{ROW}\Uparrow}$ or $\overline{\text{TABLE}\Uparrow}$. Let $\mathbb{p} :: \text{PSET}$ be a provenance annotation.
The **push operator** is

$$\Psi(v^{\Uparrow}, \mathbb{p}) := \begin{cases} \textbf{let } \blacktriangleleft v, \mathbb{p}_* \blacktriangleright := v^{\Uparrow} \\ \textbf{ in } \blacktriangleleft v, \mathbb{p}_* \cup \mathbb{p} \blacktriangleright & , v^{\Uparrow} :: \overline{\text{BASE}\Uparrow} \\ \textbf{let } [v_1^{\Uparrow}, \ldots] := v^{\Uparrow} \\ \textbf{ in } [\Psi(v_1^{\Uparrow}, \mathbb{p}), \ldots] & , v^{\Uparrow} :: \overline{\text{ARRAY}\Uparrow} \\ \textbf{let } \langle \mathit{col}_1 \mapsto v_1^{\Uparrow}, \ldots \rangle := v^{\Uparrow} \\ \textbf{ in } \langle \mathit{col}_1 \mapsto \Psi(v_1^{\Uparrow}, \mathbb{p}), \ldots \rangle & , v^{\Uparrow} :: \overline{\text{ROW}\Uparrow} \\ \textbf{let } (v_1^{\Uparrow}, \ldots) := v^{\Uparrow} \\ \textbf{ in } (\Psi(v_1^{\Uparrow}, \mathbb{p}), \ldots) & , v^{\Uparrow} :: \overline{\text{TABLE}\Uparrow} \end{cases}$$

.

For example, let $v^{\Uparrow} = \langle \blacktriangleleft 42, \mathbb{p}_1 \blacktriangleright, \blacktriangleleft 43, \mathbb{p}_2 \blacktriangleright \rangle$ be a row value with two attributes and their provenance annotations. Let $\mathbb{p}_+$ be another provenance annotation. Application of $\Psi(v^{\Uparrow}, \mathbb{p}_+)$ yields

$$\begin{aligned} &\Psi(\langle \blacktriangleleft 42, \mathbb{p}_1 \quad \blacktriangleright, \blacktriangleleft 43, \mathbb{p}_2 \quad \blacktriangleright \rangle, \mathbb{p}_+) \\ = \;& \langle \blacktriangleleft 42, \mathbb{p}_1 \cup \mathbb{p}_+ \blacktriangleright, \blacktriangleleft 43, \mathbb{p}_2 \cup \mathbb{p}_+ \blacktriangleright \rangle \end{aligned}$$

. The typical use case for the push operator is the provenance analysis of predicates. For example, a `WHERE` predicate decides if a certain row $v_{\square}^{\Uparrow}$ gets filtered or not. This decision applies to all values sitting inside of $v_{\square}^{\Uparrow}$.

The push operator does not change the type of $v^{\Uparrow}$. For any $v_a^{\Uparrow} :: \tau$ and $\mathbb{p}$,

$$\Psi(v_a^{\Uparrow} :: \tau, \mathbb{p}) = v_b^{\Uparrow} :: \tau$$

.

5.3.6 Collect $\curlyvee(\cdot)$

Definition 5.9: Collect Operator

Let $v^{\Uparrow} :: \tau$ with τ one of the types $\overline{\text{BASE}\Uparrow}$, $\overline{\text{ARRAY}\Uparrow}$, $\overline{\text{ROW}\Uparrow}$ or $\overline{\text{TABLE}\Uparrow}$.
The **collect operator** is

$$\curlyvee(v^{\Uparrow}) \coloneqq \begin{cases} \begin{array}{l} \textbf{let } \blacktriangleleft v, \mathbb{p} \blacktriangleright \coloneqq v^{\Uparrow} \\ \quad \textbf{in } (v, \mathbb{p}) \end{array} & , v^{\Uparrow} :: \overline{\text{BASE}\Uparrow} \\[2ex] \begin{array}{l} \textbf{let } [v_1^{\Uparrow}, \ldots] \coloneqq v^{\Uparrow}, \\ \qquad (v_1, \mathbb{p}_1) \coloneqq \curlyvee(v_1^{\Uparrow}), \\ \qquad \vdots \\ \quad \textbf{in } ([v_1, \ldots], \bigcup_{i=1,\ldots} \mathbb{p}_i) \end{array} & , v^{\Uparrow} :: \overline{\text{ARRAY}\Uparrow} \\[2ex] \begin{array}{l} \textbf{let } \langle v_1^{\Uparrow}, \ldots v_n^{\Uparrow} \rangle \coloneqq v^{\Uparrow}, \\ \qquad (v_1, \mathbb{p}_1) \coloneqq \curlyvee(v_1^{\Uparrow}), \\ \qquad \vdots \\ \quad \textbf{in } (\langle v_1, \ldots \rangle, \bigcup_{i=1,\ldots} \mathbb{p}_i) \end{array} & , v^{\Uparrow} :: \overline{\text{ROW}\Uparrow} \\[2ex] \begin{array}{l} \textbf{let } \Lbag v_1^{\Uparrow}, \ldots v_n^{\Uparrow} \Rbag \coloneqq v^{\Uparrow}, \\ \qquad (v_1, \mathbb{p}_1) \coloneqq \curlyvee(v_1^{\Uparrow}), \\ \qquad \vdots \\ \quad \textbf{in } (\Lbag v_1, \ldots \Rbag, \bigcup_{i=1,\ldots} \mathbb{p}_i) \end{array} & , v^{\Uparrow} :: \overline{\text{TABLE}\Uparrow} \end{cases}$$

.

As an example, the collect operator is used in the provenance definition of the `ORDER BY` clause. Let

$$v^{\Uparrow} \coloneqq [\blacktriangleleft \mathsf{abc}, \{1^{e}, 3^{e}\} \blacktriangleright, \blacktriangleleft \mathsf{def}, \{2^{e}, 3^{e}\} \blacktriangleright]$$

be an array value utilized as sort criterion. The collect operator yields

$$\curlyvee(v^{\Uparrow}) = ([\mathsf{abc}, \mathsf{def}], \{1^{e}, 2^{e}, 3^{e}\})$$

. The components of the result are:

- $[\mathsf{abc}, \mathsf{def}]$, which can be handed over to the low–level sort routine. This ensures that the provenance annotations are not considered as additional sort criteria.
- $\{1^{e}, 2^{e}, 3^{e}\}$, which can be processed independently from the sort operation and re–integrated afterwards.

6 SQL and its Provenance Semantics

This chapter builds upon the lifted relation model (LRM) introduced previously. We are about to formalize a SQL dialect in form of a definitional interpreter. Any SQL expression being defined yields an LRM value, i.e. the regular result value augmented with data provenance.

Related Work SQL is based on SEQEL by D. Chamberlin and R. Boyce [CB74]. For example, an original feature of SEQEL is the characteristic `SELECT` ... `FROM` ... `WHERE` ... syntax. The first commercial DBMS products with SQL implementations were the *Oracle Database* (1979) and IBM's *DB2* (1983). In 1986, the first SQL (ANSI) standard appeared. It got revised and (massively) extended over the past decades. Today, SQL is considered the dominant query language in the field of relational databases. Looking only at the most recent part 2 of the standard (which focuses on the SQL language itself), the document [SQL16] stretches over $1,700$ pages. We cover a small SQL dialect in our provenance definition. The provenance definition is based on annotation propagation. We consider an early work by Y. Wang and S. Madnick [WM90] the first in employing this technique.

Language Context The dialect of SQL we are about to define is also called *backend dialect*. Figure 6.1 puts it in context. In comparison to the frontend dialect (which is

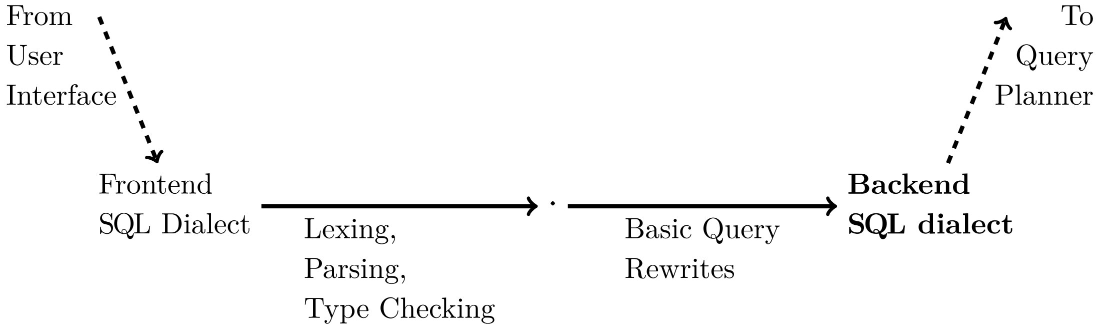


Figure 6.1: **The context of the (backend) SQL dialect. Dashed edges hint towards the previous/next processing steps.**

exposed to database users), the lexing, parsing and type checking steps have been carried out already (and succeeded). The so–called *basic query rewrites* enable us to simplify the succeeding formalization steps. For example, the rewrites expand `SELECT *` to actual column names. From the perspective of this work, the backend dialect is the starting point. The backend dialect and its provenance semantics are formalized in the body of this chapter.

SQL Expressions SQL expressions are categorized according to their result types.

- Cell expressions $e^{cell} :: \textsc{ecell}$ (to be defined) evaluate to cell values of type $\overline{\textsc{cell}\Uparrow}$ (see Definition 5.4).
- Table expressions $e^{table} :: \textsc{etable}$ (to be defined) evaluate to table values of type $\overline{\textsc{table}\Uparrow}$ (see Definition 5.4).
- Additional expressions are employed for aggregations (e^{agg}), window functions (e^{win}) and the SFW expression gets decomposed into its clauses (e^{clause}).

UDFs The dialect supports cell–valued ($\overline{\textsc{cell}\Uparrow}$) UDFs and table–valued ($\overline{\textsc{table}\Uparrow}$) UDFs. Both are restricted to cell–valued arguments. UDFs are stored in a (read–only) UDF environment F. Its type is

$$\overline{\textsc{udfs}} \coloneqq \{\!\!\{ udf \mapsto (xs, \textsc{ecell}) \}\!\!\} \uplus \{\!\!\{ udf \mapsto (xs, \textsc{etable}) \}\!\!\}$$

where

- udf is the name of a UDF,
- xs is an ordered list of parameter names and
- $\textsc{ecell}$ (or $\textsc{etable}$) is the UDF body.

The symbol $\uplus$ denotes the combination of two types, which makes $\overline{\textsc{udfs}}$ a super type.

Data Provenance Our provenance definition for SQL distinguishes between Where–provenance and Why–provenance (introduced in Section 5.2). One contribution of our work is that both notions can be mixed. Each individual provenance annotation $\mathfrak{p}$ can contain both types of provenance. A high–level perspective is provided below.

- In a preparation step, the database Δ is augmented with provenance annotations, resulting in $\Delta^{\Uparrow} = \mathsf{liftrm}(\Delta)$. These annotations contain single identifiers of Where–provenance.

- A SQL query reads data from $\Delta^{\Uparrow}$ and the result table is assembled. The annotations (of Where–provenance) sitting in $\Delta^{\Uparrow}$ are carried along and also arrive in the result table. This technique is generally known as annotation propagation.
- Whenever SQL predicates are evaluated, the associated provenance annotation $\mathbb{p}$ is turned into Why–provenance $\mathsf{Y}(\mathbb{p})$. A transformation back into Where–provenance does not exist.

We are going to define the SQL dialect and its data provenance in parallel and on a per–expression basis. This chapter concludes with the definition of

$$\mathsf{Prov}(\mathsf{liftrm}(\Delta), Q, F) = v^{\Uparrow}$$

which formalizes the data provenance $v^{\Uparrow}$ for database Δ, query Q and UDF definitions F.

6.1 Cell Expressions

An EBNF–like grammar for the SQL cell expressions e^{cell} :: ECELL is provided in Figure 6.2. Our definition of data provenance for the individual cell expression employs the natural semantics formalism by G. Kahn [Kah87] (introduced in Chapter 3). Conclusions are denoted

$$\boxed{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e^{cell} \Mapsto^{cell}_{def} v^{\Uparrow}}$$

where

- $\Gamma^{\Uparrow} :: \{\!|\, var \mapsto \overline{\text{CELL}\Uparrow} \,|\!\}$ is the environment of tuple variables.
- $\Delta^{\Uparrow} :: \overline{\text{DB}\Uparrow}$ is the (read–only) database instance. However, more tables can be added to $\Delta^{\Uparrow}$ through the `WITH` expression.
- F contains the (read–only) UDF definitions (see below).
- e^{cell} :: ECELL is the SQL expression currently being defined.
- $v^{\Uparrow} :: \overline{\text{CELL}\Uparrow}$ is the result value with provenance annotation.

Arrows ($\square^{\Uparrow}$) denote (lifted) meta variables with integrated provenance annotations.

UDF Definitions The dialect supports cell–valued ($\overline{\text{CELL}\Uparrow}$) UDFs and table–valued ($\overline{\text{TABLE}\Uparrow}$) UDFs. Both are restricted to cell–valued arguments.

The F environment contains all UDF definitions for the current query. Its type is

$$
\begin{array}{rll}
e^{cell} ::= & \ell & \\
| & \texttt{ROW}(e^{cell}, \ldots) & \\
| & e^{cell}\,.\,col & \\
| & e^{cell} \circledast_{row} e^{cell} & \\
| & \circledast(e^{cell}, \ldots) & \\
| & \texttt{ISNULL}(e^{cell}) & \\
| & var & \\
| & \texttt{ARRAY}(e^{cell}, \ldots) & \\
| & \texttt{LENGTH}(e^{cell}) & \\
| & \texttt{CASE} & \\
 & \quad \texttt{WHEN}\ e^{cell}\ \texttt{THEN}\ e^{cell}, & \\
 & \quad \vdots & \\
 & \quad \texttt{ELSE}\ e^{cell} & \\
 & \texttt{END} & \\
| & \texttt{EQCASE}\ e^{cell} & \\
 & \quad \texttt{WHEN}\ e^{cell}\ \texttt{THEN}\ e^{cell}, & \\
 & \quad \vdots & \\
 & \quad \texttt{ELSE}\ e^{cell} & \\
 & \texttt{END} & \\
| & \texttt{TOROW}(e^{table}) & \\
| & \texttt{EXISTS}(e^{table}) & \\
| & udf(e^{cell}, \ldots) &
\end{array}
$$

Figure 6.2: **SQL cell expressions.**

$$F :: \{\!|\, udf \mapsto (xs, \textsc{ecell}) \,|\!\} \Cup \{\!|\, udf \mapsto (xs, \textsc{etable}) \,|\!\}$$

where

- *udf* is the name of an UDF,
- *xs* is a shortcut for the (ordered) list of parameter names and
- the expression types ECELL and ETABLE represent the body of a cell–valued or table–valued UDF.

The symbol $\Cup$ denotes that both types of UDFs are merged together. UDF names must be globally unique. F is a read–only data structure which is accessed via $F[\![\,\square\,]\!]$ with $\square$ being a UDF name.

6.1.1 Literals ℓ

$$\textsc{def-literal}$$
$$\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \ell \Mapsto^{cell}_{def} \blacktriangleleft \ell, \varnothing \blacktriangleright$$

Value Semantics Literal expressions ℓ expand according to Figure 6.3(a) (syntactic details omitted). The mapping to the base domains of Definition 4.2 is straightforward. For example, the boolean literals (see Figure 6.3(b)) evaluate to $\mathsf{dom}(\textsc{bool}) = \{\mathsf{true}, \mathsf{false}, \mathsf{null}^{\textsc{bool}}\}$.

Provenance Semantics The provenance annotation of literals is generally empty ($\varnothing$). This is not a limit of the approach but a question of focus. In this work, we focus on data provenance for values sitting in the database. The provenance analysis of query constants is part of our future work on the topic of How–provenance (outlined in Chapter 13).

$$\begin{aligned} \ell ::=\; & bool \\ |\; & int \\ |\; & float \\ |\; & text \end{aligned}$$

(a) **Literal non–terminals.**

$$\begin{aligned} bool ::=\; & \texttt{TRUE} \\ |\; & \texttt{FALSE} \\ |\; & \texttt{NULL} \end{aligned}$$

(b) **Boolean keywords.**

Figure 6.3: **SQL literals.**

6.1.2 Row Constructors `ROW`

$$\begin{array}{l}\text{DEF-ROW}\\ \dfrac{\left|\ \Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e_i^{cell} \Mapsto_{def}^{cell} v_i^{\Uparrow}\ \right|_{i=1..n}}{\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \mathtt{ROW(}\ e_1^{cell},\ \ldots\ e_n^{cell}\ \mathtt{)} \Mapsto_{def}^{cell} \langle v_1^{\Uparrow}, \ldots v_n^{\Uparrow}\rangle}\end{array}$$

Value Semantics The `ROW`$(\cdot)$ expression assembles a row value $\langle\cdot\rangle$. Recursion is employed to construct the individual attributes. Attribute names are implicit.

Provenance Semantics The resulting provenance annotations are nested in the individual attributes $v_{i=1..n}^{\Uparrow}$. The row itself (=a container of values) has no additional provenance annotation.

6.1.3 Column References $.\,col$

$$\begin{array}{l}\text{DEF-COL}\\ \dfrac{\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e^{cell} \Mapsto_{def}^{cell} v^{\Uparrow} \qquad v^{\Uparrow} :: \overline{\text{ROW}\Uparrow}}{\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e^{cell}.\,col \Mapsto_{def}^{cell} v^{\Uparrow}[\![\,col\,]\!]}\end{array}$$

Value Semantics The column reference $e^{cell}.\,col$ unwraps a single attribute from a row value. In consequence, e^{cell} must yield a row value (to be checked statically).

Unqualified column references (col instead of $e^{cell}.\,col$) are not supported (in the backend dialect). A query rewrite (outlined in Figure 6.1) resolves this issue.

Provenance Semantics The provenance semantics is analogous to the value semantics. Put in other words, $e^{cell}.\,col$ keeps the provenance annotations sitting in the col attribute.

6.1.4 Comparison of Row Values

$$\begin{array}{l}\text{DEF-ROWOP}\\ \dfrac{\begin{array}{lll}\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e_1^{cell} \Mapsto_{def}^{cell} v_1^{\Uparrow} & v_1^{\Uparrow} :: \overline{\text{ROW}\Uparrow} & (v_1,\mathbb{p}_1) \coloneqq \curlyvee(v_1^{\Uparrow})\\ \Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e_2^{cell} \Mapsto_{def}^{cell} v_2^{\Uparrow} & v_2^{\Uparrow} :: \overline{\text{ROW}\Uparrow} & (v_2,\mathbb{p}_2) \coloneqq \curlyvee(v_2^{\Uparrow})\\ \multicolumn{3}{c}{v \coloneqq v_1 \widehat{\circledast}_{row} v_2 \qquad v :: \text{BOOL} \qquad \mathbb{p} \coloneqq \mathbb{p}_1 \cup \mathbb{p}_2}\end{array}}{\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e_1^{cell} \circledast_{row}\, e_2^{cell} \Mapsto_{def}^{cell} \blacktriangleleft v,\mathbb{p} \blacktriangleright}\end{array}$$

Value Semantics The abstract operator $\circledast_{row}$ compares two row values with each other. It supports the default comparison operators =, >, ≥, < and ≤. The collect operator $\curlyvee(\cdot)$ (see Definition 5.9) is used to recursively extract all provenance annotations from the row values (turning $v_{\square}^{\Uparrow}$ into $v_{\square}$). Then, the low–level comparison operator $\widehat{\circledast}_{row}$ (incapable to process provenance annotations) carries out the value–only comparison operation.

Provenance Semantics The provenance definition assumes that the entire row value is relevant for the result of the comparison. Lazy implementations of $\widehat{\circledast}_{row}$ are not in the scope of this work.

The provenance computation consists of two steps.

- $\curlyvee(v_{\square}^{\Uparrow})$ recursively merges all provenance annotations sitting in the row value $v_{\square}^{\Uparrow}$ (greedy provenance semantics).
- The resulting data provenance is the union ($\mathbb{p}_1 \cup \mathbb{p}_2$) of both row values.

6.1.5 Generic Operators

$$\frac{\textsc{def-operator}\quad \left| \, \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e_i^{cell} \Mapsto_{def}^{cell} \blacktriangleleft v_i, \mathbb{p}_i \blacktriangleright \right|_{i=1..n} \qquad \left| \; v_i :: \overline{\text{BASE}} \; \right|_{i=1..n} \qquad v := \widehat{\circledast}(v_1, \dots v_n) \qquad \mathbb{p} := \bigcup_{i=1..n} \mathbb{p}_i}{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \circledast(e_1^{cell}, \dots e_n^{cell}) \Mapsto_{def}^{cell} \blacktriangleleft v, \mathbb{p} \blacktriangleright}$$

Value Semantics The generic operators (abstract notation: $\circledast$) cover a huge number of n–ary operators. Each argument $e_{\square}^{cell}$ yields a value $\blacktriangleleft v_{\square}, \mathbb{p}_{\square} \blacktriangleright$ with $v_{\square}$ atomic. Figure 6.4 provides examples of operators being supported in our SQL dialect. Most of the aliases use infix notation and may have a syntax highlighting. Many operators (e.g. +) are overloaded — types are listed on the right–hand side. Implicit type casts are not permitted. Instead, explicit casts (e.g. totext(·)) exist. The low–level implementation of $\circledast$ is denoted $\widehat{\circledast}$ and carries out the value–only computation (see DEF-OPERATOR). Lazy operators are not considered.

Provenance Semantics The data provenance of an n–ary operator is defined as the data provenance of its n arguments ($\bigcup_{i=1..n} \mathbb{p}_i$).

operator	alias	type
and(., .)	AND	$(\mathrm{BOOL}, \mathrm{BOOL}) \rightarrowtail \mathrm{BOOL}$
or(., .)	OR	$(\mathrm{BOOL}, \mathrm{BOOL}) \rightarrowtail \mathrm{BOOL}$
not(.)	NOT	$\mathrm{BOOL} \rightarrowtail \mathrm{BOOL}$
plus(., .)	+	$(\mathrm{INT}, \mathrm{INT}) \rightarrowtail \mathrm{INT}$
plus(., .)	+	$(\mathrm{DEC}, \mathrm{DEC}) \rightarrowtail \mathrm{DEC}$
round(., .)		$(\mathrm{DEC}, \mathrm{INT}) \rightarrowtail \mathrm{DEC}$
concat(., .)	||	$(\mathrm{TEXT}, \mathrm{TEXT}) \rightarrowtail \mathrm{TEXT}$
eq(., .)	=	$(\mathrm{TEXT}, \mathrm{TEXT}) \rightarrowtail \mathrm{TEXT}$
gte(., .)	>=	$(\mathrm{DEC}, \mathrm{DEC}) \rightarrowtail \mathrm{BOOL}$
totext(.)		$\mathrm{BOOL} \rightarrowtail \mathrm{TEXT}$
totext(.)		$\mathrm{INT} \rightarrowtail \mathrm{TEXT}$
totext(.)		$\mathrm{DEC} \rightarrowtail \mathrm{TEXT}$
totext(.)		$\mathrm{TEXT} \rightarrowtail \mathrm{TEXT}$
	⋮	

Figure 6.4: **Concrete examples for the generic operator ⊛.**

6.1.6 ISNULL

$$\frac{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e^{cell} \Mapsto_{def}^{cell} v^{\Uparrow} \qquad v^{\Uparrow} :: \overline{\mathrm{BASE}\Uparrow} \qquad \lhd v, \mathbb{p} \rhd := v^{\Uparrow}}{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \mathtt{ISNULL}(e^{cell}) \Mapsto_{def}^{cell} \lhd v = \mathsf{null}, \mathbb{p} \rhd}\ \text{DEF-ISNULL}$$

Value Semantics The ISNULL(e^{cell}) expression is defined separately from the generic operator ⊛ for better awareness of null semantics. Most SQL expressions just forward null values towards the result table. However, in DEF-ISNULL the two–valued meta–syntactical = is employed. When v = null is evaluated, the result is exclusively true or false.

Provenance Semantics From provenance perspective, the null value is just another value with provenance annotation $\mathbb{p}$. In any case (if null or not), this annotation is propagated to the result.

6.1.7 Variable References *var*

$$\frac{\Gamma^{\Uparrow} = \{\!|\ldots, var \mapsto v^{\Uparrow}, \ldots|\!\}}{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash var \Mapsto_{def}^{cell} \Gamma^{\Uparrow}[\![\, var \,]\!]}\ \text{DEF-VAR}$$

Value Semantics The variable reference *.var* performs a lookup in the variable environment $\Gamma^{\Uparrow}$. If the variable is not found, the query fails statically.

Provenance Semantics Provenance annotations already exist in $\Gamma^{\Uparrow}$. On variable lookup, these annotations are forwarded to the result.

6.1.8 Array Constructors `ARRAY`

$$\frac{\text{DEF-ARRAY}\qquad \Big|\ \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e_i^{cell} \Longmapsto_{def}^{cell} v_i^{\Uparrow}\ \Big|_{i=1..n}}{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \texttt{ARRAY(}\ e_1^{cell}\ \texttt{,}\ \ldots\ e_n^{cell}\ \texttt{)} \Longmapsto_{def}^{cell} [v_1^{\Uparrow}, \ldots v_n^{\Uparrow}]}$$

Value Semantics The `ARRAY(·)` expression assembles the according array value $[\cdot]$. Recursion is employed to construct the individual elements, formalized in DEF-ARRAY. This SQL dialect supports nesting of array values (instead of multi–dimensional arrays).

Provenance Semantics The resulting data provenance is nested in the individual elements $v_{i=1..n}^{\Uparrow}$ of the resulting array value $[v_1^{\Uparrow}, \ldots v_n^{\Uparrow}]$. The array itself (i.e., the container) has no additional provenance annotation.

6.1.8.1 Operations on Array Values

Let `a` and `b` be two array values. Typical operations are

- concatenation (`a || b`),
- retrieval of the *i*–th array element (`a[i]`),
- comparison of arrays (`a <= b`) or
- cardinality (`length(a)`).

The provenance semantics of array concatenation is that the individual elements keep their individual provenance annotation. The provenance semantics of element access and comparison is analogous to the semantics of row values. We skip the formalization.

Array Length

$$\frac{\text{DEF-ARRAY-LENGTH}\qquad \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e^{cell} \Longmapsto_{def}^{cell} v^{\Uparrow} \qquad v^{\Uparrow} :: \overline{\text{ARRAY}\Uparrow} \qquad (v, \mathbb{p}) \coloneqq \curlyvee(v^{\Uparrow})}{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \texttt{LENGTH}(e^{cell}) \Longmapsto_{def}^{cell} \blacktriangleleft |v|, \varnothing \blacktriangleright}$$

DEF-CASE

$$\dfrac{\begin{array}{c}\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e^{cell}_{else} \Mapsto^{cell}_{def} v^{\Uparrow}_{else}\\ \Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e^{cell}_{w_1} \Mapsto^{cell}_{def} \blacktriangleleft v_{w_1}, \mathbb{p}_{w_1} \blacktriangleright \qquad v_{w_1} :: \text{BOOL} \qquad \Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e^{cell}_{t_1} \Mapsto^{cell}_{def} v^{\Uparrow}_{t_1}\\ \Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \begin{array}{l}\texttt{CASE}\\ \quad \texttt{WHEN } e^{cell}_{w_2} \texttt{ THEN } e^{cell}_{t_2},\\ \quad \vdots\\ \quad \texttt{WHEN } e^{cell}_{w_n} \texttt{ THEN } e^{cell}_{t_n},\\ \quad \texttt{ELSE } e^{cell}_{else}\\ \texttt{END}\end{array} \Mapsto^{cell}_{def} v^{\Uparrow}_{tail}\\ v^{\Uparrow} := \begin{cases} v^{\Uparrow}_{else} & , n = 0\\ \Psi(v^{\Uparrow}_{t_1}, \Upsilon(\mathbb{p}_{w_1})) & , v_{w_1} = \mathsf{true}\\ \Psi(v^{\Uparrow}_{tail}, \Upsilon(\mathbb{p}_{w_1})) & , \textbf{else}\end{cases}\end{array}}{\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \begin{array}{l}\texttt{CASE}\\ \quad \texttt{WHEN } e^{cell}_{w_1} \texttt{ THEN } e^{cell}_{t_1},\\ \quad \vdots\\ \quad \texttt{WHEN } e^{cell}_{w_n} \texttt{ THEN } e^{cell}_{t_n},\\ \quad \texttt{ELSE } e^{cell}_{else}\\ \texttt{END}\end{array} \Mapsto^{cell}_{def} v^{\Uparrow}}$$

Figure 6.5: `CASE` **semantics.**

Value Semantics The cardinality of the array value v (as in DEF-ARRAY-LENGTH) is evaluated using the meta–syntactical cardinality operator $|v|$.

Provenance Semantics The data provenance of `LENGTH` is empty ($\varnothing$). Why is that? Intuitively, the cardinality depends on the array elements. Our opinion is that only a specific data provenance is desirable. Data provenance being empty or data provenance including everything are both not useful. The empty data provenance is preferred because of its compactness.

In rule DEF-ARRAY-LENGTH, the $\blacktriangleleft |v|, \varnothing \blacktriangleright$ can easily be switched to $\blacktriangleleft |v|, \mathbb{p} \blacktriangleright$. In a software product, this might be implemented as a configuration option.

6.1.9 `CASE`

The `CASE` semantics is listed in Figure 6.5. The input expression consists of $n \geq 0$

$$\texttt{WHEN } e^{cell}_{w_i} \texttt{ THEN } e^{cell}_{t_i}$$

branches with $i = 1, \dots n$. The first $e^{cell}_{w_i}$ being satisfied determines the resulting $e^{cell}_{t_i}$ expression. Otherwise, the `ELSE` branch is evaluated (which must always be provided).

Value Semantics The formalization of DEF-CASE uses a three–way case distinction (in meta–language).

- If no `WHEN`... branches are provided ($n = 0$), the `ELSE` branch is evaluated.
- If the first predicate $e^{cell}_{w_1}$ qualifies, the the according `THEN` expression is evaluated.
- Otherwise, recursive evaluation is carried out with branches $i = 2, \dots n$.

Take note that the expression $v_{w_1} = \mathsf{true}$ is different from v_{w_1}. The meta–syntactical $=$ is 2–valued and cannot yield null.

Provenance Semantics The `CASE` expression chooses a certain `THEN` branch (or the `ELSE` expression). The provenance annotations found in the associated subexpression are forwarded to the result.

Additionally, the `CASE` expression is the first expression which involves predicates, i.e. the $e^{cell}_{w_i}$. According to our provenance notions (see Chapter 5), predicates are modeled as Why–provenance. Thus, the Why operator ($\mathsf{Y}(\cdot)$) according to Definition 5.2 is employed. For the base case $v_{w_1} = \mathsf{true}$, the definition denotes $\mathsf{Y}(\mathbb{p}_{w_1})$. This Why–provenance is then "pushed down" the (lifted) result value, i.e. $\Psi(\overset{\Uparrow}{v}_{t_1}, \mathsf{Y}(\mathbb{p}_{w_1}))$. Through recursion, the Why–provenance gets accumulated until the first branch qualifies.

6.1.10 `EQCASE`

The `EQCASE` expression heavily resembles the `CASE` expression and may be considered syntactic sugar. The rewrite strategy shown in Figure 6.6 is employed if `EQCASE` occurs in the input query. The idea of the rewrite is to make the implicit $=$ operator explicit.

Nonetheless, `EQCASE` is considered an expression of the backend dialect. We make use of it in Chapter 8.

6.1.11 `TOROW`

$$\frac{\begin{array}{c}\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e^{table} \Mapsto^{table}_{def} \overset{\Uparrow}{v}_{table} \qquad \text{TABLE}\Uparrow(\tau) \coloneqq \mathsf{typeof}(\overset{\Uparrow}{v}_{table}) \\ \overset{\Uparrow}{v}_{row} \coloneqq \begin{cases} \mathsf{nullvalue}^{\Uparrow}(\tau) & , |\overset{\Uparrow}{v}_{table}| = 0 \\ \overset{\Uparrow}{v}_{table}[\![1]\!] & , |\overset{\Uparrow}{v}_{table}| = 1 \end{cases}\end{array}}{\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \mathtt{TOROW}(e^{table}) \Mapsto^{cell}_{def} \overset{\Uparrow}{v}_{row}}\ \text{DEF-TOROW}$$

$$\begin{array}{l}
\texttt{EQCASE}\ e_0^{cell}\\
\qquad \texttt{WHEN}\ e_{w_1}^{cell}\ \texttt{THEN}\ e_{t_1}^{cell}\ ,\\
\qquad \vdots\\
\qquad \texttt{WHEN}\ e_{w_n}^{cell}\ \texttt{THEN}\ e_{t_n}^{cell}\ ,\\
\qquad \texttt{ELSE}\ e_{else}^{cell}\\
\texttt{END}
\end{array}$$

(a) **Input expression.**

$$\begin{array}{l}
\texttt{CASE}\\
\qquad \texttt{WHEN}\ e_0^{cell}\texttt{=}e_{w_1}^{cell}\ \texttt{THEN}\ e_{t_1}^{cell}\ ,\\
\qquad \vdots\\
\qquad \texttt{WHEN}\ e_0^{cell}\texttt{=}e_{w_n}^{cell}\ \texttt{THEN}\ e_{t_n}^{cell}\ ,\\
\qquad \texttt{ELSE}\ e_{else}^{cell}\\
\texttt{END}
\end{array}$$

(b) **Rewritten expression.**

Figure 6.6: **Substitution of EQCASE with CASE.**

Value Semantics The TOROW(e^{table}) expression is an explicit cast from a table to a (single) row value. e^{table} is supposed to yield a table t with $|t| \le 1$. A runtime error is raised otherwise and the query evaluation is stopped. If $|t| = 0$, an artificial row is created with all values set to null.

Rule DEF-TOROW has a case distinction to switch between cardinalities 0 and 1. Case 1 is straight–forward but case 0 is more intricate as our SQL dialect does not support type–generic null values (just $\mathsf{null} :: \overline{\text{BASE}}$ is defined). We employ the auxiliary function $\mathsf{nullvalue}^{\Uparrow}(\tau)$ which expects an LRM type τ as argument and assembles a null–like value with type τ. For example,

$$\mathsf{nullvalue}^{\Uparrow}(\text{ROW}\Uparrow(\mathit{foo} \mapsto \text{TEXT}\Uparrow)) = \langle \blacktriangleleft \mathsf{null}, \varnothing \blacktriangleright \rangle$$

. The definition of $\mathsf{nullvalue}^{\Uparrow}(\cdot)$ would be based on a case distinction between the types according to Definition 5.4. Recursively, it turns any type into its value equivalent (padded with null and $\varnothing$). Row values retain all attributes (as exemplified above) and array values have length 0. The formalization is omitted.

Provenance Semantics The provenance semantics is analogous to the value semantics. The null values are basically created from nothing and hence, the data provenance is empty ($\varnothing$). In other words, $\mathsf{nullvalue}^{\Uparrow}(\cdot)$ produces pairs $\blacktriangleleft \mathsf{null} :: \overline{\text{BASE}}, \varnothing \blacktriangleright$ (exemplified above).

6.1.11.1 Comparison to the SQL Standard

Figure 6.7 shows a grammar piece quoted from the SQL standard. On the bottom, a *subquery* is defined consisting of a pair of parentheses and a nested expression. An according parser would carry out a three–way case distinction:

```
<scalar subquery> ::=
  <subquery>
<row subquery> ::=
  <subquery>
<table subquery> ::=
  <subquery>

<subquery> ::=
  <left paren> <query expression> <right paren>
```

Figure 6.7: **Grammar rules for subqueries according to [SQL16][p. 472].**

- keep the tabular result or
- cast it to a *row* or
- cast it to a *scalar*.

The concrete case depends on the expression context. In contrast, the backend dialect of our work is more explicit and `TOROW` always casts to a row value (second case from above). If the row value is to be unwrapped, the expression can be rewritten into `TOROW`(e^{cell}).col.

6.1.12 `EXISTS`

DEF-EXISTS

$$\frac{\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e^{table} \Rrightarrow^{table}_{def} \overset{\Uparrow}{v}_{table} \qquad v^{\Uparrow} := \begin{cases} \blacktriangleleft \mathsf{false}, \varnothing \blacktriangleright & , |\overset{\Uparrow}{v}_{table}| = 0 \\ \blacktriangleleft \mathsf{true}, \mathsf{Y}(\mathsf{snd}(\curlyvee(\overset{\Uparrow}{v}_{table}[\![1]\!]))) \blacktriangleright & , \textbf{else} \end{cases}}{\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \texttt{EXISTS}(e^{table}) \Rrightarrow^{cell}_{def} v^{\Uparrow}}$$

Value Semantics `EXISTS`(e^{table}) has a nested table expression e^{table}. The value semantics is to decide if e^{table} yields an empty table (cardinality 0) or not. A single row is enough to determine the `true` condition.

Provenance Semantics If the sub–expression yields an empty table ($|\overset{\Uparrow}{v}_{table}| = 0$), the provenance is empty ($\varnothing$) as well.

In the **else** case, a single row is extracted from the intermediate table ($\overset{\Uparrow}{v}_{table}[\![1]\!]$). The first row always exists, although being non–deterministic due to generally pseudo–random row order. This randomness may appear unsatisfying, however the value semantics of `EXISTS` also does not care about row ordering. Any row is good enough to determine the `true` case. The full term is $\mathsf{Y}(\mathsf{snd}(\curlyvee(\overset{\Uparrow}{v}_{table}[\![1]\!])))$ which carries out the transformation steps

- recursively collect all provenance annotations of the row $((v, \mathbb{p}) \coloneqq \curlyvee(\cdot))$,
- pick the $\mathtt{snd}(\cdot)$ (provenance–only) component and
- turn the provenance into Why–provenance (for `EXISTS` being a predicate).

6.1.12.1 Cell–Valued UDFs

DEF-UDF-CELL

$$\frac{\begin{array}{c} \Big|\ \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e_i^{cell} \Mapsto_{def}^{cell} v_i^{\Uparrow}\ \Big|_{i=1..n} \\ (xs,\ e^{cell}) \coloneqq F[\![\,udf\,]\!] \qquad n = |xs| \qquad \Gamma_*^{\Uparrow} \coloneqq \{\!\!\{ \Big|\ xs[\![\,i\,]\!] \mapsto v_i^{\Uparrow},\ \Big|_{i=1..n} \}\!\!\} \\ \Gamma_*^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e^{cell} \Mapsto_{def}^{cell} v^{\Uparrow} \end{array}}{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash udf\,(e_1^{cell}, \dots e_n^{cell}) \Mapsto_{def}^{cell} v^{\Uparrow}}$$

Value Semantics Each cell–valued UDF *udf* called in a SQL query has a corresponding entry in the dictionary $F = \{\!\!\{\dots, udf \mapsto (xs, e^{cell}), \dots\}\!\!\}$ or the query fails statically. The pair (xs, e^{cell}) consists of

- an ordered list of parameter names $xs = [var_1, var_2, \dots]$ and
- the (cell–valued) body of the UDF e^{cell}.

Rule DEF-UDF-CELL formalizes UDF calls. The arguments e_i^{cell} yield the values $v_i^{\Uparrow}$. A new variable environment $\Gamma_*^{\Uparrow} \coloneqq \{\!\!\{\dots, var_i \mapsto v_i^{\Uparrow}, \dots\}\!\!\}$ is assembled. Then, e^{cell} is evaluated providing $\Gamma_*^{\Uparrow}$ as its variable environment. The variables bindings of the original $\Gamma^{\Uparrow}$ are not visible to the UDF.

UDFs are restricted to cell expressions and table expressions with their bodies in SQL. Adding more languages is considered future work.

Provenance Semantics The provenance semantics is analogous to the value semantics.

6.2 Table Expressions

The table expressions e^{table} :: ETABLE are listed in Figure 6.8. Conclusions are denoted

$$\boxed{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e^{table} \Mapsto_{def}^{table} v^{\Uparrow}}$$

where

- $\Gamma^{\Uparrow}$, $\Delta^{\Uparrow}$ and F are analogous to cell expressions (see Section 6.1),
- e^{table} is the expression currently being defined and

$$
\begin{array}{rl}
e^{table} ::= & tab \\
| & \texttt{VALUES}(e^{cell}, \ldots) \\
| & \texttt{WITH}(tab\ \texttt{AS}\ e^{table},\ \ldots\ )\ e^{table} \\
| & e^{table}\ \texttt{UNION ALL}\ e^{table} \\
| & udf(e^{cell}, \ldots) \\
| & \begin{array}{rl}
\texttt{FROM} & (e^{from}, \ldots) \\
\texttt{WHERE} & e^{cell} \\
\texttt{GROUP BY} & (e^{cell}, \ldots) \\
\texttt{AGGREGATES} & (e^{agg}\ \texttt{AS}\ col, \ldots) \\
\texttt{HAVING} & e^{cell} \\
\texttt{WINDOWS} & (e^{win}\ \texttt{OVER}\ e^{over}\ \texttt{AS}\ col, \ldots) \\
\texttt{SELECT} & (e^{cell}\ \texttt{AS}\ col, \ldots) \\
\texttt{ORDER BY} & (e^{cell}\ asc, \ldots) \\
\texttt{DISTINCT ON} & (e^{cell}, \ldots) \\
\texttt{OFFMIT} & \ell\ \ell \\
\multicolumn{2}{c}{e^{table}\ \texttt{UNION ALL}\ e^{table}}
\end{array} \\
| & \texttt{BIND}\ var
\end{array}
$$

Figure 6.8: **SQL table expressions.**

- $v^{\Uparrow} :: \overline{\text{TABLE}\Uparrow}$ is the result table with provenance annotations.

Views are not considered in our SQL dialect. However, a view may be represented as a table–valued UDF with no arguments.

6.2.1 Table References *tab*

$$
\text{DEF-TAB}\quad
\frac{\Delta^{\Uparrow} = \{\!\!\{\ldots, tab \mapsto v^{\Uparrow}, \ldots\}\!\!\}}{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash tab \Mapsto^{table}_{def} \Delta^{\Uparrow}[\![\, tab \,]\!]}
$$

Value Semantics The table reference *tab* performs a lookup in the table environment $\Delta^{\Uparrow}$. If the corresponding entry does not exist, the query fails statically. The entries of $\Delta^{\Uparrow}$ incorporate (lifted) base tables and `WITH` definitions.

Provenance Semantics Provenance annotations and values get propagated in pairs.

6.2.2 VALUES

$$\textsc{def-values}\quad \frac{\Big|\ \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e_i^{cell} \Mapsto_{def}^{cell} v_i^{\Uparrow} \qquad v_i^{\Uparrow} :: \overline{\mathrm{ROW}\Uparrow}\ \Big|_{i=1..n}}{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \texttt{VALUES}(e_1^{cell}, \dots e_n^{cell}) \Mapsto_{def}^{table} \llparenthesis v_1^{\Uparrow}, \dots v_n^{\Uparrow} \rrparenthesis}$$

Value Semantics The $\texttt{VALUES}(e_1^{cell}, \dots)$ expression assembles a table. Each $e_\square^{cell}$ evaluates to a row value. The types of all rows $v_\square^{\Uparrow}$ must match (checked statically, not formalized).

Provenance Semantics Provenance annotations and values get propagated in pairs. The expressions $e_\square^{cell}$ are not necessarily literals. For example, tuple variables can be referenced which may yield non–empty provenance annotations.

6.2.3 WITH

$$\textsc{def-with}\quad \frac{\begin{array}{c}\Big|\ \Gamma^{\Uparrow}; \Delta_{i-1}^{\Uparrow}; F \vdash e_i^{table} \Mapsto_{def}^{table} v_i^{\Uparrow}\ \Big|_{i=1..n+1} \\ \Big|\ \Delta_i^{\Uparrow} \coloneqq \Delta_{i-1}^{\Uparrow} + \{\!\!\{ tab_i \mapsto v_i^{\Uparrow} \}\!\!\}\ \Big|_{i=1..n}\end{array}}{\Gamma^{\Uparrow}; \Delta_0^{\Uparrow}; F \vdash \begin{array}{l}\texttt{WITH} \\ \quad (\ \ tab_1\ \texttt{AS}\ e_1^{table}, \\ \qquad \vdots \\ \qquad tab_n\ \texttt{AS}\ e_n^{table}\ ) \\ e_{n+1}^{table}\end{array} \Mapsto_{def}^{table} v_{n+1}^{\Uparrow}}$$

Value Semantics The WITH expression according to rule DEF-WITH defines n an auxiliary tables tab_i. New entries are added to the table environment $\Delta^{\Uparrow}$ in ascending order. Thus, a table expression e_k^{table} can read from all tables $tab_{k'}$ with $k' < k$. The $e_\square^{table}$ must be topologically ordered. This task is not formalized but considered another simple query rewrite (see the overview in Figure 6.1). Cyclic dependencies among the table definitions raise a static error. Finally, e_{n+1}^{table} evaluates to $v_{n+1}^{\Uparrow}$ which is the result table of the WITH expression.

Provenance Semantics Provenance annotations and values get propagated in pairs.

Future Work Recursion (using the WITH RECURSIVE expression) is not covered in this work. We consider it the very first item of future work (discussed in Chapter 13).

6.2.4 UNION ALL

DEF-UNION-ALL

$$\frac{\begin{array}{c}\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e_a^{cell} \Mapsto_{def}^{cell} \llparenthesis \overset{\Uparrow}{v}_1,\dots \overset{\Uparrow}{v}_n \rrparenthesis \\ \Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e_b^{cell} \Mapsto_{def}^{cell} \llparenthesis \overset{\Uparrow}{v}_{n+1},\dots \overset{\Uparrow}{v}_{n+m} \rrparenthesis\end{array}}{\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e_a^{table}\texttt{UNION ALL}e_b^{table} \Mapsto_{def}^{table} \llparenthesis \overset{\Uparrow}{v}_1,\dots,\overset{\Uparrow}{v}_n,\overset{\Uparrow}{v}_{n+1},\dots \overset{\Uparrow}{v}_{n+m} \rrparenthesis}$$

Value Semantics The UNION ALL expression concatenates two tables. Its definition is provided in DEF-UNION-ALL. The order of rows in the resulting table is pseudo–random (we do however not formalize the randomness).

Provenance Semantics Provenance annotations and values get propagated in pairs.

6.2.5 Table–Valued UDFs

Both value semantics and provenance semantics of table–valued UDFs are analogous to cell–valued UDFs (see Section 6.1.12.1). The essential difference is that the UDF lookup

$$(xs,\ e^{table}) := F[\![\, udf \,]\!]$$

yields a table expression e^{table} instead of a cell expression. The according result value is a table (with provenance annotations).

6.3 SFW Expression

Due to its size and complexity, the SFW expression is challenging to formalize. For this very reason, we break up with the traditional SFW syntax and formalize the expression listed in Figure 6.9. Our SFW expression is designed to form a straight pipeline of data transformation. For example,

$$\texttt{SELECT}\ e_1^{cell}\ \texttt{AS}\ col_1,\ \dots$$

evaluates the expression e_1^{cell} and binds it to the column name col_1. Afterwards, col_1 may be specified as a sort criterion in the ORDER BY–clause. *Logically*, SELECT is evaluated first (creating an intermediate table) and ORDER BY is evaluated afterwards (reading the table emitted by SELECT).

We chose what appears to be the (most) logical dependency of SFW clauses and designed the SFW syntax accordingly. This commitment is not to be mistaken with the physical query evaluation. A query optimizer can enforce any evaluation order as long as the query semantics (which we are going to specify) is preserved.

$$\begin{array}{lrl}
e^{table} ::= \ldots & & \\
\quad | & \texttt{FROM} & (e^{from}, \ldots) \\
& \texttt{WHERE} & e^{cell} \\
& \texttt{GROUP BY} & (e^{cell}, \ldots) \\
& \texttt{AGGREGATES} & (e^{agg}\ \texttt{AS}\ col, \ldots) \\
& \texttt{HAVING} & e^{cell} \\
& \texttt{WINDOWS} & (e^{win}\ \texttt{OVER}\ e^{over}\ \texttt{AS}\ col, \ldots) \\
& \texttt{SELECT} & (e^{cell}\ \texttt{AS}\ col, \ldots) \\
& \texttt{ORDER BY} & (e^{cell}\ asc, \ldots) \\
& \texttt{DISTINCT ON} & (e^{cell}, \ldots) \\
& \texttt{OFFMIT} & \ell\ \ell
\end{array}$$

Figure 6.9: **The SFW expression (repeated from Figure 6.8).**

Related Work G. Bierman, E. Meijer and M. Torgersen propose a SQL–like grammar which also deviates from the common SFW structure. In [BMT07, Sec. 3] they provide a grammar of **Query Expressions** qe, beginning with the `from` keyword.

6.3.0.1 Semantics Overview

A short overview of the SFW semantics is provided. For the purpose of this overview, we assume that all clauses are utilized and have sensible sub–expressions.

- `FROM` $(e^{from}, \ldots)$: Each expression e^{from} reads from an intermediate table. An n–way cross join is carried out.
- `WHERE` e^{cell}: The expression e^{cell} serves as a filter predicate eliminating unqualified rows.
- `GROUP BY` $(e^{cell}, \ldots)$: Individual rows are turned into groups of rows, according to the expressions e^{cell}.
- `AGGREGATES` $(e^{agg}$ `AS` $col, \ldots)$: Each group is aggregated according to the aggregate functions e^{agg}. The result is assigned to an internal column name col.
- `HAVING` e^{cell} The expression e^{cell} serves as a filter predicate eliminating unqualified groups.
- `WINDOWS` $(e^{win}$ `OVER` e^{over} `AS` $col, \ldots)$: The window function e^{win} is computed with frame specification e^{over} and the internal column name col is used to store the result.

- `SELECT` (e^{cell} `AS` $col, \ldots$): This clause evaluates generic expressions e^{cell} and binds them to the final column names *col*. Aggregations and window functions have already been evaluated and their results can be accessed using the internal column names (see above).
- `ORDER BY` (e^{cell} $asc, \ldots$): The intermediate table is sorted according to the criteria e^{cell} with sort directions *asc*.
- `DISTINCT ON` ($e^{cell}, \ldots$): Duplicate rows are removed. The criteria for duplicate comparison are the e^{cell}.
- `OFFMIT` ℓ ℓ: A constant number of preceding and trailing rows is removed. (Name derived from the well known clauses *offset* and *limit*.)

Most of the clauses discussed above are well known from common SQL dialects. The clauses `AGGREGATES` and `WINDOWS` are custom to our backend SQL dialect. The advantage of introducing them is that aggregation expressions e^{agg} and window function expressions e^{win} `OVER` e^{over} can now be restricted to those two clauses. Thus, complexity of the other clauses is reduced.

The backend dialect does not support `SELECT DISTINCT`. Instead, the more flexible `DISTINCT ON` is available. A simple rewrite (copy all result column names from `SELECT` to `DISTINCT ON`) emulates `SELECT DISTINCT` logic.

6.3.1 Decomposition

As the first step in formalizing the SFW semantics, the *monolithic SFW expression* is decomposed into *individual clauses* which can be formalized independently from each other. Figure 6.10 provides an overview of this translation. Both perspectives (monolithic and individual) are supposed to yield the same result table.

- Dashed arrows denote the correspondence between two clauses.
- Solid arrows represent data flow of intermediate tables. Most individual clause consume and produce one of those tables. The intermediate tables are augmented with tuple variables and get formalized next.
- Thin arrows represent lifted tables of type $\overline{\text{TABLE}\Uparrow}$. This is the result type of the SFW evaluation.

Small letters are used to denote individual clauses. For example, `SELECT` becomes `select`.

- `from` is the first clause. Its sub–expressions are supposed to reference base tables (or other table expressions). The decomposed `from` clause is supposed to materialize the cross product for simplicity of the formalization. Also, tuple variables are

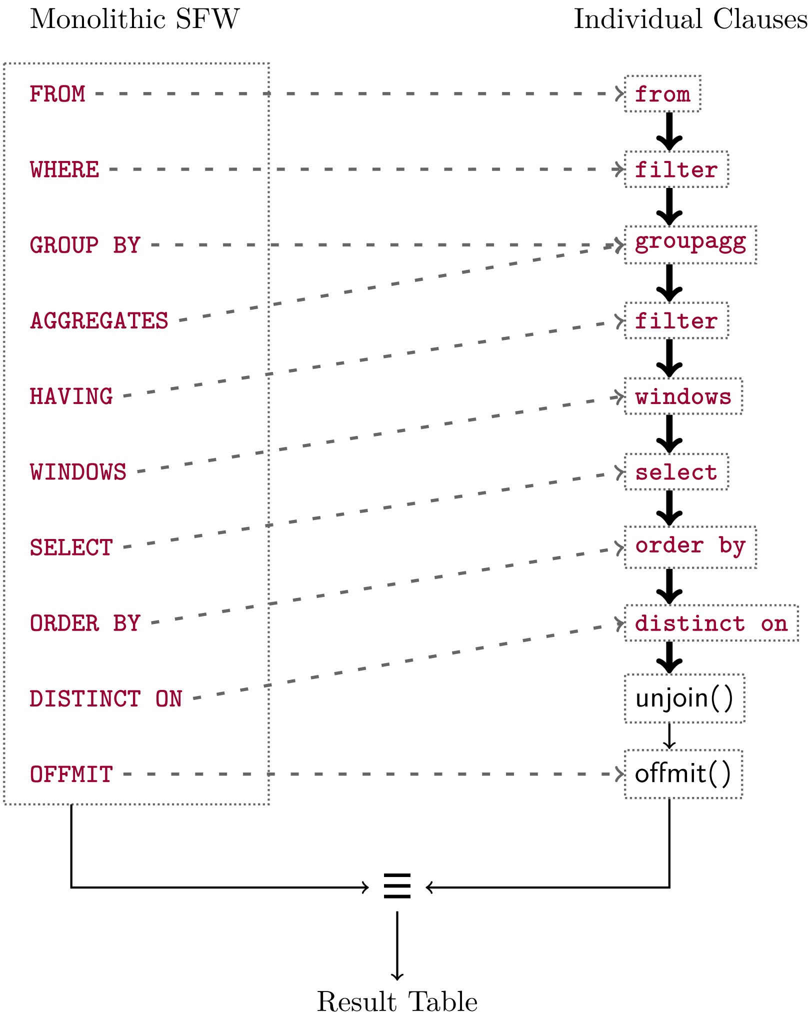


Figure 6.10: **Decomposition of the SFW expression.**

introduced. Query optimization (by the DBMS) is supposed to be carried out in a later stage and has strategies to circumvent computing the full cross product.

- `filter` appears twice. Its first appearance is as a substitute for the `WHERE` clause. It evaluates a predicate expression and drops unqualified tuples.
- `groupagg` carries out grouping *and* aggregation in a single step. As a major advantage, this saves the rest of the SFW query from distinguishing between tuples and groups. Groups are entirely local to `groupagg` and appear nowhere else[1].
- `filter` again filters unqualified tuples. But now, each tuple is an aggregation result (of a former group). Globally, the two `filter` clauses are different. Locally, they share the very same semantics. Our formalization exploits this redundancy.
- `order by` sorts the table. The succeeding clauses must retain the order of tuples (not formalized).
- $\mathsf{unjoin}()$ throws away all tuple variables and turns the intermediate data structure into type $\overline{\textsc{table}\Uparrow}$.

6.3.2 Joined Tables

Joined tables are the main data structure used in the SFW formalization. They are nothing else than the cartesian product of the input tables, qualified with tuple variables. The type of a joined table is denoted $\textsc{jtable}\Uparrow(\mathit{var} \mapsto \textsc{row}\Uparrow(\cdot), \ldots)$ where var is the tuple variable specified in the `from` clause and $\textsc{row}\Uparrow(\cdot)$ the row type of the corresponding input table.

For example, let $\mathsf{planets}\Uparrow$ and $\mathsf{stars}\Uparrow$ of Figure 6.11(a) be two input tables. The `from` clause in Figure 6.11(b) (the `LATERAL` keyword can be ignored for now) then generates the joined table listed in Figure 6.11(c). The value constructor for joined tables is denoted $J(\cdot)$. The type of the example table is

$$\begin{aligned}\textsc{jtable}(\lBrace p &\mapsto \textsc{row}(\mathsf{name} \mapsto \textsc{text}, \mathsf{density} \mapsto \textsc{text}),\\ s &\mapsto \textsc{row}(\mathsf{name} \mapsto \textsc{text})\rBrace)\end{aligned}$$

.

Single rows of a joined table can be added to the variable environment $\Gamma^{\Uparrow}$. Using an example row, this is denoted

$$\Gamma^{\Uparrow} \Cup \lBrace p \mapsto \langle \blacktriangleleft\mathsf{Earth}, \mathbb{p}_1\blacktriangleright, \blacktriangleleft\mathsf{rocky}, \mathbb{p}_5\blacktriangleright\rangle, s \mapsto \langle \blacktriangleleft\mathsf{Sol}, \mathbb{p}_{10}\blacktriangleright\rangle\rBrace$$

[1] As a small exception, window functions can define (group–like) partitions.

planets⇧

name	*density*
$\blacktriangleleft\textsf{Earth}, \mathbb{p}_1\blacktriangleright$	$\blacktriangleleft\textsf{rocky}, \mathbb{p}_5\blacktriangleright$
$\blacktriangleleft\textsf{Mars}, \mathbb{p}_2\blacktriangleright$	$\blacktriangleleft\textsf{rocky}, \mathbb{p}_6\blacktriangleright$
$\blacktriangleleft\textsf{Jupiter}, \mathbb{p}_3\blacktriangleright$	$\blacktriangleleft\textsf{gaseous}, \mathbb{p}_7\blacktriangleright$
$\blacktriangleleft\textsf{Neptune}, \mathbb{p}_4\blacktriangleright$	$\blacktriangleleft\textsf{icy}, \mathbb{p}_8\blacktriangleright$

stars⇧

name
$\blacktriangleleft\textsf{Sol}, \mathbb{p}_{10}\blacktriangleright$

(a) **Input tables.**

```
from planets LATERAL () AS p,
     stars   LATERAL () AS s
```

(b) `from` **clause.**

$$
\begin{aligned}
J(&\{\!\!\{p \mapsto \langle\blacktriangleleft\textsf{Earth}, \mathbb{p}_1\blacktriangleright, \blacktriangleleft\textsf{rocky}, \mathbb{p}_5\blacktriangleright\rangle, s \mapsto \langle\blacktriangleleft\textsf{Sol}, \mathbb{p}_{10}\blacktriangleright\rangle\}\!\!\},\\
&\{\!\!\{p \mapsto \langle\blacktriangleleft\textsf{Mars}, \mathbb{p}_2\blacktriangleright, \blacktriangleleft\textsf{rocky}, \mathbb{p}_6\blacktriangleright\rangle, s \mapsto \langle\blacktriangleleft\textsf{Sol}, \mathbb{p}_{10}\blacktriangleright\rangle\}\!\!\},\\
&\{\!\!\{p \mapsto \langle\blacktriangleleft\textsf{Jupiter}, \mathbb{p}_3\blacktriangleright, \blacktriangleleft\textsf{gaseous}, \mathbb{p}_7\blacktriangleright\rangle, s \mapsto \langle\blacktriangleleft\textsf{Sol}, \mathbb{p}_{10}\blacktriangleright\rangle\}\!\!\},\\
&\{\!\!\{p \mapsto \langle\blacktriangleleft\textsf{Neptune}, \mathbb{p}_4\blacktriangleright, \blacktriangleleft\textsf{icy}, \mathbb{p}_8\blacktriangleright\rangle, s \mapsto \langle\blacktriangleleft\textsf{Sol}, \mathbb{p}_{10}\blacktriangleright\rangle\}\!\!\}\\
&)
\end{aligned}
$$

(c) **The corresponding joined table.**

Figure 6.11: **Example: a joined table being constructed.**

$$\begin{aligned} e^{clause} ::=\ & \texttt{from}\ (e^{from}, \dots) \\ \mid\ & \texttt{filter}\ e^{cell} \\ \mid\ & \texttt{groupagg}\ (e^{cell}, \dots)(e^{cell}\ \texttt{AS}\ col, \dots) \\ \mid\ & \texttt{windows}\ (e^{win}\ \texttt{OVER}\ e^{over}\ \texttt{AS}\ col, \dots) \\ \mid\ & \texttt{select}\ (e^{cell}\ \texttt{AS}\ col, \dots) \\ \mid\ & \texttt{order by}\ (e^{cell}\ asc, \dots) \\ \mid\ & \texttt{distinct on}\ (e^{cell}, \dots) \end{aligned}$$

Figure 6.12: e^{clause} **expressions.**

. The operation $\Cup$ merges two dictionaries and favors entries of the right–handed dictionary in case of conflicting keys. We make heavy use of this operation to formalize the loop semantics of the individual SFW clauses. The asymmetry of $\Cup$ implements variable shadowing.

6.3.3 Formalized Decomposition

Through decomposition, the SFW expressions breaks down into the individual clauses of Figure 6.12. These expressions and the two auxiliary functions unjoin(·) and offmit(·) get formalized afterwards. First, we formalize the decomposition step.

Value Semantics Rule DEF-SFW of Figure 6.13 formalizes the SFW decomposition step. It has basically the same semantics as the informal version in Figure 6.10, i.e. a straight pipeline of data transformation steps. The joined tables $J^{\Uparrow}_{\square}$ are intermediate results of individual SFW clauses. The pipeline starts with $J(\{\!|\,|\!\})$ (=a joined table consisting of a single, empty row) which is fed into the `from` expression (more explanation regarding to `from` will be carried out below). After evaluation of `from` and assignment of $J^{\Uparrow}_0$, the very same $J^{\Uparrow}_0$ is fed into the evaluation of `filter`. This pattern repeats until `distinct on` is evaluated. The semi–final $J^{\Uparrow}_7$ is fed into two auxiliary functions which assemble $v^{\Uparrow}$, the final result of the SFW expression.

Provenance Semantics Provenance annotations and values get propagated in pairs.

6.3.4 e^{clause} Expressions

Conclusions are denoted

$$\boxed{J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e^{clause} \Mapsto^{sfw}_{def} J^{\Uparrow}_{res}}$$

DEF-SFW

$$
\frac{
\begin{array}{l}
J(\{\!\!\{\}\!\!\});\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \texttt{from}\ (e_1^{from}, \ldots e_n^{from}) \Mapsto_{def}^{sfw} J_0^{\Uparrow} \\
\quad J_0^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \texttt{filter}\ e_{whe}^{cell} \Mapsto_{def}^{sfw} J_1^{\Uparrow} \\
\quad J_1^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \texttt{groupagg}\ (e_{gro_1}^{cell}, \ldots e_{gro_m}^{cell})\ (e_1^{agg}\ \texttt{AS}\ col_{agg_1}, \ldots e_p^{agg}\ \texttt{AS}\ col_{agg_p}) \Mapsto_{def}^{sfw} J_2^{\Uparrow} \\
\quad J_2^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \texttt{filter}\ e_{hav}^{cell} \Mapsto_{def}^{sfw} J_3^{\Uparrow} \\
\quad J_3^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \begin{array}{l} \texttt{windows}\ (e_1^{win}\ \texttt{OVER}\ e_1^{over}\ \texttt{AS}\ col_{win_1}, \ldots \\ \qquad\quad e_q^{win}\ \texttt{OVER}\ e_q^{over}\ \texttt{AS}\ col_{win_q}) \end{array} \Mapsto_{def}^{sfw} J_4^{\Uparrow} \\
\quad J_4^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \texttt{select}\ (e_{sel_1}^{cell}\ \texttt{AS}\ col_{sel_1}, \ldots e_{sel_r}^{cell}\ \texttt{AS}\ col_{sel_r}) \Mapsto_{def}^{sfw} J_5^{\Uparrow} \\
\quad J_5^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \texttt{order by}\ (e_{ord_1}^{cell}\ asc_1, \ldots e_{ord_u}^{cell}\ asc_u) \Mapsto_{def}^{sfw} J_6^{\Uparrow} \\
\quad J_6^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \texttt{distinct on}\ (e_{dis_1}^{cell}, \ldots e_{dis_v}^{cell}) \Mapsto_{def}^{sfw} J_7^{\Uparrow} \\
\qquad\qquad\qquad\qquad v^{\Uparrow} \ \coloneqq\ \mathsf{offmit}(\mathsf{unjoin}(J_7^{\Uparrow}), \ell_{offset}, \ell_{limit})
\end{array}
}{
\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash
\begin{array}{rl}
\texttt{FROM} & (e_1^{from}, \ldots e_n^{from}) \\
\texttt{WHERE} & e_{whe}^{cell} \\
\texttt{GROUP BY} & (e_{gro_1}^{cell}, \ldots e_{gro_m}^{cell}) \\
\texttt{AGGREGATES} & (e_1^{agg}\ \texttt{AS}\ col_{agg_1}, \ldots e_p^{agg}\ \texttt{AS}\ col_{agg_p}) \\
\texttt{HAVING} & e_{hav}^{cell} \\
\texttt{WINDOWS} & (e_1^{win}\ \texttt{OVER}\ e_1^{over}\ \texttt{AS}\ col_{win_1}, \ldots \\
& \ e_q^{win}\ \texttt{OVER}\ e_q^{over}\ \texttt{AS}\ col_{win_q}) \\
\texttt{SELECT} & (e_{sel_1}^{cell}\ \texttt{AS}\ col_{sel_1}, \ldots e_{sel_r}^{cell}\ \texttt{AS}\ col_{sel_r}) \\
\texttt{ORDER BY} & (e_{ord_1}^{cell}\ asc_1, \ldots e_{ord_u}^{cell}\ asc_u) \\
\texttt{DISTINCT ON} & (e_{dis_1}^{cell}, \ldots e_{dis_v}^{cell}) \\
\texttt{OFFMIT} & \ell_{off}\ \ell_{lim}
\end{array}
\Mapsto_{def}^{table} v^{\Uparrow}
}
$$

Figure 6.13: **The SFW semantics.**

where

- $J^{\Uparrow}$ :: JTABLE$\Uparrow(\cdot)$ is the joined table (handed over from the previous SFW clause),
- $\Gamma^{\Uparrow}$, $\Delta^{\Uparrow}$ and F are analogous to cell expressions (see Section 6.1),
- e^{clause} is the expression currently being defined and
- $J^{\Uparrow}_{res}$ :: JTABLE$\Uparrow(\cdot)$ is the resulting joined table.

The two joined tables $J^{\Uparrow}$ and $J^{\Uparrow}_{res}$ do not share the same type. For example, a `select` clause specifies new columns.

6.3.4.1 `from`

The `from` $(e_1^{from}, \dots e_n^{from})$ expression undergoes an additional expansion step (for presentation reasons). The according production rule is listed directly below.

$$e^{from} ::= e^{table}\ \texttt{LATERAL}\ (var_{lat}, \dots)\ \texttt{AS}\ var_{bind}$$

On the right–hand side, e^{table} specifies the table expression to be evaluated and var_{bind} the name of a non–optional tuple variable. Additionally, `LATERAL` semantics is supported. If variable names var_{lat} (bound in preceding `from` entries) are provided, these variables become visible for e^{table}. It is a design decision to enforce explicit `LATERAL` variables (i.e., non–standard SQL). In our opinion, this improves the readability of queries.

Value Semantics Rule DEF-FROM-REC uses recursion to implement m–way cross joins. In each recursion step, a single join (between the existing $J^{\Uparrow}$ and the current e^{table}) is carried out. (Plain cross joins easily exceed reasonable query runtimes. As a reminder, we specify the semantics of `from` and not its execution plan.) After all joins have been processed, rule DEF-FROM-STOP yields the final $J^{\Uparrow}$. If $m = 0$, the single and empty row $J(\{\!|\,|\!\})$ (see Figure 6.13) becomes the result.

Taking a closer look at DEF-FROM-REC, $n := |J^{\Uparrow}|$ denotes the cardinality of the current joined table. The expression e_1^{table} is evaluated n times, because sensible `LATERAL` is assumed in general. We construct a new environment $\Gamma^{\Uparrow} \uplus J^{\Uparrow}[\![\,i\,]\!]$ for each evaluation. Thus, the n different valuations of `LATERAL`–specific tuple variables are accessible to the evaluation of e_1^{table}. The resulting $v_i^{\Uparrow}$ are then used to implement the cross join, assembling $J^{\Uparrow}_{+}$. If `LATERAL` variables are not specified, the $v_i^{\Uparrow}$ are all the same and the ordinary cross join is carried out.

Provenance Semantics Provenance annotations and values get propagated in pairs.

DEF-FROM-REC

$$\dfrac{\begin{array}{c} n := |J^{\Uparrow}| \\ \Big|\ \Gamma^{\Uparrow} \uplus J\llbracket i \rrbracket^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e_1^{table} \Mapsto_{def}^{table} v_i^{\Uparrow}\ \Big|_{i=1..n} \\ J_+^{\Uparrow} := J\big(\Big|\ J^{\Uparrow}\llbracket i \rrbracket \uplus \{\!|\, var_1 \mapsto v_i^{\Uparrow}\llbracket k \rrbracket \,|\!\},\ \Big|_{i=1..n,k=1..|v_i^{\Uparrow}|}\big) \\ J_+^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \begin{array}{l} \texttt{from}\ (e_2^{table}\ \texttt{LATERAL}\ vs_1\ \texttt{AS}\ var_2, \\ \quad\vdots \\ \quad e_m^{table}\ \texttt{LATERAL}\ vs_m\ \texttt{AS}\ var_m \\ \quad) \end{array} \Mapsto_{def}^{sfw} J_{res}^{\Uparrow} \end{array}}{J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \begin{array}{l} \texttt{from}\ (e_1^{table}\ \texttt{LATERAL}\ vs_1\ \texttt{AS}\ var_1, \\ \quad\vdots \\ \quad e_m^{table}\ \texttt{LATERAL}\ vs_m\ \texttt{AS}\ var_m \\ \quad) \end{array} \Mapsto_{def}^{sfw} J_{res}^{\Uparrow}}$$

DEF-FROM-STOP

$$J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \texttt{from}\ () \Mapsto_{def}^{sfw} J^{\Uparrow}$$

Figure 6.14: **Semantics of `from`.**

6.3.4.2 `filter`

The `WHERE`– and `HAVING`–clauses are both mapped to instances of `filter` (see Figure 6.10). The `filter` semantics is formalized in rule DEF-FILTER and the auxiliary function of Figure 6.15.

Value Semantics For each input row (=each variable environment $\Gamma^{\Uparrow} \uplus J^{\Uparrow}\llbracket i \rrbracket$), the predicate e^{cell} is evaluated. The resulting $v_i^{\Uparrow}$ is always an annotated boolean. The auxiliary function $\mathsf{appendif}(\cdot)$ adds any row $v_{cur}^{\Uparrow}$ which satisfies the predicate to the result.

Provenance Semantics The interesting part is the true–case of $\mathsf{appendif}(.)$, i.e.

$$\Psi(v_{cur}^{\Uparrow}, \Upsilon(\mathbb{p}_{pred}))$$

. The predicate e^{cell} decides about the existence of $v_{cur}^{\Uparrow}$ in the result table. We consider this dependency as Why–provenance (as described in Section 5.2.3). The elements of $\mathbb{p}_{pred}$ are first turned into Why–provenance and the resulting annotation is recursively added to the entire row.

$$\textsc{def-filter}\quad \dfrac{\begin{array}{c} n \coloneqq |J^{\Uparrow}| \\ \left| \Gamma^{\Uparrow} \mathbin{ɯ} J^{\Uparrow}[\![\,i\,]\!]; \Delta^{\Uparrow}; F^{\Uparrow} \vdash e^{cell} \Mapsto^{cell}_{def} v_i^{\Uparrow} \qquad v_i^{\Uparrow} :: \blacktriangleleft\textsc{Bool}, \textsc{PSet}\blacktriangleright \right|_{i=1..n} \\ J_0^{\Uparrow} \coloneqq J\,() \qquad \left| J_i^{\Uparrow} \coloneqq \mathsf{appendif}(J_{i-1}^{\Uparrow},\ v_i^{\Uparrow},\ J^{\Uparrow}[\![\,i\,]\!]) \right|_{i=1..n} \end{array}}{J^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \texttt{filter}\ e^{cell} \Mapsto^{sfw}_{def} J_n^{\Uparrow}}$$

(a) **Main rule.**

$$\mathsf{appendif}(J\,(v_1^{\Uparrow},\dots v_m^{\Uparrow}), \blacktriangleleft v_{pred}, \mathbb{p}_{pred}\blacktriangleright, v_{cur}^{\Uparrow}) \coloneqq$$

$$\begin{cases} J\,(v_1^{\Uparrow},\dots v_m^{\Uparrow}, \Psi(v_{cur}^{\Uparrow}, \Upsilon(\mathbb{p}_{pred}))) & , v_{pred} = \mathsf{true} \\ J\,(v_1^{\Uparrow},\dots v_m^{\Uparrow}) & , v_{pred} = \mathsf{false} \\ J\,(v_1^{\Uparrow},\dots v_m^{\Uparrow}) & , v_{pred} = \mathsf{null} \end{cases}$$

(b) **Auxiliary function.**

Figure 6.15: **Semantics of `filter`.**

If $v_{cur}^{\Uparrow}$ does not satisfy the predicate, the provenance annotations are dropped together with the values.

6.3.4.3 `select`

$$\textsc{def-select}\quad \dfrac{\begin{array}{c} n \coloneqq |J^{\Uparrow}| \qquad \left| \Gamma^{\Uparrow} \mathbin{ɯ} J^{\Uparrow}[\![\,i\,]\!]; \Delta^{\Uparrow}; F \vdash e_k^{cell} \Mapsto^{cell}_{def} v_{i,k}^{\Uparrow} \right|_{i=1..n,k=1..m} \\ J_{res}^{\Uparrow} \coloneqq (\left| J^{\Uparrow}[\![\,i\,]\!] \mathbin{ɯ} \{\!\{\texttt{sel} \mapsto \langle v_{i,1}^{\Uparrow},\dots v_{i,m}^{\Uparrow}\rangle\}\!\}, \right|_{i=1..n}) \end{array}}{J^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \texttt{select}\ (e_1^{cell}\ \texttt{AS}\ col_1, \dots e_m^{cell}\ \texttt{AS}\ col_m) \Mapsto^{sfw}_{def} J_{res}^{\Uparrow}}$$

Rule DEF-SELECT formalizes the `select` clause. Due to substitution of aggregates and window functions (these are processed in their dedicated clauses), the formalization of the `select` clause is focused on simple cell expressions.

Value Semantics The `select` expression evaluates $n \cdot m$ cells for the $n \coloneqq |J^{\Uparrow}|$ input rows and m columns. Each input row $J^{\Uparrow}[\![\,i\,]\!]$ yields an output row

$$J^{\Uparrow}[\![\,i\,]\!] \mathbin{ɯ} \{\!\{\texttt{sel} \mapsto \langle v_{i,1}^{\Uparrow}, \dots\rangle\}\!\}$$

with the new rows (column names implicit) bound to an artificial tuple variable `sel`. Succeeding SFW clauses can reference `sel` and its columns. For example, subexpressions in the `order by` clause can reference such columns and employ them as sort criteria. In a later stage of the SFW evaluation (namely, $\mathsf{unjoin}(\cdot)$) the contents of `sel` get unwrapped and constitute the sole result of the SFW expression.

Provenance Semantics Provenance annotations and values get propagated in pairs.

6.3.4.4 `order by`

$$\textsc{def-order-by}\quad\frac{\begin{array}{c} J_s^{\Uparrow} \mathrel{:=} \mathsf{sort}^{\Uparrow}(J^{\Uparrow}, \Gamma^{\Uparrow}, \Delta^{\Uparrow}, F, (e_1^{cell}\ \ asc_1, \ldots e_m^{cell}\ \ asc_m)) \\ n \mathrel{:=} |J_s^{\Uparrow}| \\ \Big|\ \Gamma^{\Uparrow} \uplus J_s^{\Uparrow}[\![\, i \,]\!]; \Delta^{\Uparrow}; F \vdash e_k^{cell} \Mapsto_{def}^{cell} v_{i,k}^{\Uparrow} \qquad (v_{i,k}, \mathbb{p}_{i,k}) \mathrel{:=} \curlyvee(v_{i,k}^{\Uparrow})\ \Big|_{i=1..n,k=1..m} \\ \Bigg|\ \mathbb{p}_i \mathrel{:=} \mathsf{Y}(\bigcup_{k=1}^{m} \mathbb{p}_{i,k})\ \Bigg|_{i=1..n} \\ J_{res}^{\Uparrow} \mathrel{:=} J\{\!|\Psi(J_s^{\Uparrow}[\![\, 0 \,]\!], \mathbb{p}_0), \ldots \Psi(J_s^{\Uparrow}[\![\, n \,]\!], \mathbb{p}_n)|\!\} \end{array}}{J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \texttt{order by}\ (e_1^{cell}\ \ asc_1, \ldots e_m^{cell}\ \ asc_m) \Mapsto_{def}^{sfw} J_{res}^{\Uparrow}}$$

Value Semantics Rule DEF-ORDER-BY formalizes the `order by` expression. We employ the auxiliary function $\mathsf{sort}^{\Uparrow}(\cdot)$ which carries out the actual sorting and yields the sorted (joined) table $J_s^{\Uparrow}$. The function $\mathsf{sort}^{\Uparrow}(\cdot)$ ignores the provenance annotations (does not consider them as sort criteria) but keeps the annotations attached to their values.

Provenance Semantics We consider the sort criteria as predicates (in the broader sense). These criteria decide over the position of a row in the result table. Therefore, they are modeled as Why–provenance.

The first part of rule DEF-ORDER-BY produces the sorted table $J_s^{\Uparrow}$. The sort criteria e_k^{cell} get evaluated in order to assemble a single provenance annotation $\mathbb{p}_i$ for each row. Then, this Why–provenance is recursively added to its associated row.

Outlook A formalization of the intricate $\mathsf{sort}^{\Uparrow}(\cdot)$ is not carried out. The main feature of our approach to provenance analysis (subject of Chapter 8) is able to separate provenance annotations from values. Thus, mixed workloads (values and provenance) do not occur and an implementation of $\mathsf{sort}^{\Uparrow}(\cdot)$ is not needed.

DEF-DISTINCT-ON-VOID

$$J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \texttt{distinct on ()} \Mapsto^{sfw}_{def} J^{\Uparrow}$$

DEF-DISTINCT-ON

$$\dfrac{\begin{array}{c} m \geq 1 \qquad J^{\Uparrow}_{d} \coloneqq \mathsf{distincton}^{\Uparrow}(J^{\Uparrow}, \Gamma^{\Uparrow}, \Delta^{\Uparrow}, F, (e^{cell}_{1}, \ldots e^{cell}_{m})) \\ n \coloneqq |J^{\Uparrow}_{d}| \\ \Big| \; \Gamma^{\Uparrow} \uplus J^{\Uparrow}_{d}[\![i]\!]; \Delta^{\Uparrow}; F \vdash e^{cell}_{k} \Mapsto^{cell}_{def} v^{\Uparrow}_{i,k} \qquad (v_{i,k}, \mathbb{p}_{i,k}) \coloneqq \curlyvee(v^{\Uparrow}_{i,k}) \; \Big|_{i=1..n, k=1..m} \\ \Big| \; \mathbb{p}_{i} \coloneqq \mathsf{Y}(\bigcup_{k=1}^{m} \mathbb{p}_{i,k}) \; \Big|_{i=1..n} \\ J^{\Uparrow}_{res} \coloneqq J\{\!|\Psi(J^{\Uparrow}_{d}[\![0]\!], \mathbb{p}_{0}), \ldots \Psi(J^{\Uparrow}_{d}[\![n]\!], \mathbb{p}_{n})|\!\} \end{array}}{J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \texttt{distinct on } (e^{cell}_{1}, \ldots e^{cell}_{m}) \Mapsto^{sfw}_{def} J^{\Uparrow}_{res}}$$

Figure 6.16: **Semantics of `distinct on`.**

6.3.4.5 `distinct on`

The formalization of the `distinct on` clause is very similar to `order by`. The rules are provided in Figure 6.16.

Value Semantics Rule DISTINCT-ON-DEF-EMPTY deals with the trivial case when no criteria are provided (all rows are kept). The other cases are handled by rule DEF-DISTINCT-ON. The structure of this rule (and its semantics) is very similar to DEF-ORDER-BY. The main difference is that function $\mathsf{distincton}^{\Uparrow}(\cdot)$ removes duplicates. The function does not inspect provenance annotations for the purpose of duplicate elimination. However, all provenance annotations (of not eliminated rows) stick to their associated values. The cardinalities are $|J^{\Uparrow}_{d}| \leq |J^{\Uparrow}|$.

Provenance Semantics In the remaining formalization, the criteria $e^{cell}_{1}, \ldots e^{cell}_{m}$ are analyzed for their data provenance $\mathbb{p}_{i}$ (specific for every i–th row), Why–provenance is generated and then added to the i–th row. The data provenance of eliminated rows is dropped.

6.3.4.6 unjoin(.)

The unjoin(.) auxiliary function has the type

$$\mathsf{unjoin(.)} :: \text{JTABLE}\Uparrow(\ldots, \texttt{sel} \mapsto \tau, \ldots) \rightarrowtail \text{TABLE}\Uparrow(\tau)$$

and definition

$$\mathsf{unjoin}(J(\overset{\Uparrow}{v}_1, \ldots \overset{\Uparrow}{v}_n)) \coloneqq (\overset{\Uparrow}{v}_1[\![\,\texttt{sel}\,]\!], \ldots \overset{\Uparrow}{v}_n[\![\,\texttt{sel}\,]\!])$$

. Essentially, it dereferences `sel` (which contains the final columns evaluated in the **`select`** clause). At this point in the pipeline (see Figure 6.10 for the overview), all intermediate results (like, variables bound in the **`from`** clause) can be thrown away. The final evaluation step is carried out through $\mathsf{offmit}(\cdot)$ (see below) which removes rows according to literal expressions.

6.3.4.7 offmit(·)

$\mathsf{offmit}(\cdot)$ takes care of removing leading (`OFFSET` semantics) and trailing (`LIMIT` semantics) rows. It has the type

$$\mathsf{offmit}(\cdot) :: (\text{TABLE}\Uparrow(\tau), \text{INT}, \text{INT}) \rightarrowtail \text{TABLE}\Uparrow(\tau)$$

and definition

$$\begin{aligned}
&\mathsf{offmit}((\overset{\Uparrow}{v}_1, \ldots \overset{\Uparrow}{v}_n), o, l) \coloneqq \\
&\qquad \begin{cases}
(\overset{\Uparrow}{v}_1, \ldots \overset{\Uparrow}{v}_n) & , (o = \mathsf{null}) \land (l = \mathsf{null}) \\
(\overset{\Uparrow}{v}_{o+1}, \ldots \overset{\Uparrow}{v}_n) & , (o \neq \mathsf{null}) \land (l = \mathsf{null}) \\
(\overset{\Uparrow}{v}_1, \ldots \overset{\Uparrow}{v}_l) & , (o = \mathsf{null}) \land (l \neq \mathsf{null}) \\
\mathsf{offmit}(\mathsf{offmit}((\overset{\Uparrow}{v}_1, \ldots \overset{\Uparrow}{v}_n), o, \mathsf{null}), \mathsf{null}, l) & , (o \neq \mathsf{null}) \land (l \neq \mathsf{null})
\end{cases}
\end{aligned}$$

. The expressions o and l are integer literals. We employ null values to denote `OFFSET 0` and `LIMIT` ∞ (=all rows are to be kept).

6.4 Aggregations and Window Functions

6.4.1 Aggregations in the Backend Dialect

Before we carry out the formalization of aggregations, we go one step back and discuss aggregations from the perspectives of the frontend and backend SQL dialects.

Figure 6.17(a) shows an example query in common SQL. The hidden complexity in this example is located in `SELECT p.density` which is not as simple as it seems. `p.density`

```
   SELECT p.density AS density,
          COUNT(*) AS count
     FROM planets AS p
 GROUP BY p.density
```

(a) **Frontend dialect.**

```
 1         FROM planets AS p
 2        WHERE TRUE
 3     GROUP BY p.density
 4   AGGREGATES THE(p.density) AS a1,
 5              COUNTSTAR() AS a2
 6       HAVING TRUE
 7      WINDOWS ()
 8       SELECT aggs.a1 AS density,
 9              aggs.a2 AS count
10     ORDER BY ()
11  DISTINCT ON ()
12       OFFMIT NULL NULL
```

(b) **Backend dialect.**

Figure 6.17: **Example: rewrite of aggregations.**

is not an ordinary column reference but reaches into the current group of rows (let the group size be n rows) and yields *just one* of the n identical values of `p.density`. This semantics is different from non–aggregating context where `p.density` would be evaluated per input row. In other words, the semantics of an expression depends on the context. This observation is not new but has already been formulated by S. Peyton Jones and P. Wadler in [PW07][Sec. 3.4]. They have introduced the aggregate function the(·) in order to make this contextual knowledge explicit.

We adopt their approach for SQL and when our backend dialect is produced, we turn `p.density` into `THE(p.density)` (see Figure 6.17(b) on line 4). This rewrite step allows us to process `THE` and ordinary aggregation functions (like `COUNTSTAR`) in the same way.

On top of that, the backend dialect introduces a new SFW clause called `AGGREGATES`. In that clause, the collected aggregations from the entires SFW block are collected. Results of aggregations get bound to artificial columns and the tuple variable aggs. The `SELECT` clause of Figure 6.17(b) is totally unaware of aggregations. This substitution is carried out for `HAVING`, `ORDER BY` and `DISTINCT ON` as well and greatly reduces their complexity. In an analogous way, this substitution is carried out for window functions, collecting them in the `WINDOWS` clause.

6.4.2 `groupagg`

The `groupagg` expression is a merge of the `GROUP BY` and `AGGREGATES` clause. Merging them together is of advantage because groups (=an additional data structure) can be kept local to `groupagg`. Figure 6.18 provides an overview of the types used inside of `groupagg`. The

$$\begin{tikzcd}
\textsc{jtable}\Uparrow(\tau) \arrow[d, "\texttt{GROUP BY}"] \\
\{\!\!\{\textsc{int} \mapsto \textsc{jtable}\Uparrow(\tau)\}\!\!\} \arrow[d, "\texttt{AGGREGATES}"] \\
\textsc{jtable}\Uparrow(\tau')
\end{tikzcd}$$

(a) **Types.**

$$\begin{tikzcd}
\textsc{jtable}\Uparrow(\{\!\!\{var_1 \mapsto \tau_1, \ldots\}\!\!\}) \arrow[d, "\texttt{GROUP BY}"] \\
\begin{array}{l}\{\!\!\{1 \mapsto \textsc{jtable}\Uparrow(\{\!\!\{var_1 \mapsto \tau_1, \ldots\}\!\!\}), \\ \;\; 2 \mapsto \textsc{jtable}\Uparrow(\{\!\!\{var_1 \mapsto \tau_1, \ldots\}\!\!\}), \\ \;\; \ldots\}\!\!\}\end{array} \arrow[d, "\texttt{AGGREGATES}"] \\
\textsc{jtable}\Uparrow(\{\!\!\{\mathsf{aggs} \mapsto \overline{\textsc{row}\Uparrow}\}\!\!\})
\end{tikzcd}$$

(b) **Variable names.**

Figure 6.18: **Overview: data structures used in the `groupagg` formalization.**

input is a joined table, emitted by the previous clause (i.e., `filter`). First, grouping is carried out according to the `GROUP BY` clause. The result is a dictionary where each group corresponds to a value in the dictionary. The row type τ does not change in this process. Second, each group is aggregated into a single row. We introduce the tuple variable aggs to qualify those rows. After aggregation, aggs is the only variable left. For example, all `FROM` bindings (except for the correlated tuple variables still sitting in $\Gamma^{\Uparrow}$) are gone. Rule DEF-GROUPAGG is presented in Figure 6.19. We discuss the provenance semantics first.

Provenance Semantics Similar to `order by` and `distinct on`, Why–provenance is generated. The grouping criteria $(e_1^{cell}, \ldots e_r^{cell})$ determine the group of a row. Rule DEF-GROUPAGG evaluates the criteria (for each of the n rows) and collects the Why–provenance in $\mathbb{p}_i$. Then, the Why–provenance is added to the associated row. The intermediate result is $J_y^{\Uparrow}$. In the remaining rule, the provenance annotations just get forwarded.

Value Semantics Regarding to `GROUP BY`, we make use of the helper function $\mathsf{groupby}^{\Uparrow}(\cdot)$. The expressions $e_\square^{cell}$ get evaluated for each row and according groups are formed, constituting the dictionary $G^{\Uparrow}$. The contents of $G^{\Uparrow}$ are exemplified in Figure 6.18, i.e. keys are unique (ascending integers) and the dictionarie's values are the groups. If the `GROUP BY` clause is empty, a single group is created.

DEF-GROUPAGG

$$\begin{array}{c}
n \coloneqq |J^{\Uparrow}| \\
\left| \; \Gamma^{\Uparrow} \uplus J^{\Uparrow}\llbracket\, i \,\rrbracket; \Delta^{\Uparrow}; F \vdash e_k^{cell} \Rrightarrow_{def}^{cell} v_{i,k}^{\Uparrow} \qquad (v_{i,k}, \mathbb{p}_{i,k}) \coloneqq \curlyvee(v_{i,k}^{\Uparrow}) \; \right|_{i=1..n,k=1..r} \\
\left| \; \mathbb{p}_i \coloneqq \mathsf{Y}(\bigcup_{k=1}^{m} \mathbb{p}_{i,k}) \; \right|_{i=1..n} \\
J_y^{\Uparrow} \coloneqq J\{\Psi(J^{\Uparrow}\llbracket\, 0 \,\rrbracket, \mathbb{p}_0), \ldots \Psi(J^{\Uparrow}\llbracket\, n \,\rrbracket, \mathbb{p}_n)\} \\
G^{\Uparrow} \coloneqq \mathsf{groupby}^{\Uparrow}(J_y^{\Uparrow}, \Gamma^{\Uparrow}, \Delta^{\Uparrow}, F, (e_1^{cell}, \ldots e_r^{cell})) \\
g \coloneqq |G^{\Uparrow}| \\
\left| \; G^{\Uparrow}\llbracket\, i \,\rrbracket; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e_k^{agg} \Rrightarrow_{def}^{agg} v_{agg_{i,k}}^{\Uparrow} \; \right|_{i=1..g,k=1..m} \\
\left| \; v_i^{\Uparrow} \coloneqq \{\!\!\{ \mathrm{aggs} \mapsto \langle col_1 \mapsto v_{agg_{i,1}}^{\Uparrow}, \ldots col_m \mapsto v_{agg_{i,m}}^{\Uparrow} \rangle \}\!\!\} \; \right|_{i=1..g} \\
\hline
J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash
\begin{array}{l}
\texttt{groupagg}\ (e_1^{cell}, \ldots e_r^{cell}) \\
(e_1^{agg}\ \texttt{AS}\ col_1, \\
\vdots \\
\ e_m^{agg}\ \texttt{AS}\ col_m)
\end{array}
\Rrightarrow_{def}^{sfw} J\{v_1^{\Uparrow}, \ldots v_g^{\Uparrow}\}
\end{array}$$

DEF-GROUPAGG-VOID

$$J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \texttt{groupagg}\ ()\ \ () \Rrightarrow_{def}^{sfw} J^{\Uparrow}$$

Figure 6.19: **Semantics of `groupagg`.**

$$\begin{aligned} e^{agg} ::= {} & \texttt{SUM}(e^{cell}) \\ \mid {} & \texttt{AVG}(e^{cell}) \\ \mid {} & \texttt{MIN}(e^{cell}) \\ \mid {} & \texttt{MAX}(e^{cell}) \\ \mid {} & \texttt{COUNT}(e^{cell}) \\ \mid {} & \texttt{THE}(e^{cell}) \\ \mid {} & \texttt{COUNTSTAR}() \end{aligned}$$

Figure 6.20: **Aggregation functions.**

In the final part of rule DEF-GROUPAGG, each of the g group gets aggregated, formalized as $\cdot \vdash \cdot \Mapsto^{agg}_{def} \cdot$ (see below). The result table accordingly has g rows. The only tuple variable in the result table is aggs.

Rule DEF-GROUPAGG-VOID just forwards the input table. This rule keeps all tuple variables.

6.5 Aggregation Functions

A representative set of aggregation functions (expressions e^{agg}) is listed in Figure 6.20. Conclusions are denoted

$$\boxed{J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e^{agg} \Mapsto^{agg}_{def} v^{\Uparrow}}$$

where

- $J^{\Uparrow}$ is a single group,
- $\Gamma^{\Uparrow}$, $\Delta^{\Uparrow}$ and F are known already,
- e^{agg} is an aggregation expression and
- $v^{\Uparrow} :: \overline{\text{CELL}\Uparrow}$ is the result value.

In this work, all aggregations share the same provenance semantics but different value semantics. We do not discuss the semantics of null in aggregation context.

6.5.1 SUM

$$\frac{\begin{array}{c}n \coloneqq |J^{\Uparrow}| \\ \Big| \; \Gamma^{\Uparrow} \uplus J^{\Uparrow}[\![\, i \,]\!]; \Delta^{\Uparrow}; F \vdash e^{cell} \Mapsto^{cell}_{def} \blacktriangleleft v_i, \mathbb{p}_i \blacktriangleright \; \Big|_{i=1..n} \\ v^{\Uparrow} \coloneqq \blacktriangleleft v_1 + \cdots + v_n, \bigcup_{i=1}^{n} \mathbb{p}_i \blacktriangleright \end{array}}{J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \texttt{SUM}(e^{cell}) \Mapsto^{agg}_{def} v^{\Uparrow}} \;\textsc{def-sum}$$

Value Semantics The SUM aggregate is defined for number types only (INT and DEC in our case), however we do not formalize the type restrictions. The expression e^{cell} evaluates to n pairs of value and provenance annotation. Both get aggregated individually.

Provenance Semantics The intuition for the data provenance of SUM is that *all* values are added up in order to constitute a grand total. We employ $\bigcup$ to combine all sets $\mathbb{p}_i$ of the group.

6.5.2 SUM–Like Provenance Semantics

The aggregates AVG, MIN, MAX and COUNT share the provenance semantics of SUM. More specific provenance semantics for MIN and MAX could be realized (e.g., only the single minimum or maximum input value) but is not pursued in this work. The COUNT aggregate counts values ≠ null but our provenance semantics does not make this distinction.

6.5.3 THE

$$\frac{\begin{array}{c}n \coloneqq |J^{\Uparrow}| \qquad n \geq 1 \\ \Big| \; \Gamma^{\Uparrow} \uplus J^{\Uparrow}[\![\, i \,]\!]; \Delta^{\Uparrow}; F \vdash e^{cell} \Mapsto^{cell}_{def} v_i^{\Uparrow} \qquad (v_i, \mathbb{p}_i) \coloneqq \curlyvee(v_i^{\Uparrow}) \; \Big|_{i=1..n} \\ v_1 = \cdots = v_n \qquad v^{\Uparrow} \coloneqq \blacktriangleleft v_1, \bigcup_{i=1}^{n} \mathbb{p}_i \blacktriangleright \end{array}}{J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \texttt{THE}(\; e^{cell} \;) \Mapsto^{agg}_{def} v^{\Uparrow}} \;\textsc{def-the}$$

Value Semantics Rule DEF-THE formalizes the THE aggregate. A groups (of rows) must have at least one row (or the group does not exist). The first value in this row constitutes the value result of THE. Any other value could be chosen as well. It is checked that all values match, using the two–valued =.

Provenance Semantics The provenance semantics is consistent with `MIN` and `MAX`. One single value is selected as result but the data provenance is derived from all values.

6.5.4 `COUNTSTAR()`

$$\dfrac{\textsc{def-countstar}}{J^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash \texttt{COUNTSTAR()} \Mapsto^{agg}_{def} \lhd |J^{\Uparrow}|, \varnothing \rhd}$$

Value Semantics The `COUNTSTAR()` aggregate yields the cardinality of a table.

Provenance Semantics As a design decision, we yield the empty data provenance for `COUNTSTAR()`. The argument is similar to Section 6.1.8.1, i.e. aggregating the entire table is considered too unspecific.

6.6 Window Functions

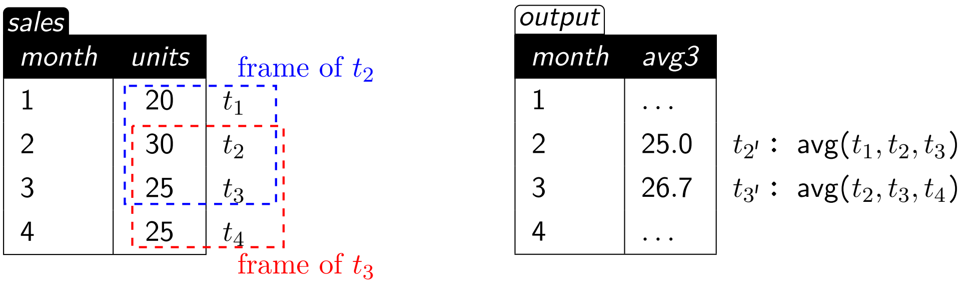
6.6.1 Introductory Example

We aim for partial support of SQL window functions. In contrast to aggregate functions, a window function does not change the row count. We call the row currently being processed the *current row*. The example from Figure 6.21 lists the input table *sales*. The *window frames* of t_2 and t_3 are highlighted. The window functions yields column *avg3* (see table *output*) which is the specific average per frame. Any aggregate function (here: `AVG`) can be turned into a window function (which is the focus of our work). The corresponding query is listed in Figure 6.21(c). On line `6`, `AVG` decorated with the additional `OVER` clause makes it a window function. The `OVER` clause is syntactically rich.

- `PARTITION BY` (very similar to `GROUP BY`): distribute the input rows over partitions; the window frames will not cross partitions borders (but process each partition individually). Without partition criteria, the entire input table becomes one partition.
- `ORDER BY` specifies the row order. Each partition gets sorted individually.
- `ROWS` determines the frame width, i.e. the number of *preceding* and *following* rows. Special values are 0 (which denotes the current row) and `NULL` (which denotes the first or last row in the partition). Negative values would raise an error.

The example query specifies a sliding window of width 3. The expression `ROWS 1 1` specifies

- `1` preceding row,



month	units	
1	20	t_1
2	30	t_2
3	25	t_3
4	25	t_4

(a) **Input table.**

month	avg3	
1	...	
2	25.0	$t_{2'}$: avg(t_1, t_2, t_3)
3	26.7	$t_{3'}$: avg(t_2, t_3, t_4)
4	...	

(b) **Output table.**

```
       FROM sales LATERAL () AS s
      WHERE TRUE
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS AVG(s.units) OVER (PARTITION BY ()
                                   ORDER BY s.month ASC
                                       ROWS 1 1        ) AS w1
     SELECT s.month AS month,
            wins.w1 AS avg3
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

(c) **SQL query with a window function.**

Figure 6.21: **Example: syntax and semantics of a window function.**

- the current row and
- `1` following row.

Frames at partition borders get shortened accordingly (not exemplified).

6.6.2 `windows`

As a reminder, entries of the `windows` clause are denoted

$$e^{win}\ \texttt{OVER}\ e^{over}\ \texttt{AS}\ col$$

. The expansion rules for e^{over} and e^{win} are provided below. Our provenance definition is restricted to window functions derived from aggregations.

$$\begin{aligned} e^{over} ::= (\quad & \texttt{PARTITION BY}\ (e^{cell}, \ldots) \\ & \texttt{ORDER BY}\ (e^{cell} asc, \ldots) \\ & \texttt{ROWS}\ \ell\ \ell \\ &) \\ e^{win} ::= e^{agg} & \end{aligned}$$

Semantics Rule DEF-WINDOWS is listed in Figure 6.22. Basically, k window functions for i rows get evaluated. Each row corresponds to a window frame which is evaluated using the auxiliary function $\mathsf{getframe}^{\Uparrow}(\cdot)$. This auxiliary function hides a considerable amount of complexity and is not formalized. All environments are handed over to $\mathsf{getframe}^{\Uparrow}(\cdot)$ in order to evaluate sub–expressions. Also, i is handed over to determine the current row (=frame). The semantics of `PARTITION BY` and `ORDER BY` correspond to the semantics of `GROUP BY` and `ORDER BY`, respectively. The window frames $J^{\Uparrow}_{j,k}$ are specific for the i–th current row and for the k–th window function.

In the next step, $\cdot \vdash \cdot \Mapsto^{win}_{def} \cdot$ is evaluated (formalized below) which yields one cell value $v^{\Uparrow}_{i,k}$ for each frame. The $v^{\Uparrow}_{i,k}$ get bound to column names and the artificial tuple variable `wins` is introduced (analogous to aggregations).

6.6.3 Window Functions e^{win}

The conclusion for a window function e^{win} is denoted

$$\boxed{J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e^{win} \Mapsto^{win}_{def} v^{\Uparrow}}$$

where

DEF-WINDOWS

$$\frac{\begin{array}{c} n := |J^{\Uparrow}| \\ \left| J^{\Uparrow}_{i,k} := \mathsf{getframe}^{\Uparrow}\left(J^{\Uparrow}, \Gamma^{\Uparrow}, \Delta^{\Uparrow}, F, i, \begin{array}{r} \texttt{PARTITION BY } (e^{cell}_{par,k,1},\ \ldots) \\ \texttt{ORDER BY } (e^{cell}_{ord,k,1}\ \ asc_{k,1},\ \ldots) \\ \texttt{ROWS } \ell_{prec,k}\ \ \ell_{foll,k} \end{array} \right) \right|_{i=1..n,k=1..m} \\ \left| J^{\Uparrow}_{i,k}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash e^{win}_{k} \Mapsto^{win}_{def} v^{\Uparrow}_{i,k} \right|_{i=1..n,k=1..m} \\ \left| v^{\Uparrow}_{i} := \langle col_1 \mapsto v^{\Uparrow}_{i,1}, \ldots col_m \mapsto v^{\Uparrow}_{i,m} \rangle \right|_{i=1..n} \\ J^{\Uparrow}_{res} := J \llparenthesis J^{\Uparrow}[\![1]\!] \uplus \{\!|\texttt{wins} \mapsto v^{\Uparrow}_{1}|\!\}, \ldots J^{\Uparrow}[\![n]\!] \uplus \{\!|\texttt{wins} \mapsto v^{\Uparrow}_{n}|\!\} \rrparenthesis \end{array}}{J^{\Uparrow}; \Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \begin{array}{l} \texttt{windows} \\ \quad e^{win}_{1}\ \texttt{OVER (} \\ \qquad \texttt{PARTITION BY } (e^{cell}_{par,1,1},\ \ldots) \\ \qquad\quad \texttt{ORDER BY } (e^{cell}_{ord,1,1}\ \ asc_{1,1},\ \ldots) \\ \qquad\qquad \texttt{ROWS } \ell_{prec,1}\ \ \ell_{foll,1} \\ \quad \texttt{) AS } col_1, \\ \quad \vdots \\ \quad e^{win}_{m}\ \texttt{OVER (} \\ \qquad \texttt{PARTITION BY } (e^{cell}_{par,m,1},\ \ldots) \\ \qquad\quad \texttt{ORDER BY } (e^{cell}_{ord,m,1}\ \ asc_{m,1},\ \ldots) \\ \qquad\qquad \texttt{ROWS } \ell_{prec,m}\ \ \ell_{foll,m} \\ \quad \texttt{) AS } col_m \end{array} \Mapsto^{sfw}_{def} J^{\Uparrow}_{res}}$$

Figure 6.22: **Semantics of `windows`.**

- $J^{\Uparrow}$ is the current frame,
- $\Gamma^{\Uparrow}$, $\Delta^{\Uparrow}$ and F are known already,
- e^{win} is the window function and
- $v^{\Uparrow} :: \overline{\text{CELL}\Uparrow}$ is the result value for this window function and frame.

6.6.3.1 Aggregation to Window Function

$$\frac{J^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e^{win} \Mapsto^{agg}_{def} v^{\Uparrow}}{J^{\Uparrow};\Gamma^{\Uparrow};\Delta^{\Uparrow};F \vdash e^{win} \Mapsto^{win}_{def} v^{\Uparrow}}\ \text{DEF-WINDOW-FUNCTION}$$

In this work, we only define the data provenance for window functions e^{win} created from aggregation functions e^{agg}. The corresponding rule is DEF-WINDOW-FUNCTION. Supporting *real* window functions is considered future work.

6.7 Data Provenance for SQL

Based upon the previous definitions of a SQL (backend) dialect and its provenance semantics, we can now formalize the *data provenance* for SQL.

Definition 6.1: Data Provenance

Let $\Delta :: \overline{\text{DB}}$ be a database instance and $\Delta^{\Uparrow} = \mathsf{liftrm}(\Delta)$ be the lifted database. Let $Q ::$ ETABLE be a table expression and F the associated UDF definitions. Let the definitional interpreter

$$\{\!|\,|\!\};\Delta^{\Uparrow};F \vdash Q \Mapsto^{table}_{def} v^{\Uparrow}$$

evaluate to $v^{\Uparrow}$.
Then $v^{\Uparrow}$ is the **data provenance** regarding to Δ, Q and F, denoted

$$\mathsf{Prov}(\mathsf{liftrm}(\Delta), Q, F) = v^{\Uparrow}$$

.

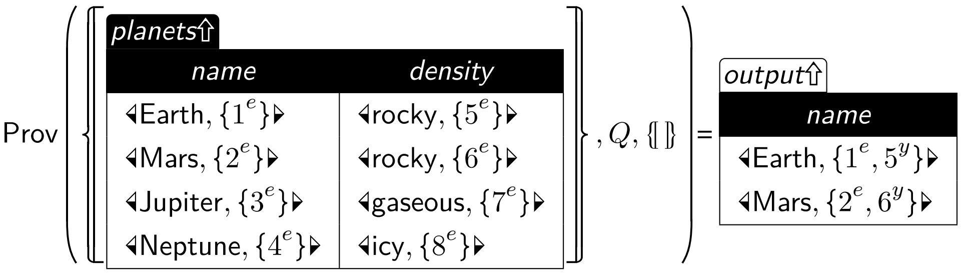
Example We repeat the example from Section 5.1 using the notation from Definition 6.1. Query Q' listed in Figure 6.23(a) is formulated in a frontend query dialect (see our distinction in Figure 6.1). The formal part of the provenance analysis starts

```
SELECT p.name AS name
  FROM planets AS p
 WHERE p.density='rocky'
```

(a) **Raw query Q' (frontend dialect).**

```
       FROM planets AS p
      WHERE p.density='rocky'
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT p.name AS name
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

(b) **Query Q (backend dialect).**



$\mathsf{Prov}\left(\left[\!\left[\text{planets}\right]\!\right], Q, \{\!|\,|\!\}\right) =$ output

planets⇧

name	density
$\blacktriangleleft$Earth, $\{1^e\}\blacktriangleright$	$\blacktriangleleft$rocky, $\{5^e\}\blacktriangleright$
$\blacktriangleleft$Mars, $\{2^e\}\blacktriangleright$	$\blacktriangleleft$rocky, $\{6^e\}\blacktriangleright$
$\blacktriangleleft$Jupiter, $\{3^e\}\blacktriangleright$	$\blacktriangleleft$gaseous, $\{7^e\}\blacktriangleright$
$\blacktriangleleft$Neptune, $\{4^e\}\blacktriangleright$	$\blacktriangleleft$icy, $\{8^e\}\blacktriangleright$

output⇧

name
$\blacktriangleleft$Earth, $\{1^e, 5^y\}\blacktriangleright$
$\blacktriangleleft$Mars, $\{2^e, 6^y\}\blacktriangleright$

(c) **Provenance analysis of Q.**

Figure 6.23: **An example *data provenance* according to Definition 6.1.**

with query Q of Figure 6.23(b), formulated in the backend dialect. The data provenance according to $\mathsf{Prov}(\cdot,\cdot,\cdot)$ is listed in Figure 6.23(c).

Outlook The provenance definition $\mathsf{Prov}(\cdot,\cdot,\cdot)$ is not designed to be implemented. The definition employs pairs of value and provenance (i.e., $\blacktriangleleft v, \mathbb{p} \blacktriangleright$). However, SQL fragments like . . . `WHERE` $\blacktriangleleft$`TRUE`, $\mathbb{p}_\Box\blacktriangleright$ . . . cannot be evaluated using an unmodified DBMS. The `WHERE` predicate must yield boolean values.

We present our solution to this implementation issue in Chapter 8. The core idea is to process values v and provenance $\mathbb{p}$ in two different queries.

Part III

Provenance Analysis

7 Normalization

This chapter introduces a *normalization* procedure. The normalization is an important service which helps to make the rewrite rules for provenance analysis (subject of the next chapter) simpler. The normalization uses expression duplication. This is not an issue since we are restricted to a read–only query dialect.

Related Work This normalization strategy is based on [MDG18a] by T. Müller, B. Dietrich and T. Grust.

7.1 Definition

Definition 7.1: Query Normalization

Let e^{table} be a query formulated in the backend SQL dialect and F its corresponding UDF definitions.
The **query normalization**

$$\begin{aligned}&\mathsf{normalize}(\cdot,\cdot) :: (\textsc{etable}, \overline{\textsc{udfs}}) \twoheadrightarrow \textsc{etable}\\&\mathsf{normalize}(e^{table}, F) = e^{table}_{norm}\end{aligned}$$

consists of the sequential rewriting steps

- UDF inlining,
- correlation normalization and
- SFW normalization.

We specify all three normalization steps in the body of this chapter.

7.2 UDF Inlining

Our approach to provenance analysis (subject of the next chapter) augments tuple variables with context information (in an additional attribute). However, an UDF call like

$$udf(var\,.\,col)$$

propagates the contents of attribute *col* towards the UDF body, ignoring the remaining attributes. This is an issue for non–trivial UDF bodies (for example, a body with an SFW expression). We consider the support of recursive UDFs future work.

In this work, we employ *inlining* and support only non–recursive UDFs. For example, let

- $F \coloneqq \{\!\!\{ \texttt{func} \mapsto ([\texttt{x}], \texttt{x+x}) \}\!\!\}$ be the UDF environment and
- $q \coloneqq \texttt{func}(e_1^{cell})$ be a query fragment.

Through inlining, the UDF call gets substituted with the UDF body (here: `x+x`) and the parameters (here: `[x]`) get substituted with the argument expression. The resulting query fragment is $q' \coloneqq e_1^{cell}\texttt{+}e_1^{cell}$.

7.3 Correlation Normalization

Extended SQL Dialect For the purpose of correlation normalization, we augment the backend dialect with an additional table expression e^{table} :: ETABLE. Its syntax is

$$\texttt{BIND}\ var$$

and it is semantically equivalent to `SELECT` *var*.$*$, i.e. all attributes of the correlated tuple variable are selected. The resulting table has cardinality 1, because the tuple variable always binds a single row. The main features of `BIND` are that it provides protection from later rewrite steps and it is way more compact than the full SFW expression of the backend dialect. As a final rewrite step (directly before query evaluation), `BIND` is turned into `SELECT`, as explained above.

Correlation A correlated subquery is a subquery which references a free tuple variable. The example query of Figure 7.1 computes the planets which have a moon orbiting them. The subquery on lines 2-12 references `p` which gets bound in the outer query. The subquery gets evaluated for each valuation of `p`. We observe that this semantics is basically a cross join between *planets* and *moons*.

```
       FROM planets LATERAL () AS p
      WHERE EXISTS(       FROM moons LATERAL () AS m
                         WHERE p.name=m.planet -- 'p' is a free
                                               --   variable
                      GROUP BY ()
                    AGGREGATES ()
                        HAVING TRUE
                       WINDOWS ()
                        SELECT TRUE
                      ORDER BY ()
                   DISTINCT ON ()
                        OFFMIT NULL NULL)
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT p.name AS has_moons
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

(a) **SFW expression with a correlated subquery.**

planets

name	*density*	
Mercury	rocky	t_1
Venus	rocky	t_2
Earth	rocky	t_3
Mars	rocky	t_4

moons

name	*planet*	
Luna	Earth	t_{11}
Phobos	Mars	t_{12}
Deimos	Mars	t_{13}

(b) **Input tables.**

output

has_moons	
Earth	t_{21}
Mars	t_{22}

(c) **Result table.**

Figure 7.1: **Example query: which planets have moons?**

```
       FROM planets LATERAL () AS p
      WHERE EXISTS(       FROM moons LATERAL () AS m,
                               BIND p  -- outer ’p’ (correlated)
                                 LATERAL () AS p
                         WHERE p.name=m.planet -- ’p’ is a bound
                                               --   variable
                      GROUP BY ()
                    AGGREGATES ()
                        HAVING TRUE
                       WINDOWS ()
                        SELECT TRUE
                      ORDER BY ()
                   DISTINCT ON ()
                        OFFMIT NULL NULL)
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT p.name AS has_moons
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

Figure 7.2: **Query from Figure 7.1(a) after normalization.**

Example Rewrite Figure 7.2 lists the rewritten example query which makes the implicit join semantics explicit. Variable p gets added to the entries in the `FROM` clause (line 3) and gets bound to a fresh p (line 4), shadowing the first one. The query semantics does not change, because the value of p stays the same and there are as many subquery evaluations as there have been before. The essential outcome of this rewrite is that all variables (correlated and not) get bound in the `FROM` clause and later rewrite steps can treat all variables uniformly. The `BIND` expression protects those artificial `FROM` entries from getting rewritten any further.

Normalization Algorithm This rewrite step traverses a query Q and triggers the following steps for any `FROM` clause.

- Scan all subexpressions of the current SFW expression for free variables (unlimited scan depth).
- If a free variable *var* is found, add the entry

  `BIND` *var* `LATERAL () AS` *var*

  to the `FROM` clause.

The rewritten query Q' is input to the SFW normalization, see below.

Correctness Duplicate `BIND` (or `BIND`–equivalent) entries in the same `FROM` clause are not an issue.

Argument: Let the unmodified `FROM` clause yield n rows. The rewrite adds an additional join partner (i.e. the `BIND` expression) which evaluates to tables of cardinality 1. The rewritten `FROM` clause yields $n \cdot 1 = n$ rows.

7.4 SFW Normalization

7.4.1 Motivation and Example

The SFW expression poses a challenge due to its size in in notation and complexity in semantics. Regarding to the provenance definition (carried out in Chapter 6), the 10 clauses of the SFW expression got formalized individually. In order to stay compatible with DBMS implementations of SQL, the normalization is designed to be less invasive. The basic idea of this SFW normalization is to create additional SFW expression (ratio: one in, five out). Each of the normalized SFW expressions only deals with a fraction of the entire SFW logic. As an example input to normalization, see the query in Figure 7.3.

```
      FROM planets LATERAL () AS p,
           moons   LATERAL () AS m
     WHERE p.id=m.planet
  GROUP BY ()
AGGREGATES ()
    HAVING TRUE
   WINDOWS ()
    SELECT p.name AS planet,
           m.name AS moon
  ORDER BY ()
DISTINCT ON ()
    OFFMIT NULL NULL
```

(a) **Query before normalization.**

planets

id	*name*	
1	Mercury	t_1
2	Venus	t_2
3	Earth	t_3
4	Mars	t_4

moons

name	*planet*	
Luna	3	t_{11}
Phobos	4	t_{12}
Deimos	4	t_{13}

(b) **Input tables.**

output

planet	*moon*	
Earth	Luna	t_3
Mars	Phobos	t_4
Mars	Deimos	t_4

(c) **Output table.**

Figure 7.3: **Example: input query and tables.**

The query consists of a simple join of rocky planets and their moons. The column *planets . id* is a primary key.

Through normalization, the query listed in Figure 7.4 is produced. The inner SFW expression carries out the join logic and provides a single tuple variable `v` to the outer SFW expression. The outer expression has its focus on a different task, i.e. the evaluation of the original `SELECT` with the final column names *planet* and *moon*. The two nested SFW expressions implement the same value semantics as the query in Figure 7.3.

7.4.2 Overview

In this section, we specify *five normalized SFW expressions*. Each of them has its focus on a certain responsibility (e.g., join logic). The query listed in Figure 7.4 is fully normalized already, i.e. it utilizes only two of the five expressions. The five normalized SFW expressions can be distinguished in later rewrite steps, using pattern matching. The structure of the SFW normalization is sketched in Figure 7.5. For each SFW expression

```
       FROM (
                        FROM planets LATERAL () AS p,
                             moons   LATERAL () AS m
                       WHERE p.id=m.planet
                    GROUP BY ()
                  AGGREGATES ()
                      HAVING TRUE
                     WINDOWS ()
                      SELECT p.name AS p_name,
                             m.name AS m_name
                    ORDER BY ()
                 DISTINCT ON ()
                      OFFMIT NULL NULL
             ) LATERAL () AS v
      WHERE TRUE
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT v.p_name AS planet,
            v.m_name AS moon
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

Figure 7.4: **Normalization of the SFW expression from Figure 7.3: the *join* logic sits in a dedicated SFW expression.**

$$\mathsf{normalizeSFW}(e) \coloneqq \begin{array}{l} \mathsf{normalizeOrderBy}(\\ \quad \mathsf{normalizeWindows}(\\ \quad\quad \mathsf{normalizeAggregations}(\\ \quad\quad\quad \mathsf{normalizeFrom}(e) \\ \quad\quad) \\ \quad) \\ ) \end{array}$$

Figure 7.5: **Normalization procedure of the SFW expression.**

e, the four normalization steps are carried out.

7.4.3 `FROM` and `WHERE`

Normalization of `FROM` is the first normalization step and hence applies to the most general SFW expression e listed in Figure 7.6(a). For now, we assume that there exist ≥ 2 entries in the `FROM` clause. The cases < 2 are discussed afterwards.

Figure 7.6(b) lists the result expression $\mathsf{normalizeFrom}(e)$ which consists of two (nested) SFW expressions. This duplication implements a separation of responsibilities regarding to join logic:

- The inner SFW expression specifies multiple tuple variables (on line 2) and a predicate e_w^{cell} (on line 3).
- The outer SFW expression is only aware of a single, artificial tuple variable `v` (bound on line 12).

For consistency, the auxiliary function $\mathsf{sub}(\cdot)$ (short for: *substitute*) turns any variable reference into a reference of `v`. The `SELECT` ($e_{v,1}^{cell}$ `AS` $col_{v,1}$, …) clause on line 8 accordingly propagates all referenced columns. If conflicts in column names would occur, according renaming is carried out. For example, lines 9-10 in Figure 7.4 introduce the prefixed column names `p_name` and `m_name` in order to avoid such conflict.

7.4.3.1 Case Distinction

The number of n bindings in the input `FROM` (e_1^{from}, …, e_n^{from}) determines three cases:

- $n \geq 2$: Discussed above.
- $n = 1$: The tuple variable is renamed to `v` and references are updated. If the input query specifies `WHERE TRUE`, the SFW expression is not duplicated.

```
      FROM ($e_1^{from}$, ...)
     WHERE $e_w^{cell}$
  GROUP BY ($e_{g,1}^{cell}$, ...)
AGGREGATES ($e_1^{agg}$ AS $col_{a,1}$, ...)
    HAVING $e_h^{cell}$
   WINDOWS ($e_1^{over}$ AS $col_{w,1}$, ...)
    SELECT ($e_{s,1}^{cell}$ AS $col_{s,1}$, ...)
  ORDER BY ($e_{o,1}^{cell}$ ASC|DESC, ...)
DISTINCT ON ($e_{d,1}^{cell}$, ...)
    OFFMIT $\ell_{off}$ $\ell_{lim}$
```

(a) **Expression e.**

```
      FROM (
                  FROM ($e_1^{from}$, ...)
                 WHERE $e_w^{cell}$
              GROUP BY ()
            AGGREGATES ()
                HAVING TRUE
               WINDOWS ()
                SELECT ($e_{v,1}^{cell}$ AS $col_{v,1}$, ...)
              ORDER BY ()
           DISTINCT ON ()
                OFFMIT NULL NULL
           ) LATERAL () AS v
     WHERE TRUE
  GROUP BY (sub($e_{g,1}^{cell}$), ...)
AGGREGATES (sub($e_1^{agg}$) AS $col_{a,1}$, ...)
    HAVING sub($e_h^{cell}$)
   WINDOWS (sub($e_1^{over}$) AS $col_{w,1}$, ...)
    SELECT (sub($e_{s,1}^{cell}$) AS $col_{s,1}$, ...)
  ORDER BY (sub($e_{o,1}^{cell}$) ASC|DESC, ...)
DISTINCT ON (sub($e_{d,1}^{cell}$), ...)
    OFFMIT $\ell_{off}$ $\ell_{lim}$
```

(b) **Expression** normalizeFrom(e)**.**

Figure 7.6: **Normalization of join logic.**

- $n = 0$: A dummy `FROM (VALUES (ROW())) LATERAL () AS v` is created, i.e. a single row with zero columns. If a `WHERE TRUE` is specified in the input, the inner SFW expression is skipped.

Outcome After the `FROM` normalization has been carried out, the outer SFW expression only knows about variable v and this variable always exists. Join logic and `WHERE` predicate are specified in their dedicated subquery.

7.4.4 `GROUP BY`, `AGGREGATES` and `HAVING`

This normalization step isolates aggregation logic from the rest of the SFW expression. That means, it focuses on the clauses `GROUP BY`, `AGGREGATES` and `HAVING`. It is a feature of our backend SQL dialect that aggregation functions only occur in the `AGGREGATES` clause and nowhere else.

If the `AGGREGATES` clause is empty (i.e. `AGGREGATES ()`) the query is non–aggregating. In such cases, this normalization step is skipped. Otherwise, normalization is carried out according to Figure 7.7. The input expression e is already normalized regarding

```
         FROM $e^{table}$ LATERAL () AS v
        WHERE TRUE
     GROUP BY ($e^{cell}_{g,1}$, ...)
   AGGREGATES ($e^{agg}_{1}$ AS $col_{a,1}$, ...)
       HAVING $e^{cell}_{h}$
      WINDOWS ($e^{over}_{1}$ AS $col_{w,1}$, ...)
       SELECT ($e^{cell}_{s,1}$ AS $col_{s,1}$, ...)
     ORDER BY ($e^{cell}_{o,1}$ ASC|DESC, ...)
  DISTINCT ON ($e^{cell}_{d,1}$, ...)
       OFFMIT $\ell_{off}$ $\ell_{lim}$
```

(a) **Expression e.**

```
         FROM (
                  FROM $e^{table}$ LATERAL () AS v
                 WHERE TRUE
              GROUP BY ($e^{cell}_{g,1}$, ...)
            AGGREGATES ($e^{agg}_{1}$ AS $col_{a,1}$, ...)
                HAVING $e^{cell}_{h}$
               WINDOWS ()
                SELECT (aggs.$col_{a,1}$ AS $col_{a,1}$, ...)
              ORDER BY ()
           DISTINCT ON ()
                OFFMIT NULL NULL
              ) LATERAL () AS v
        WHERE TRUE
     GROUP BY ()
   AGGREGATES ()
       HAVING TRUE
      WINDOWS (sub($e^{over}_{1}$) AS $col_{w,1}$, ...)
       SELECT (sub($e^{cell}_{s,1}$) AS $col_{s,1}$, ...)
     ORDER BY (sub($e^{cell}_{o,1}$) ASC|DESC, ...)
  DISTINCT ON (sub($e^{cell}_{d,1}$), ...)
       OFFMIT $\ell_{off}$ $\ell_{lim}$
```

(b) **Expression** normalizeAggregations(e)**.**

Figure 7.7: **Normalization of aggregation logic.**

to FROM and WHERE (discussed above). On the right–hand side, normalizeAggregations(e) assembles two nested SFW expressions.

- The inner SFW expression carries out regular grouping, aggregation and filtering. Aggregation results are bound to the aggs tuple variable. The (inner) SELECT clause references all aggregated columns and makes them available to the outer SFW expression.

- The outer SFW expression binds the aggregation results to v and the auxiliary function sub(·) renames references of aggs into v.

Outcome After this normalization step has been carried out, the outer SFW expression is unaware of any aggregation logic.

7.4.5 WINDOWS

The windows normalization is analogous to the normalization of aggregations. If the WINDOWS–clause is non–empty, this normalization is carried out. The function sub(·) replaces references to wins with references to v. Column name conflicts may occur (see

```
       FROM $e^{table}$ LATERAL () AS v
      WHERE TRUE
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ($e_1^{over}$ AS $col_{w,1}$, ...)
     SELECT ($e_{s,1}^{cell}$ AS $col_{s,1}$, ...)
   ORDER BY ($e_{o,1}^{cell}$ ASC|DESC, ...)
DISTINCT ON ($e_{d,1}^{cell}$, ...)
     OFFMIT $\ell_{off}$ $\ell_{lim}$
```

(a) **Expression** e**.**

```
       FROM (
                  FROM $e^{table}$ LATERAL () AS v
                 WHERE TRUE
              GROUP BY ()
            AGGREGATES ()
                HAVING TRUE
               WINDOWS ($e_1^{over}$ AS $col_{w,1}$, ...)
                SELECT (v.$col_{v,1}$ AS $col_{v,1}$, ...,
                        wins.$col_{w,1}$ AS $col_{w,1}$, ...)
              ORDER BY ()
           DISTINCT ON ()
                OFFMIT NULL NULL
              ) LATERAL () AS v
      WHERE TRUE
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT (sub($e_{s,1}^{cell}$) AS $col_{s,1}$, ...)
   ORDER BY (sub($e_{o,1}^{cell}$) ASC|DESC, ...)
DISTINCT ON (sub($e_{d,1}^{cell}$), ...)
     OFFMIT $\ell_{off}$ $\ell_{lim}$
```

(b) **Expression** normalizeWindows(e)**.**

Figure 7.8: **Normalization of window function logic.**

lines 8-9 in Figure 7.8(b)). If $col_{v,i} = col_{wins,k}$ (with i and k a valid index in range), this can be resolved by adding prefixes to column names.

7.4.6 `ORDER BY`, `DISTINCT ON` and `OFFMIT`

This final normalization step checks for non–trivial `ORDER BY`, `DISTINCT ON` and `OFFMIT` clauses. If at least one is non–trivial, the normalization according to Figure 7.9 produces an additional (outer) SFW expression. Expression normalizeOrderBy(e) consists of two nested SFW expressions:

- The inner SFW expression evaluates the original `SELECT` clause. Additionally, all existing columns in tuple variable v are propagated outwards. Conflicts in column names may occur. These can be resolved using prefixes (not formalized here).

- The outer SFW expression evaluates the original `ORDER BY`, `DISTINCT ON` and `OFFMIT` clauses.

```
       FROM $e^{table}$ LATERAL () AS v
      WHERE TRUE
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT ($e^{cell}_{s,1}$ AS $col_{s,1}$, ...)
   ORDER BY ($e^{cell}_{o,1}$ ASC|DESC, ...)
DISTINCT ON ($e^{cell}_{d,1}$, ...)
     OFFMIT $\ell_{off}$ $\ell_{lim}$
```

(a) **Expression e.**

```
       FROM (
                   FROM $e^{table}$ LATERAL () AS v
                  WHERE TRUE
               GROUP BY ()
             AGGREGATES ()
                 HAVING TRUE
                WINDOWS ()
                 SELECT ($e^{cell}_{s,1}$ AS $col_{s,1}$, ...,
                         v.$col_{v,1}$ AS $col_{v,1}$, ...)
               ORDER BY ()
            DISTINCT ON ()
                 OFFMIT NULL NULL
            ) LATERAL () AS v
      WHERE TRUE
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT (v.$col_{s,1}$ AS $col_{s,1}$, ...)
   ORDER BY ($e^{cell}_{o,1}$ ASC|DESC, ...)
DISTINCT ON ($e^{cell}_{d,1}$, ...)
     OFFMIT $\ell_{off}$ $\ell_{lim}$
```

(b) **Expression** normalizeOrderBy(e)**.**

Figure 7.9: **Normalization of sort logic.**

7.4.7 The Fully Normalized SFW Expression

After the four normalization steps have been carried out, a single SFW expression may have turned into five nested SFW expressions. Their individual responsibilities are

- join logic,
- aggregation logic,
- window function logic,
- select logic and
- sort logic.

The huge benefit of this normalization is that the provenance analysis can be structured accordingly. Instead of rewriting a single (and complex) SFW expression, five more lightweight SFW expressions are to be rewritten. We found no impact on the query performance (subject of Chapter 9).

8 Detached Provenance Analysis

8.1 Overview

We introduce a method which derives the data provenance P for a SQL query Q and a database Δ. Figure 8.1 provides an overview of the problem statement (on the left, according to Definition 6.1[1]) and our solution (on the right) being called *detached provenance analysis*. Looking at the right–hand side, the main feature of our analysis method is to derive the data provenance P in two computation steps.

- Phase $\mathbb{1}$ has its sole focus on evaluating predicates. The specific outcomes (per tuple and predicate) are recorded and stored for later use.
- Phase $\mathbb{2}$ has its sole focus on propagating provenance annotations and does not access the input database Δ. When the outcome of a specific predicate is required, phase $\mathbb{2}$ looks up the recorded data from phase $\mathbb{1}$.

Put in other words, the two phases are designed to realize a *separation of concerns*. The two concerns are the domain of values (phase $\mathbb{1}$) and the domain of provenance annotations (phase $\mathbb{2}$). This separation makes the detached provenance analysis executable using existing, unmodified DBMS infrastructure. A modern DBMS is the result of decades of database research. The detached provenance analysis leverages this research.

Related work The term *detached provenance analysis* is new but one of the earliest works on the topic has been published by T. Müller and T. Grust [MG15]. The approach of that particular work employs compilation of the SQL query into an imperative program. Then, an earlier version of the detached analysis is used to analyze the imperative program and indirectly, yield the data provenance for the original SQL query. The most recent work by T. Müller, B. Dietrich and T. Grust [MDG18a] skips the compilation step and carries out the detached analysis directly on SQL level. This thesis is based on the approaches of [MDG18a].

C. Jay and J. Cockett [JC94] introduce the perspective of *shape*. For example, a list $[42, 43]$ can be decomposed into its data (e.g., 42 and 43) and its shape (e.g., $[\square, \square]$).

[1] For presentation reasons, we approximate $\mathsf{Prov}(\cdot,\cdot,\cdot)$ with $\sim\!\mathsf{Prov}(\cdot,\cdot)$.

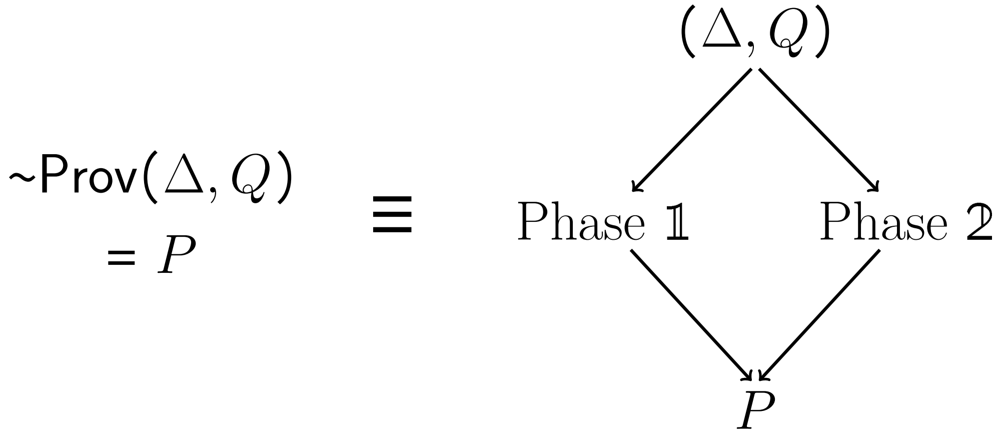


Figure 8.1: **A provenance analysis (left) turned into a *detached provenance analysis* (right).**

In our phase 𝟚, the shape is kept but all data is replaced with provenance annotations (e.g., $[\mathbb{p}_1, \mathbb{p}_2]$).

8.2 Introductory Example

We exemplify the phases 𝟙 and 𝟚 and explain how they work together in carrying out a provenance analysis.

Example Query Query Q from Figure 8.2 is the running example. Semantically, it selects planets with rocky density. For the example database Δ, two rows get filtered and two are in the result table. In general, the query normalization of Chapter 7 is required before the detached analysis takes place. Exclusively because Q is rather simple, its normalization can be skipped.

Data Provenance The data provenance according to Definition 6.1 is denoted

$$\mathsf{Prov}(\mathsf{liftrm}(\Delta), Q, F)$$

. Applying this formalism (UDFs are not defined, hence $F \coloneqq \{\!\!\{\,\}\!\!\}$) to the running example yields the provenance results presented in Figure 8.3. It is the goal of the detached provenance analysis to compute the very same result using a DBMS as backend.

```
      FROM planets LATERAL () AS p
     WHERE p.density='rocky'
  GROUP BY ()
AGGREGATES ()
    HAVING TRUE
   WINDOWS ()
    SELECT p.name AS name
  ORDER BY ()
DISTINCT ON ()
    OFFMIT NULL NULL
```

(a) **Query Q formulated in the backend SQL dialect.**

$\Delta :=$ {⟦ *planets*

name	*density*	
Earth	rocky	t_1
Mars	rocky	t_2
Jupiter	gaseous	t_3
Neptune	icy	t_4

⟧}

(b) **Input database Δ.**

output

name	
Earth	t_{101}
Mars	t_{102}

(c) **Result table.**

Figure 8.2: **Example query and data.**

Prov ({⟦ *planets*⇧

name	*density*	
◂Earth, $\{11^e\}$▸	◂rocky, $\{21^e\}$▸	t_1
◂Mars, $\{12^e\}$▸	◂rocky, $\{22^e\}$▸	t_2
◂Jupiter, $\{13^e\}$▸	◂gaseous, $\{23^e\}$▸	t_3
◂Neptune, $\{14^e\}$▸	◂icy, $\{24^e\}$▸	t_4

⟧}, Q, {| |})

= *output*⇧

name	
◂Earth, $\{11^e, 21^y\}$▸	t_{101}
◂Mars, $\{12^e, 22^y\}$▸	t_{102}

Figure 8.3: **Example data provenance.**

output⇧

name	
◂Earth, $\{11^e, 21^y\}$▸	t_{101}
◂Mars, $\{12^e, 22^y\}$▸	t_{102}

(a) **Original table (for reference).**

*output*𝟙

ρ	*name*
t_{101}	Earth
t_{102}	Mars

(b) **Output table in phase 𝟙.**

*output*𝟚

ρ	*name*
t_{101}	$\{11^e, 21^y\}$
t_{102}	$\{12^e, 22^y\}$

(c) **Output table in phase 𝟚.**

Figure 8.4: **The output table from Figure 8.3 in the detached approach.**

8.2.1 *One* Table Detached

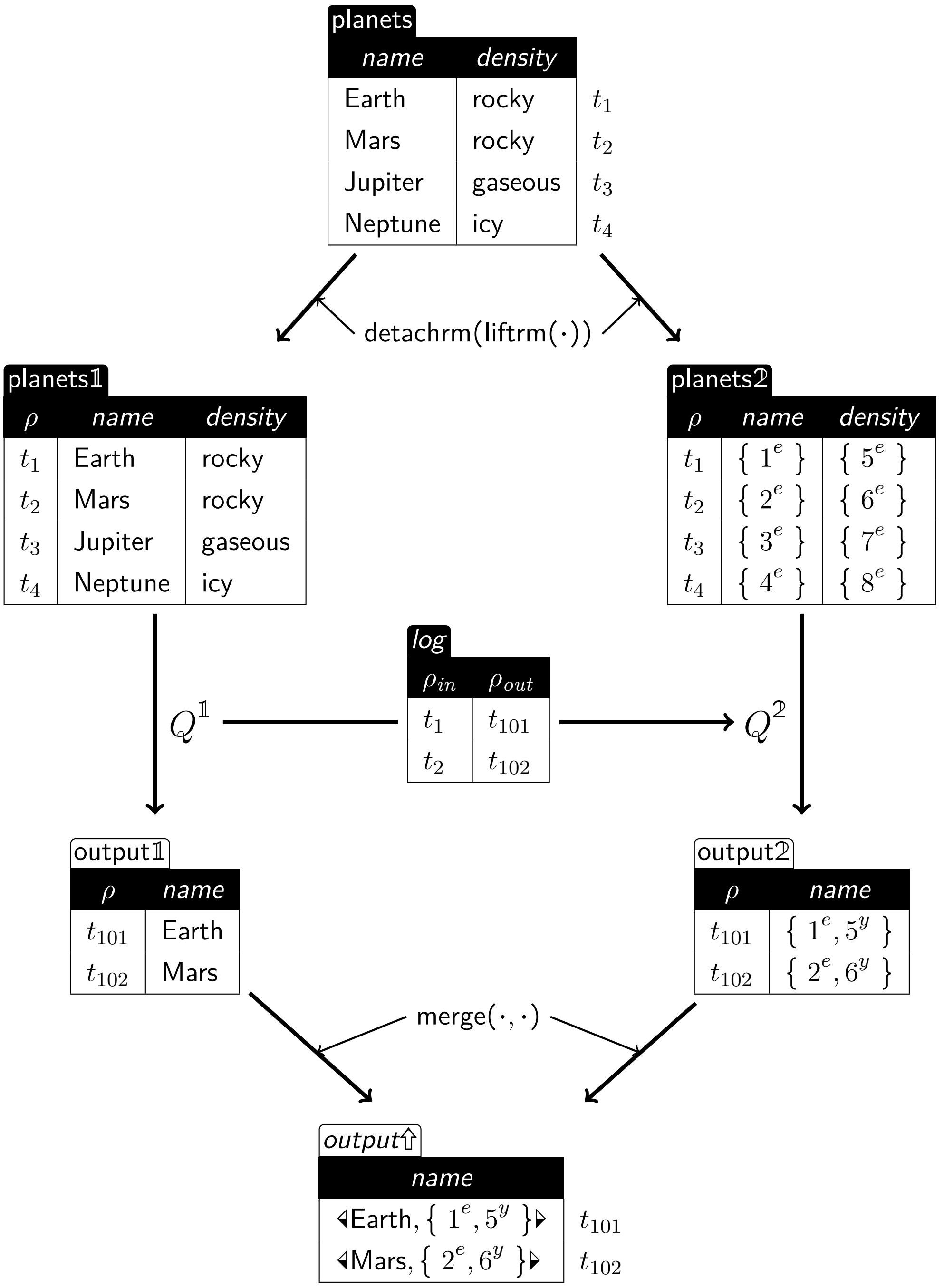
The very basic idea of the detached provenance analysis is to derive data provenance *detached* from actual values. We exemplify this for *output*⇧. Figure 8.4 contains the original table and its detached counterparts, denoted *output*𝟙 and *output*𝟚. Take note of the symmetry between *output*𝟙 and *output*𝟚: both tables have the same column names and cardinality but they have disjunct domains (i.e., *regular values* and *provenance annotations*). As another important feature, the tables share the same row identifiers ρ. These identifiers constitute actual columns in the detached approach. In comparison, *output*⇧ has the same annotations but in meta language. Adding the ρ columns is mandatory, because in plain bag semantics, there would be no deterministic way to reconstruct the matching pairs of value and provenance (e.g., ◂Earth, $\{11^e, 21^y\}$▸).

The bottom line of this example is that the two tables (*output*𝟙, *output*𝟚) and the single table *output*⇧ are equivalent and can be translated into each other.

8.2.2 *All* Tables Detached

Ultimately, we aim to detach *all* parts of the provenance analysis. Keeping the current example and zooming out a bit, the tables from Figure 8.5 appear. The figure exhibits the same diamond shape as Figure 8.1. The provenance computation is carried out in two (roughly independent) computation cycles. Phase 𝟙 evaluates $Q^{\mathbb{1}}$ and phase 𝟚 evaluates $Q^{\mathbb{2}}$. We are going to discuss the components of Figure 8.5 in turn.

For the discussion of *output*⇧, see above. The auxiliary function merge(·, ·) denotes the



planets

name	*density*	
Earth	rocky	t_1
Mars	rocky	t_2
Jupiter	gaseous	t_3
Neptune	icy	t_4

planets𝟙

ρ	*name*	*density*
t_1	Earth	rocky
t_2	Mars	rocky
t_3	Jupiter	gaseous
t_4	Neptune	icy

planets𝟚

ρ	*name*	*density*
t_1	$\{ 1^e \}$	$\{ 5^e \}$
t_2	$\{ 2^e \}$	$\{ 6^e \}$
t_3	$\{ 3^e \}$	$\{ 7^e \}$
t_4	$\{ 4^e \}$	$\{ 8^e \}$

log

ρ_{in}	ρ_{out}
t_1	t_{101}
t_2	t_{102}

output𝟙

ρ	*name*
t_{101}	Earth
t_{102}	Mars

output𝟚

ρ	*name*
t_{101}	$\{ 1^e, 5^y \}$
t_{102}	$\{ 2^e, 6^y \}$

output⇧

name	
◂Earth, $\{ 1^e, 5^y \}$▸	t_{101}
◂Mars, $\{ 2^e, 6^y \}$▸	t_{102}

Figure 8.5: **The tables involved in the analysis of the running example. Edge semantics according to their labels and the textual description.**

reintegration of values with provenance annotations.

In the upper part of the figure, the input table *planets* is processed using

$$\mathsf{detachrm}(\mathsf{liftrm}(\Delta))$$

. In this example, Δ contains the only table *planets*. $\mathsf{liftrm}(\cdot)$ generates the provenance annotations which are to be propagated towards the result table (formal introduction in Section 5.3.4). The auxiliary function $\mathsf{detach}\Delta(\cdot)$ detaches provenance annotations from values and yields the two tables *planets*$\mathbb{1}$ and *planets*$\mathbb{2}$.

In the center section of the figure, the queries $Q^{\mathbb{1}}$ and $Q^{\mathbb{2}}$ get evaluated (discussion of the query text is deferred). They consume and produce the accordingly named input and output tables. Both queries are the result from rewriting the input query Q according to the rewrite rules we present in the body of this chapter.

The table *log* gets conveyed from $Q^{\mathbb{1}}$ to $Q^{\mathbb{2}}$. This can be considered an integral overhead of the detached analysis approach. Since the table *planets*$\mathbb{2}$ only deals with provenance sets, $Q^{\mathbb{2}}$ cannot evaluate `WHERE` predicates (e.g., `p.density='rocky'`) to booleans. In order to compensate for this deficiency, the row identifiers of qualified rows get recorded in $Q^{\mathbb{1}}$ and are made available to $Q^{\mathbb{2}}$. In the example, this is the responsibility of table *log*. We call this process *logging*. Only through logging, both tables *output*$\mathbb{1}$ and *output*$\mathbb{2}$ can maintain a synchronized row count. In order to carry out the final $\mathsf{merge}(\cdot,\cdot)$ step and reconnect the corresponding rows with each other, this kind of synchronization is mandatory.

8.2.3 Phase $\mathbb{1}$

Evaluating $Q^{\mathbb{1}}$ is (due to logging) a non–optional preprocessing step before the actual provenance analysis (carried out through $Q^{\mathbb{2}}$) can take place. $Q^{\mathbb{1}}$, Q and the relevant tables are listed in Figure 8.6. The resemblance between $Q^{\mathbb{1}}$ and Q is very high by intention. Both queries are supposed to compute (nearly) the same result table, but $Q^{\mathbb{1}}$ carries along ρ columns and uses logging to record relevant ρ values and provide them to $Q^{\mathbb{2}}$. Hence, $Q^{\mathbb{1}}$

- references table *planets*$\mathbb{1}$ (which contains all values of *planets*, augmented with a ρ column) and
- yields another additional ρ column in its output table (see Figure 8.6(a) on line 8). As motivated above, the ρ identifier gets logged using the (side–effecting) UDF `writeLog(p.`ρ`)`.

```
       FROM planets𝟙
              LATERAL () AS p
      WHERE p.density='rocky'
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT writeLog(p.ρ) AS ρ,
            p.name AS name
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

(a) **Query $Q^{\mathbb{1}}$ (generated from Q).**

```
       FROM planets
              LATERAL () AS p
      WHERE p.density='rocky'
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT
            p.name AS name
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

(b) **Query Q (for reference).**

planets𝟙

ρ	*name*	*density*
t_1	Earth	rocky
t_2	Mars	rocky
t_3	Jupiter	gaseous
t_4	Neptune	icy

(c) **Input table.**

output𝟙

ρ	*name*
t_{101}	Earth
t_{102}	Mars

(d) **Output table.**

log

ρ_{in}	ρ_{out}
t_1	t_{101}
t_2	t_{102}

(e) **Logging table.**

Figure 8.6: $Q^{\mathbb{1}}$ **and the involved tables.**

```
       FROM planets𝟚
              LATERAL () AS p,
            readlog(p.ρ)
              LATERAL (p) AS l
      WHERE TRUE
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT l.ρ AS ρ,
            p.name
              ∪ toY(p.density ∪ ∅)
              AS name
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

(a) **Query $Q^{\mathbb{2}}$ (generated from Q).**

```
       FROM planets
            LATERAL () AS p


      WHERE p.density=’rocky’
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT
            p.name

              AS name
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

(b) **Query Q (for reference).**

planets𝟚

ρ	*name*	*density*
t_1	$\{\ 1_e\ \}$	$\{\ 5_e\ \}$
t_2	$\{\ 2_e\ \}$	$\{\ 6_e\ \}$
t_3	$\{\ 3_e\ \}$	$\{\ 7_e\ \}$
t_4	$\{\ 4_e\ \}$	$\{\ 8_e\ \}$

(c) **Input table.**

output𝟚

ρ	*name*
t_{101}	$\{\ 1_e, 5_y\ \}$
t_{102}	$\{\ 2_e, 6_y\ \}$

(d) **Output table.**

log

ρ_{in}	ρ_{out}
t_1	t_{101}
t_2	t_{102}

(e) **Logging table.**

Figure 8.7: $Q^{\mathbb{2}}$ **and the involved tables.**

The produced logging table is listed in Figure 8.6(e). It contains one row for each tuple that has survived the WHERE clause. Also, ρ identifiers are exchanged with fresh ones (this is important for n–way joins, discussed later). In the scope of this work, all logs are relational.

8.2.4 Phase 𝟚

$Q^{\mathbb{2}}$, Q and the relevant tables are listed in Figure 8.7.

Logging In general, `WHERE` e^{cell} cannot be evaluated in phase $\mathbb{2}$, because the predicate e^{cell} would yield data provenance instead of a boolean value. If we fed this to a DBMS, a (static) type error could be expected.

Looking at line 5 of $Q^{\mathbb{2}}$, the `WHERE` predicate is gone and has been replaced with a type–correct `TRUE` value. As a replacement, `readlog(p.ρ)` (lines 3-4) now implements the filtering semantics. The SQL code of the table–valued `readlog(p.ρ)` UDF stays abstract for now. Semantically, it accesses the logging table (produced in phase $\mathbb{1}$, listed in Figure 8.7(e)) and looks up the current `p.ρ` identifier. For example, t_1 is sitting in *log*, i.e. the corresponding tuple has passed the `WHERE` clause (in phase $\mathbb{1}$). The new identifier ρ_{out} is bound to `l.ρ` and used from this point on.

As another example, t_3 is not included in *log*. This lets `readlog(p.ρ)` produce an empty table, i.e. the current binding of `p` has no join partner and gets dropped.

In conclusion, logging enforces symmetry between the two phases and therefore enables us to run phase $\mathbb{2}$ values–less.

Provenance Annotations Going from phase $\mathbb{1}$ to phase $\mathbb{2}$, every data value becomes a provenance annotation. The (initial) annotations of base tables (like *planets*$\mathbb{2}$) have been created through application of $\mathsf{liftrm}(\cdot)$. Through evaluation of $Q^{\mathbb{2}}$, these initial annotations get propagated towards the result table. Any annotation arriving in the result table is part of the data provenance of query Q.

With exception of log inspection, $Q^{\mathbb{2}}$ operates exclusively in the domain of provenance annotations. Looking at $Q^{\mathbb{2}}$ in Figure 8.7(a),

- the initial provenance annotations from *planets*$\mathbb{2}$ are accessed (line 1),
- the rows (i.e. annotations) get assigned to tuple variable `p` (line 2),
- some rows get filtered through logging (line 3),
- the annotations sitting in `p.name` are referenced (line 11)
- Why–provenance (elaborated below) is added (line 12) and
- the result column name is determined (line 13).

In the end, $Q^{\mathbb{2}}$ emits table *output*$\mathbb{2}$ (see Figure 8.7(d)) which contains the final provenance annotations.

Why–Provenance According to the provenance definition for the `WHERE`–clause (see Section 6.3.4.2), the associated predicate yields Why–provenance. This provenance is then added (using $\cup$) to the individual output row.

$Q^{\mathbb{2}}$ implements just this provenance semantics. The original `WHERE`–predicate

- `p.density='rocky'` (line 5 in Figure 8.7(a)) becomes
- `p.density` $\cup\ \varnothing$ (line 12 in Figure 8.7(a))

as a result of query rewriting .

8.3 Formalization

8.3.1 Row Identifiers ρ

In the detached provenance analysis, each row is augmented with an additional column from the domain of row identifiers. The meta variable ρ denotes row identifiers.

Definition 8.1: Row Identifier

A **row identifier** has the type RID and the domain $\mathsf{dom}(\text{RID})$.
The auxiliary function $\mathsf{fresh}^{\rho}()$ creates new and unique row identifiers.

The purpose of row identifiers is twofold.

- The output table of phase $\mathbb{1}$ and the output table of phase $\mathbb{2}$ get merged together, based on their ρ columns. For an example, see tables *output1*, *output2* and *output*from Figure 8.5.
- Through logging, results of predicates get communicated between phases 1 and 2. For an example, see table *log* in Figure 8.5. Row identifiers map log entries to the row currently being processed.

For both applications, row identifiers must be unique per (intermediate) table. Below, we formulate an according correctness condition.

Condition 1

Row identifiers ρ are unique per (intermediate) table.

Condition 1 is referred to as **C1**.

```
1 (FROM ... AS a
2  ...
3      (FROM ... AS b
4       WHERE ... a.x+b.x ...
5       ...
6       SELECT writeFilter(a.ρ, b.ρ) ...
7       )
8  ...)
```

Figure 8.8: **Example: a correlated subquery in pseudocode.**

8.3.2 Logging and Nested Loops

SQL is a language which makes heavy use of nested loops. As an example, see Figure 8.8 which sketches a correlated subquery. The variable a is correlated (referenced on line 4). In general, the result of the example WHERE predicate depends on all combinations of a and b. If a concrete combination of rows qualifies against the WHERE predicate, the log writing on line 6 is carried out. The log would be ambiguous if a.ρ were not included in the log. For example, there could be a combination (a.ρ_1, b.ρ_1) which gets filtered and another combination (a.ρ_2, b.ρ_1) which does not get filtered. Therefore, the log must be able to distinguish between all combinations of tuple variables.

Condition 2

Log writing/reading includes the row identifiers ρ of free variables.

It is generally not desirable to include all visible variables in the log. Doing so would force query optimizers to design plans with additional loops and thus, a major impact on query performance could be expected.

A subexpression traversal can be employed to identify the free variables. The normalization step (see Section 7.3) takes care of this issue for the SFW expression. For non–SFW expressions, we discuss the correctness in the according rewrite rule. Condition 2 is referred to as **C2**.

8.3.3 Logging Locations ℓ and UDFs

Within a single query, multiple expressions may require logging. We use unique identifiers to distinguish the different logging locations. The auxiliary function

$$\boldsymbol{\ell} \coloneqq \mathsf{fresh}^{log}()$$

generates unique identifiers $\boldsymbol{\ell}$ which denote a certain logging location.

8.3.4 RM𝟙

We are going to formalize RM𝟙 which is the relational model of phase 𝟙.

Definition 8.2: RM𝟙 Types

An **RM𝟙 type** is a substitution of $\tau^{\text{BASE}\mathbb{1}}$, $\tau^{\text{CELL}\mathbb{1}}$, $\tau^{\text{ROW}\mathbb{1}}$, $\tau^{\text{TABLE}\mathbb{1}}$ or $\tau^{\text{DB}\mathbb{1}}$. The substitution rules listed in Figure 8.9(a) are applied recursively until all non–terminals are replaced.
The column identifiers col_i in $\text{ROW}(\cdot)$ are pairwise different and ordered.
The super types $\overline{\text{BASE}\mathbb{1}}$, $\overline{\text{CELL}\mathbb{1}}$, $\overline{\text{ROW}\mathbb{1}}$, $\overline{\text{TABLE}\mathbb{1}}$ and $\overline{\text{DB}\mathbb{1}}$ consist of all types which can be substituted starting from $\tau^{\text{BASE}\mathbb{1}}$, $\tau^{\text{CELL}\mathbb{1}}$, $\tau^{\text{ROW}\mathbb{1}}$, $\tau^{\text{TABLE}\mathbb{1}}$ and $\tau^{\text{DB}\mathbb{1}}$, respectively.

The only difference between Definition 8.2 and the corresponding RM definition (see Definition 4.1) is that rows have additional row identifiers. The base domains are equivalent to Definition 4.2. Also, the domains for arrays $\mathsf{dom}(\text{ARRAY}\mathbb{1}(\tau))$, tables $\mathsf{dom}(\text{TABLE}\mathbb{1}(\tau))$ and databases are nearly equivalent with exception of nested row values (which include row identifiers). The value domain $\mathsf{dom}(\text{ROW}\mathbb{1}(\cdot))$ gets formalized next.

Definition 8.3: Row Domains in RM𝟙

Let $\text{ROW}\mathbb{1}(\rho \mapsto \text{RID}, col_1 \mapsto \tau_1,\ \dots\ col_n \mapsto \tau_n)$ be a type according to Figure 8.9(a). The **row domain** is

$$\begin{aligned}&\mathsf{dom}(\text{ROW}\mathbb{1}(\rho \mapsto \text{RID}, col_1 \mapsto \tau_1, \dots col_n \mapsto \tau_n)) \coloneqq \\ &\qquad\qquad \{\langle v_0, v_1, \dots v_n\rangle \mid v_0 \in \mathsf{dom}(\text{RID}), v_1 \in \mathsf{dom}(\tau_1), \dots v_n \in \mathsf{dom}(\tau_n)\}\end{aligned}$$

.

RM𝟙 has minimal requirements (i.e., an additional column) if implemented using an off–the–shelf DBMS.

8.3.5 RM𝟚

The relational model of phase 𝟚 is called RM𝟚. Phase 𝟚 does not process regular values but propagates provenance annotations.

Definition 8.4: RM𝟚 Types

An **RM𝟚 type** is a substitution of $\tau^{\text{BASE}\mathbb{2}}$, $\tau^{\text{CELL}\mathbb{2}}$, $\tau^{\text{ROW}\mathbb{2}}$, $\tau^{\text{TABLE}\mathbb{2}}$ or $\tau^{\text{DB}\mathbb{2}}$. The substitution rules listed in Figure 8.9(b) are applied recursively until all non–terminals are replaced.
The column identifiers col_i in $\text{ROW}(\cdot)$ are pairwise different and ordered.
The super types $\overline{\text{BASE}\mathbb{2}}$, $\overline{\text{CELL}\mathbb{2}}$, $\overline{\text{ROW}\mathbb{2}}$, $\overline{\text{TABLE}\mathbb{2}}$ and $\overline{\text{DB}\mathbb{2}}$ consist of all types which can be substituted starting from $\tau^{\text{BASE}\mathbb{2}}$, $\tau^{\text{CELL}\mathbb{2}}$, $\tau^{\text{ROW}\mathbb{2}}$, $\tau^{\text{TABLE}\mathbb{2}}$ and $\tau^{\text{DB}\mathbb{2}}$, respectively.

Directly comparing the type definitions of Figure 8.9, the essential difference is that base types (BOOL, INT, DEC and TEXT) get replaced with provenance annotations (PSET) in RM𝟚. It is noteworthy that RM𝟚 only has one type of provenance annotation while RM𝟙 may support hundreds of base types. Besides of that, RM𝟙 and RM𝟚 are analogous by design. Especially the augmented row value types (having ρ as their first column) are consistent between both definitions. The value domains for rows, arrays, tables and databases are analogous to phase 𝟙 and are omitted. The PSET domain is according to Definition 5.3.

8.3.6 SQL

The detached provenance analysis is based on rewrites of SQL queries, i.e. the query language stays (nearly) the same. Figure 8.10 provides an overview of the rewrite steps. Query Q (with UDF definitions F) is supposed to be analyzed for its data provenance. The scope of this work starts with Q formulated in the backend SQL dialect. In a first rewrite step, Q gets normalized into Q^{norm}. The normalization is subject of Chapter 7 and is considered done. The main rewrite

$$\begin{aligned}\mathsf{detach}(\cdot) &:: \text{ETABLE} \rightarrowtail (\text{ETABLE}, \text{ETABLE})\\ \mathsf{detach}(Q^{norm}) &= (Q^{\mathbb{1}}, Q^{\mathbb{2}})\end{aligned}$$

is subject of this chapter. The formalization of $\mathsf{detach}(\cdot)$ is carried out next. Through evaluation of the two queries $Q^{\mathbb{1}}$ and $Q^{\mathbb{2}}$, the data provenance is computed.

$$
\begin{aligned}
\tau^{\text{BASE}\mathbb{1}} &::= \text{BOOL} \\
&\mid \text{INT} \\
&\mid \text{DEC} \\
&\mid \text{TEXT} \\
\tau^{\text{CELL}\mathbb{1}} &::= \tau^{\text{BASE}\mathbb{1}} \\
&\mid \tau^{\text{ROW}\mathbb{1}} \\
&\mid \text{ARRAY}\mathbb{1}(\ \tau^{\text{CELL}\mathbb{1}}\ ) \\
\tau^{\text{ROW}\mathbb{1}} &::= \text{ROW}\mathbb{1}(\ \rho \mapsto \text{RID}, \\
&\qquad\qquad\quad \mathit{col} \mapsto \tau^{\text{CELL}\mathbb{1}}, \\
&\qquad\qquad\quad \vdots \\
&\qquad\qquad\ ) \\
\tau^{\text{TABLE}\mathbb{1}} &::= \text{TABLE}\mathbb{1}(\ \tau^{\text{ROW}\mathbb{1}}\ ) \\
\tau^{\text{DB}\mathbb{1}} &::= \{\!\!\{\, \mathit{tab} \mapsto \tau^{\text{TABLE}\mathbb{1}}, \ldots \}\!\!\}
\end{aligned}
$$

(a) **RM𝟙 types.**

$$
\begin{aligned}
\tau^{\text{BASE}\mathbb{2}} &::= \text{PSET} \\
\tau^{\text{CELL}\mathbb{2}} &::= \tau^{\text{BASE}\mathbb{2}} \\
&\mid \tau^{\text{ROW}\mathbb{2}} \\
&\mid \text{ARRAY}\mathbb{2}(\ \tau^{\text{CELL}\mathbb{2}}\ ) \\
\tau^{\text{ROW}\mathbb{2}} &::= \text{ROW}\mathbb{2}(\ \rho \mapsto \text{RID}, \\
&\qquad\qquad\quad \mathit{col} \mapsto \tau^{\text{CELL}\mathbb{2}}, \\
&\qquad\qquad\quad \vdots \\
&\qquad\qquad\ ) \\
\tau^{\text{TABLE}\mathbb{2}} &::= \text{TABLE}\mathbb{2}(\ \tau^{\text{ROW}\mathbb{2}}\ ) \\
\tau^{\text{DB}\mathbb{2}} &::= \{\!\!\{\, \mathit{tab} \mapsto \tau^{\text{TABLE}\mathbb{2}}, \ldots \}\!\!\}
\end{aligned}
$$

(b) **RM𝟚 types.**

Figure 8.9: **Substitution rules used in definitions.**

$$\begin{tikzcd}
& (Q, F) \arrow[d, "\mathsf{normalize}(\cdot,\cdot)"] & \\
& Q^{norm} \arrow[dl] \arrow[dr] \arrow[d, phantom, "\mathsf{detach}(\cdot)"] & \\
Q^{\mathbb{1}} & & Q^{\mathbb{2}}
\end{tikzcd}$$

Figure 8.10: **Overview of query rewrites.**

8.3.7 Detached Provenance Analysis

Definition 8.5: Detached Provenance Analysis

Let Q :: $\textsc{etable}$ be a query formulated in the backend SQL dialect with the corresponding UDF definitions F :: $\overline{\textsc{udfs}}$. Let Δ :: $\overline{\textsc{db}}$ be a database instance. Let P :: $\overline{\textsc{table}\Uparrow}$ be the data provenance according to $\mathsf{Prov}(\mathsf{liftrm}(\Delta), Q, F) = P$.

The **detached provenance analysis** evaluates P with

- $\begin{array}{l} \mathsf{normalize}(\cdot,\cdot) :: (\textsc{etable}, \overline{\textsc{udfs}}) \rightarrow \textsc{etable} \\ \mathsf{normalize}(Q,F) = Q^{norm} \end{array}$,
- $\begin{array}{l} \mathsf{detachrm}(\mathsf{liftrm}(\cdot)) :: \overline{\textsc{db}} \rightarrow (\overline{\textsc{db}\mathbb{1}}, \overline{\textsc{db}\mathbb{2}}) \\ \mathsf{detachrm}(\mathsf{liftrm}(\Delta)) = (\Delta^{\mathbb{1}}, \Delta^{\mathbb{2}}) \end{array}$,
- $\begin{array}{l} \mathsf{detach}(\cdot) :: \textsc{etable} \rightarrow (\textsc{etable}, \textsc{etable}) \\ \mathsf{detach}(Q^{norm}) = (Q^{\mathbb{1}}, Q^{\mathbb{2}}) \end{array}$,
- $\begin{array}{l} \mathsf{Phase}\mathbb{1}(\cdot,\cdot,\cdot) :: (\overline{\textsc{db}\mathbb{1}}, \textsc{etable}, \overline{\textsc{udfs}^{*}}) \rightarrow (\overline{\textsc{table}\mathbb{1}}, \overline{\textsc{db}}) \\ \mathsf{Phase}\mathbb{1}(\Delta^{\mathbb{1}}, Q^{\mathbb{1}}, F_{log\mathbb{1}}) = (v^{\mathbb{1}}, \Delta_{log}) \end{array}$,
- $\begin{array}{l} \mathsf{Phase}\mathbb{2}(\cdot,\cdot,\cdot,\cdot) :: (\overline{\textsc{db}\mathbb{2}}, \textsc{etable}, \overline{\textsc{udfs}}, \overline{\textsc{db}}) \rightarrow \overline{\textsc{table}\mathbb{2}} \\ \mathsf{Phase}\mathbb{2}(\Delta^{\mathbb{2}}, Q^{\mathbb{2}}, F_{log\mathbb{2}}, \Delta_{log}) = v^{\mathbb{2}} \end{array}$ and
- $\begin{array}{l} \mathsf{merge}(\cdot,\cdot) :: (\overline{\textsc{table}\mathbb{1}}, \overline{\textsc{table}\mathbb{2}}) \rightarrow \overline{\textsc{table}\Uparrow} \\ \mathsf{merge}(v^{\mathbb{1}}, v^{\mathbb{2}}) = P \end{array}$.

Normalization $\mathsf{normalize}(Q,F) = Q^{norm}$ denotes the query normalization according to Definition 7.1. Q^{norm} is the normalized query with the inlined UDF definitions F.

Database $\mathsf{detachrm}(\mathsf{liftrm}(\Delta)) = (\Delta^{\mathbb{1}}, \Delta^{\mathbb{2}})$ denotes the preparation step for the user database Δ. First, $\mathsf{liftrm}(\Delta) = \Delta^{\Uparrow}$ according to Definition 5.7 is carried out. Thereby, the LRM database $\Delta^{\Uparrow}$ :: $\overline{\textsc{db}\Uparrow}$ is created which integrates both data values and initial provenance annotations (see discussion in Section 5.2.2). Next, $\mathsf{detachrm}(\Delta^{\Uparrow})$ separates the two domains of i) regular values and ii) provenance annotations, thereby creating databases ($\Delta^{\mathbb{1}}$ :: $\overline{\textsc{db}\mathbb{1}}, \Delta^{\mathbb{2}}$ :: $\overline{\textsc{db}\mathbb{2}}$) which are specific for the corresponding phase. Table names get suffixed with an according $\mathbb{1}$ or $\mathbb{2}$. For example, table *planets* would become *planets*$\mathbb{1}$ and *planets*$\mathbb{2}$, exemplified in Figure 8.5. We omit the formalization of $\mathsf{detachrm}(\cdot)$.

Query Rewrite $\mathsf{detach}(Q^{norm}) = (Q^{\mathbb{1}}, Q^{\mathbb{2}})$ denotes the application of the main rewrite rules. These rules get specified in the body of this chapter.

Phase $\mathbb{1}$ $\mathsf{Phase}\mathbb{1}(\Delta^{\mathbb{1}}, Q^{\mathbb{1}}, F_{log\mathbb{1}}) = (v^{\mathbb{1}}, \Delta_{log})$ denotes the value–only query evaluation which is to be carried out by an off–the–shelf DBMS. Its arguments are

- the database instance $\Delta^{\mathbb{1}}$ (discussed above),
- SQL query $Q^{\mathbb{1}}$ (discussed above) and
- the logging UDFs $F_{log\mathbb{1}} :: \overline{\text{UDFS}*}$.

The UDFs type is annotated with * in order to point out an additional requirement: these UDFs write to the logging database Δ_{log}. Accordingly, they cannot be specified in the SQL backend dialect.

Phase $\mathbb{2}$ $\mathsf{Phase}\mathbb{2}(\Delta^{\mathbb{2}}, Q^{\mathbb{2}}, F_{log\mathbb{2}}, \Delta_{log}) = v^{\mathbb{2}}$ is similar to phase $\mathbb{1}$ but consumes the logs Δ_{log} instead. The log replaces the (now unnecessary) evaluation of predicates. Therefore, phase $\mathbb{2}$ can focus on propagation of provenance annotations. The other inputs to phase $\mathbb{2}$ are $\Delta^{\mathbb{2}}$ (user database substituted with initial provenance annotations), $Q^{\mathbb{2}}$ (the rewritten query) and $F_{log\mathbb{2}}$ (the UDFs responsible for log reading, corresponding directly to the UDFs in $F_{log\mathbb{1}}$). $F_{log\mathbb{2}}$ is simpler and can be expressed in the backend dialect. Finally, $v^{\mathbb{2}}$ is the result table with provenance annotations instead of values. As an example, see table *output*$\mathbb{2}$ from Figure 8.5.

Merge $\mathsf{merge}(v^{\mathbb{1}}, v^{\mathbb{2}}) = P$ carries out the re–combination of the domains of values and provenance annotations. Both input tables $v^{\mathbb{1}}$ and $v^{\mathbb{2}}$ may have a different tuple order. Joining them on column ρ yields the correct pairs of value and data provenance. This step is exemplified in Figure 8.5 (without formalization).

8.4 Provenance Annotation Operators

For provenance propagation in the LRM context (see Chapter 5), we employed the following operations.

- $\cup$ and $\bigcup$ evaluate the union of two provenance annotations.
- $\mathsf{Y}(\cdot)$ turns any provenance identifier into Why–provenance.
- $\curlyvee(\cdot)$ computes the deep union of all provenance annotations in a nested data structure.

- $\Psi(\cdot,\cdot)$ recursively adds a provenance annotation to a nested data structure.

For the detached provenance analysis, the operators $\cup$, $\bigcup$ and $\mathsf{Y}(\cdot)$ are kept without modification. The recursive operators require definitions specific to phases $\mathbb{1}$ and $\mathbb{2}$.

8.4.1 Definitions

Definition 8.6: Collect$\mathbb{2}$ Operator

Let $v :: \tau$ with τ one of the types $\overline{\textsc{base}\mathbb{2}}$, $\overline{\textsc{array}\mathbb{2}}$, $\overline{\textsc{row}\mathbb{2}}$ or $\overline{\textsc{table}\mathbb{2}}$.
The **collect$\mathbb{2}$ operator** is

$$\curlyvee^{2}(v) \coloneqq \begin{cases} \begin{array}{l} \textbf{let } \mathbb{p} \coloneqq v \\ \ \textbf{in } \mathbb{p} \end{array} & , v :: \overline{\textsc{base}\mathbb{2}} \\[1em] \begin{array}{l} \textbf{let } [v_1,\ldots] \coloneqq v, \\ \qquad \mathbb{p}_1 \coloneqq \curlyvee^{2}(v_1), \\ \qquad \vdots \\ \ \textbf{in } \bigcup_{i=1,\ldots} \mathbb{p}_i \end{array} & , v :: \overline{\textsc{array}\mathbb{2}} \\[1em] \begin{array}{l} \textbf{let } \langle \rho, v_1, \ldots v_n \rangle \coloneqq v, \\ \qquad v_1 \coloneqq \curlyvee^{2}(v_1), \\ \qquad \vdots \\ \ \textbf{in } \bigcup_{i=1,\ldots} \mathbb{p}_i \end{array} & , v :: \overline{\textsc{row}\mathbb{2}} \\[1em] \begin{array}{l} \textbf{let } \llparenthesis v_1, \ldots v_n \rrparenthesis \coloneqq v, \\ \qquad \mathbb{p}_1 \coloneqq \curlyvee^{2}(v_1), \\ \qquad \vdots \\ \ \textbf{in } \bigcup_{i=1,\ldots} \mathbb{p}_i \end{array} & , v :: \overline{\textsc{table}\mathbb{2}} \end{cases}$$

.

Semantically, $\curlyvee^{2}(v)$ evaluates to the deep union of all provenance annotations in v. Row identifiers ρ are skipped.

For example,

$$\curlyvee^{2}([\{1^{e}\},\{1^{e},2^{y}\}]) = \{1^{e},2^{y}\}$$

merges the two provenance annotations of an array.

Definition 8.7: Push2 Operator

Let $v :: \tau$ with τ one of the types $\overline{\text{BASE}\mathbb{2}}$, $\overline{\text{ARRAY}\mathbb{2}}$, $\overline{\text{ROW}\mathbb{2}}$ or $\overline{\text{TABLE}\mathbb{2}}$. Let $\mathbb{p}_+ :: \text{PSET}$ be a provenance annotation.
The **push2 operator** is

$$\Psi^2(v, \mathbb{p}_+) \coloneqq \begin{cases} \textbf{let } \mathbb{p} \coloneqq v \\ \ \textbf{in } \mathbb{p} \cup \mathbb{p}_+ & , v :: \overline{\text{BASE}\mathbb{2}} \\ \textbf{let } [v_1, \ldots] \coloneqq v \\ \ \textbf{in } [\Psi^2(v_1, \mathbb{p}_+), \ldots] & , v :: \overline{\text{ARRAY}\mathbb{2}} \\ \textbf{let } \langle \rho, v_1, \ldots \rangle \coloneqq v \\ \ \textbf{in } \langle \rho, \Psi^2(v_1, \mathbb{p}_+), \ldots \rangle & , v :: \overline{\text{ROW}\mathbb{2}} \\ \textbf{let } \llparenthesis v_1, \ldots \rrparenthesis \coloneqq v \\ \ \textbf{in } \llparenthesis \Psi^2(v_1, \mathbb{p}_+), \ldots \rrparenthesis & , v :: \overline{\text{TABLE}\mathbb{2}} \end{cases}$$

.

Semantically, $\Psi^2(v, \mathbb{p})$ recursively adds the provenance annotation $\mathbb{p}$ to all existing provenance annotations in v. The operator preserves the data type of v, i.e.

$$\Psi^2(\cdot, \cdot) :: (\tau, \text{PSET}) \rightarrowtail \tau$$

as well as the cardinalities of arrays and tables. Moreover, the row identifiers ρ are retained.

For example,

$$\Psi^2([\{1^e\}, \{1^e, 2^e\}], \{3^y\}) = [\{1^e, 3^y\}, \{1^e, 2^e, 3^y\}]$$

adds 3^y to both provenance annotations of an array.

Definition 8.8: Collect𝟙 Operator

Let $v :: \tau$ with τ one of the types $\overline{\textsc{base}\mathbb{1}}$, $\overline{\textsc{array}\mathbb{1}}$ or $\overline{\textsc{row}\mathbb{1}}$.
The **collect𝟙 operator** is

$$\curlyvee^{\mathbb{1}}(v) \coloneqq \begin{cases} v & , v :: \overline{\textsc{base}\mathbb{1}} \\ \textbf{let } [v_1, \ldots] \coloneqq v \\ \ \textbf{in } [\curlyvee^{\mathbb{1}}(v_1), \ldots] & , v :: \overline{\textsc{array}\mathbb{1}} \\ \textbf{let } \langle \rho, v_1, \ldots v_n \rangle \coloneqq v, \\ \ \textbf{in } \langle \curlyvee^{\mathbb{1}}(v_1), \ldots \curlyvee^{\mathbb{1}}(v_n) \rangle & , v :: \overline{\textsc{row}\mathbb{1}} \end{cases}$$

.

Semantically, $\curlyvee^{\mathbb{1}}(v)$ recursively throws away the row identifiers ρ (and according columns) of v. The function type is

$$\curlyvee^{\mathbb{1}}(\cdot) :: \tau \rightarrowtail \tau'$$

with $\tau \neq \tau'$ in the general case. As the main feature, τ' is an RM type according to Definition 4.1, designed to be compatible with common SQL dialects and query engines. Implementations of sort algorithms can process values of type τ' without being misguided by any row identifiers.

For example,

$$\curlyvee^{\mathbb{1}}(\langle \rho \mapsto 42, \mathit{foo} \mapsto \mathsf{payload} \rangle) = \langle \mathit{foo} \mapsto \mathsf{payload} \rangle$$

drops the row identifier 42 and the row type changes accordingly.

8.5 Rewrite of Cell Expressions

Cell expressions get rewritten according to the rules provided below. Each rewrite rule has its corresponding provenance definition in Chapter 6. Conclusions are denoted

$$\boxed{e^{cell} \Mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}, e^{cell,\mathbb{2}})}$$

where

- $e^{cell} :: \textsc{ecell}$ is the expression to be rewritten,

- $e^{cell,\mathbb{1}}$ :: ECELL is the resulting expression for phase $\mathbb{1}$ and
- $e^{cell,\mathbb{2}}$ :: ECELL is the resulting expression for phase $\mathbb{2}$.

The rewrite formalism demands two environments which we keep implicit. The two auxiliary functions

- $\mathsf{fresh}^{log}()$ :: LOGID and
- $\mathsf{fresh}^{\rho}()$ :: RID.

create unique identifiers for logging locations and rows, respectively. Order does not matter.

8.5.1 Literals ℓ

$$\frac{\textsc{detach-literal}}{\ell \Mapsto^{cell}_{detach} (\ell, \varnothing)}$$

Rule DETACH-LITERAL keeps the literal expressions ℓ for phase $\mathbb{1}$. For phase $\mathbb{2}$ however, the resulting expression is the empty annotation–constructor $\varnothing$.

Comparison with the Definitional Interpreter In Chapter 6, we employed a definitional interpreter and defined the semantics for SQL expressions. Below, rule DEF-LITERAL is reproduced.

$$\frac{\textsc{def-literal}}{\Gamma^{\Uparrow}; \Delta^{\Uparrow}; F \vdash \ell \Mapsto^{cell}_{def} \blacktriangleleft \ell, \varnothing \blacktriangleright}$$

Both rules (DETACH-LITERAL and DEF-LITERAL) distinguish between value and provenance in a similar fashion and ultimately yield the same semantics. However, the definitional interpreter yields values (i.e., $\varnothing$ :: PSET) while the rewrite rules yield expressions (i.e., $\varnothing$ :: ECELL). In the latter case, the expressions are supposed to be evaluated by the query processor of a DBMS.

8.5.2 Row Constructors ROW

$$\textsc{detach-row}\quad \frac{\begin{array}{c} \Big|\, e_i^{cell} \Mapsto^{cell}_{detach} (e_i^{cell,\mathbb{1}}, e_i^{cell,\mathbb{2}}) \,\Big|_{i=1..n} \\ \rho := \mathsf{fresh}^{\rho}() \\ e^{cell,\mathbb{1}} := \mathtt{ROW}(\rho, e_1^{cell,\mathbb{1}}, \ldots e_n^{cell,\mathbb{1}}) \\ e^{cell,\mathbb{2}} := \mathtt{ROW}(\rho, e_1^{cell,\mathbb{2}}, \ldots e_n^{cell,\mathbb{2}}) \end{array}}{\mathtt{ROW}(e_1^{cell}, \ldots e_n^{cell}) \Mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}, e^{cell,\mathbb{2}})}$$

Rule DETACH-ROW uses recursion to detach the n sub–expressions first. Then, a value–only and a provenance–only row expression gets assembled. A unique ρ identifier is generated which populates the (equally named) ρ attribute. Both SQL expressions share the same row identifier.

Correctness The `ROW` expression may get evaluated in a loop (at runtime). However, row identifiers $\rho \coloneqq \mathsf{fresh}^{\rho}()$ are generated statically. We are going to discuss the two correctness conditions from Section 8.3.

- **C1** cannot be violated since the row constructor does not create tables.
- **C2** cannot be violated since log writing/reading does not happen.

8.5.3 Column References $.\,col$

$$\textsc{detach-col}\quad \dfrac{e^{cell} \Mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}, e^{cell,\mathbb{2}})}{e^{cell}.\,col \Mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}.\,col, e^{cell,\mathbb{2}}.\,col)}$$

Rule DETACH-COLUMN is perfectly symmetric in phases $\mathbb{1}$ and $\mathbb{2}$. Both phases share the same column names and on evaluation, the corresponding value or provenance is retrieved.

8.5.4 Comparison of Row Values

$$\textsc{detach-rowop}\quad \dfrac{\begin{array}{c} e_a^{cell} \Mapsto^{cell}_{detach} (e_a^{cell,\mathbb{1}}, e_a^{cell,\mathbb{2}}) \qquad e_b^{cell} \Mapsto^{cell}_{detach} (e_b^{cell,\mathbb{1}}, e_b^{cell,\mathbb{2}}) \\ e^{cell,\mathbb{1}} \coloneqq \curlyvee^{\mathbb{1}}(e_a^{cell,\mathbb{1}}) \circledast_{row} \curlyvee^{\mathbb{1}}(e_b^{cell,\mathbb{1}}) \\ e^{cell,\mathbb{2}} \coloneqq \curlyvee^{\mathbb{2}}(e_a^{cell,\mathbb{2}}) \cup \curlyvee^{\mathbb{2}}(e_b^{cell,\mathbb{2}}) \end{array}}{e_a^{cell} \circledast_{row} e_b^{cell} \Mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}, e^{cell,\mathbb{2}})}$$

The abstract operator $\circledast_{row}$ compares two row values with each other. In our detached approach, row values get generally augmented with an additional ρ attribute (this happens in the recursive rewrites of e_a^{cell} and e_b^{cell}). However, this attribute would change the query semantics if it were included in the comparison. $\curlyvee^{\mathbb{1}}(\cdot)$ (see Definition 8.8) recursively removes this attribute in phase $\mathbb{1}$. In phase $\mathbb{2}$, all provenance annotations of the two rows get merged together (eager semantics).

8.5.5 Generic Operators

$$\begin{array}{l}\text{DETACH-OPERATOR}\\ \dfrac{\left|\; e_i^{cell} \Mapsto_{detach}^{cell} (e_i^{cell,\mathbb{1}}, e_i^{cell,\mathbb{2}}) \;\right|_{i=1..n}}{\circledast(\; e_1^{cell},\ \dots\ e_n^{cell}\;) \Mapsto_{detach}^{cell} \left(\circledast(\; e_1^{cell,\mathbb{1}},\ \dots\ e_n^{cell,\mathbb{1}}\;),\ \bigcup\limits_{i=1..n} e_i^{cell,\mathbb{2}}\right)}\end{array}$$

Compared to $\circledast_{row}$ (see above), the operands of $\circledast$ always evaluate to unnested (i.e., $\overline{\text{BASE}}$) values. The rewrite for phase $\mathbb{1}$ is trivial and the rewrite for phase $\mathbb{2}$ basically replaces $\circledast_{row}$ with $\bigcup$.

8.5.6 `ISNULL`

$$\begin{array}{l}\text{DETACH-ISNULL}\\ \dfrac{e^{cell} \Mapsto_{detach}^{cell} (e^{cell,\mathbb{1}}, e^{cell,\mathbb{2}})}{\texttt{ISNULL}(\; e^{cell}\;) \Mapsto_{detach}^{cell} (\texttt{ISNULL}(\; e^{cell,\mathbb{1}}\;), e^{cell,\mathbb{2}})}\end{array}$$

The rewrite of `ISNULL` is straightforward.

8.5.7 Variable Reference *var*

$$\begin{array}{l}\text{DETACH-VAR}\\ var \Mapsto_{detach}^{cell} (var, var)\end{array}$$

A variable reference *var* yields the corresponding value sitting in the current variable environment $\Gamma^{\mathbb{1}}$ (in phase $\mathbb{1}$) and $\Gamma^{\mathbb{2}}$ (in phase $\mathbb{2}$). Our rewrite rules are static and do not formalize the corresponding environments for the two phases (or during type checking). At runtime, their types would be

- $\Gamma^{\mathbb{1}} :: \{\!\!\{\, var \mapsto \overline{\text{CELL}\mathbb{1}}\,\}\!\!\}$ and
- $\Gamma^{\mathbb{2}} :: \{\!\!\{\, var \mapsto \overline{\text{CELL}\mathbb{2}}\,\}\!\!\}$.

Due to the specific environments, *var* yields a value or provenance annotation when evaluated in phase $\mathbb{1}$ or $\mathbb{2}$.

8.5.8 Array Constructor `ARRAY`

$$\begin{array}{l}\text{DETACH-ARRAY}\\ \dfrac{\begin{array}{l}\left|\; e_i^{cell} \Mapsto_{detach}^{cell} (e_i^{cell,\mathbb{1}}, e_i^{cell,\mathbb{2}}) \;\right|_{i=1..n}\\ e^{cell,\mathbb{1}} := \texttt{ARRAY}(\; e_1^{cell,\mathbb{1}}, \dots e_n^{cell,\mathbb{1}}\;)\\ e^{cell,\mathbb{2}} := \texttt{ARRAY}(\; e_1^{cell,\mathbb{2}}, \dots e_n^{cell,\mathbb{2}}\;)\end{array}}{\texttt{ARRAY}(\; e_1^{cell}, \dots e_n^{cell}\;) \Mapsto_{detach}^{cell} (e^{cell,\mathbb{1}}, e^{cell,\mathbb{2}})}\end{array}$$

The rewrite of array expressions is very similar to row expressions (rule DETACH-ROW) but simpler (because there is no row identifier).

8.5.9 Array Length

$$\textsc{detach-array-length}$$
$$\texttt{LENGTH(}\ e^{cell}\ \texttt{)} \Rrightarrow^{cell}_{detach} (\texttt{LENGTH(}\ e^{cell}\ \texttt{)}, \varnothing)$$

Rule DETACH-ARRAY-LENGTH directly corresponds to the definition DEF-ARRAY-LENGTH in Section 6.1.8.1. Our arguments regarding to the empty provenance annotation ($\varnothing$) can be found there.

8.5.10 CASE

Figure 8.11 lists the rewrite rule DETACH-CASE. We make use of the related EQCASE expression in the rewrite (introduced in Section 6.1.10).

Logging CASE is the first expression which involves logging. As a reminder, the main advantage and challenge of the detached provenance analysis is that phase 𝟚 is free of data values. However, the CASE expression uses predicates which *must* yield booleans, or the CASE fails to select a branch. The solution to this issue is logging. We employ the two UDFs `writecase(·)` and `readcase(·)` which can dynamically communicate the branch selection from phase 𝟙 to phase 𝟚.

`writecase`$(\ell, \rho, branch)$ is a side–effecting UDF with three arguments. This UDF is generic enough to cover all CASE expressions of a query. Because of its side–effect, it cannot be formulated in the backend dialect. We are going to specify the log (=a table) instead. Example implementations of some log–writing UDFs are provided in the experiments context (see Section 9.1.3). The arguments are

- ℓ which statically identifies the current CASE expression,
- ρ which uniquely identifies the current tuple being processed and
- *branch* which encodes the selected WHEN... branch. We employ integer constants to identify the n–th branch and 0 for the ELSE case.

logcase is the table containing the branch decisions of CASE expressions (schema listed below). Its three columns directly correspond to the three arguments of `writecase(·)` which implements an unconditional insert. The primary key stretches over ℓ and ρ, because for any concrete combination of them, there can only be one branch. null values are not possible.

DETACH-CASE

$$
\begin{array}{c}
\boldsymbol{\ell} \coloneqq \mathsf{fresh}^{log}() \\
\left|\; e^{cell}_{when,i} \Mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}_{when,i}, e^{cell,\mathbb{2}}_{when,i}) \;\right|_{i=1\ldots n} \\
\left|\; e^{cell}_{then,i} \Mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}_{then,i}, e^{cell,\mathbb{2}}_{then,i}) \;\right|_{i=1\ldots n} \qquad e^{cell}_{else} \Mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}_{else}, e^{cell,\mathbb{2}}_{else}) \\
e^{cell,\mathbb{1}} \coloneqq \begin{array}{l}
\texttt{EQCASE} \\
\quad \texttt{writecase}(\boldsymbol{\ell}, \texttt{v}.\rho, e^{cell}_{when}) \\
\quad \texttt{WHEN 1 THEN } e^{cell,\mathbb{1}}_{then,1} \\
\quad \vdots \\
\quad \texttt{WHEN } n \texttt{ THEN } e^{cell,\mathbb{1}}_{then,n} \\
\quad \texttt{ELSE } e^{cell,\mathbb{1}}_{else} \\
\texttt{END}
\end{array}
\qquad
e^{cell}_{when} \coloneqq \begin{array}{l}
\texttt{CASE} \\
\quad \texttt{WHEN } e^{cell,\mathbb{1}}_{when,1} \texttt{ THEN 1} \\
\quad \vdots \\
\quad \texttt{WHEN } e^{cell,\mathbb{1}}_{when,n} \texttt{ THEN } n \\
\quad \texttt{ELSE 0} \\
\texttt{END}
\end{array} \\
e^{cell,\mathbb{2}} \coloneqq \begin{array}{l}
\texttt{EQCASE} \\
\quad \texttt{readcase}(\boldsymbol{\ell}, \texttt{v}.\rho) \\
\quad \texttt{WHEN 1 THEN } \Psi^{\mathbb{2}}(e^{cell,\mathbb{2}}_{then,1}, \Upsilon(\bigcup_{i=1..1} e^{cell,\mathbb{2}}_{when,i})) \\
\quad \vdots \\
\quad \texttt{WHEN } n \texttt{ THEN } \Psi^{\mathbb{2}}(e^{cell,\mathbb{2}}_{then,n}, \Upsilon(\bigcup_{i=1..n} e^{cell,\mathbb{2}}_{when,i})) \\
\quad \texttt{ELSE } \Psi^{\mathbb{2}}(e^{cell,\mathbb{2}}_{else}, \Upsilon(\bigcup_{i=1..n} e^{cell,\mathbb{2}}_{when,i})) \\
\texttt{END}
\end{array} \\
\hline
\begin{array}{l}
\texttt{CASE} \\
\quad \texttt{WHEN } e^{cell}_{when,1} \texttt{ THEN } e^{cell}_{then,1} \\
\quad \vdots \\
\quad \texttt{WHEN } e^{cell}_{when,n} \texttt{ THEN } e^{cell}_{then,n} \\
\quad \texttt{ELSE } e^{cell}_{else} \\
\texttt{END}
\end{array}
\Mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}, e^{cell,\mathbb{2}})
\end{array}
$$

Figure 8.11: **Rewrite of `CASE`.**

logcase

$\underline{\ell}$	$\underline{\rho}$	*branch*

The UDF `readcase(`ℓ`,`ρ`)` is the log–reading counterpart, called in phase 𝟚. Providing the attributes for the primary key, the *branch* is looked up and returned as result. An empty result would indicate a serious error.

Rewrite Rule The rewrite rule DETACH-CASE is provided in Figure 8.11. The rule employs

- `CASE` to evaluate `WHEN` branches (phase 𝟙 only) and
- `EQCASE` to evaluate `THEN` branches (both phases).

(A solution without `EQCASE` can be realized as well but requires to distribute the logging UDFs over all n predicates. We omit the details.)

On the bottom of the rule, the n input branches and the mandatory `ELSE` branch are provided. On the very top of the rule, an identifier for the current `CASE` expression is generated and used in the logging UDFs. Phase 𝟙 uses a sub–`CASE` to evaluate the `WHEN` predicates and yields a simple integer–based identifier to uniquely identify the branch having qualified. The outer `EQCASE` transparently logs this very identifier (see discussion above) and uses it to pick the according branch for evaluation of the corresponding `THEN` expression.

In phase 𝟚, the inner `CASE` does not exist. Instead, the log is consulted and the according `THEN` branch gets evaluated. Due to predicates, Why–provenance is created (see Section 6.1.9 and rule DEF-CASE for a discussion of the provenance semantics).

(Non)–Compositionality and Correctness **C1** cannot be violated since `CASE` does not yield tables.

The argument for **C2** is more complex. In our formalization, rule DETACH-CASE references the row identifier `v.`ρ in its logging calls. However, **C2** clearly states that correlated tuple variables must be included in the log. At first sight, this looks like a violation of **C2**. However, due to normalization (see Section 7.4.3) and according to rewrite rule DETACH-JOIN (introduced later in Section 8.7.2), identifiers `v.`ρ get composed from all correlated tuple variables. Through transitivity, **C2** holds. The drawback of this approach is that `CASE` violates the compositionality of the SQL dialect. The `CASE` expression is restricted to those expression contexts where `v` exists. These are all SFW clauses except for FROM and WHERE. For example, the TPC-H benchmark contains queries with CASE in the SELECT clause.

8.5.11 TOROW

$$\frac{e^{table} \Mapsto^{table}_{detach} (e^{table,\mathbb{1}}, e^{table,\mathbb{2}})}{\texttt{TOROW}(e^{table}) \Mapsto^{cell}_{detach} (\texttt{TOROW}(e^{table,\mathbb{1}}), \texttt{TOROW}(e^{table,\mathbb{2}}))}\ \textsc{detach-torow}$$

Rule DETACH-TOROW looks simple at first sight, but has interesting details.

Let the nested table expression e^{table} evaluate to n rows.

- For $n \geq 2$, the query fails (in both phases).
- For $n = 1$, TOROW(·) yields *the* row with all of its attributes. In both phases, this includes the ρ attribute. This additional column is not an issue because the expression yields multiple columns in general. Any other value or provenance annotation sitting in that row just stays there and is part of the result.
- For $n = 0$, TOROW(·) generates a type–correct null value. I.e., a single row is produced with all attributes set to null. Rows and arrays can be nested arbitrarily deep. (In our SQL dialect, the helper makenull(·) is utilized, discussed in Section 6.1.11.) That being said, what is the semantics of null in phase 𝟙, phase 𝟚 and as a row identifier ρ?
  - In phase 𝟙, null values are retained.
  - In phase 𝟚, null values are implicitly turned into the empty provenance annotation ($\varnothing$).
  - As a row identifier, null may be kept or converted into 0 (for compatibility with primary keys in logging tables). As long as the row count is $n \leq 1$, any identifier is a unique identifier. It only needs to be consistent between the two phases.

8.5.12 EXISTS

$$\frac{e^{table} \Mapsto^{table}_{detach} (e^{table,\mathbb{1}}, e^{table,\mathbb{2}})}{\texttt{EXISTS}(e^{table}) \Mapsto^{cell}_{detach} (\texttt{EXISTS}(e^{table,\mathbb{1}}), \curlyvee^{\mathbb{2}}(e^{table,\mathbb{2}}))}\ \textsc{detach-exists}$$

Rule DETACH-EXISTS first detaches the table expression e^{table} using the according rewrite rules (table expression are subject of the next section). For phase 𝟙, the EXISTS can be retained. For phase $\mathbb{2}$, the entire tabular result of $e^{table,\mathbb{2}}$ is merged and yields a single (big) provenance annotation. In contrast, the rule DEF-EXISTS defines a single row to be included in the provenance result. Why is there a mismatch between definition and rewrite rule?

We observed from the experiments that the DBMS used lazy evaluation in phase $\mathbb{1}$. Typically, the **EXISTS** expressions sits behind a **WHERE** predicate. Hence, the first row passing the **WHERE** predicate is included in the log, the rest is ignored. Through the log, this lazy evaluation is transferred to phase $\mathbb{2}$ and becomes lazy provenance semantics. There is however no guarantee. If $e^{table,\mathbb{2}}$ were a base table, the provenance of the entire table table would be collected. A more elaborate rewrite rule may employ a LIMIT clause in order to consistently yield a single row in phase $\mathbb{2}$. We consider this future work.

8.6 Rewrite of Table Expressions

Table expressions get rewritten according to the rules provided below. Each rewrite rule has its corresponding provenance definition in Chapter 6. Conclusions are denoted

$$\boxed{e^{table} \Mapsto^{table}_{detach} (e^{table,\mathbb{1}}, e^{table,\mathbb{2}})}$$

where

- e^{table} :: ETABLE is the expression to be rewritten,
- $e^{table,\mathbb{1}}$ :: ETABLE is the resulting expression for phase $\mathbb{1}$ and
- $e^{table,\mathbb{2}}$ :: ETABLE is the resulting expression for phase $\mathbb{2}$.

8.6.1 Table References *tab*

$$\begin{array}{l}\textsc{detach-tab}\\ tab \Mapsto^{table}_{detach} (tab\mathbb{1}, tab\mathbb{2})\end{array}$$

Rule DETACH-TABLE is a mere renaming of referenced tables. The suffixes ($\mathbb{1}$ and $\mathbb{2}$) match those created by $\mathsf{detachrm}(\cdot)$.

8.6.2 VALUES

$$\begin{array}{l}\textsc{detach-values}\\ \dfrac{\begin{array}{c} \left| e_i^{cell} \Mapsto^{cell}_{detach} (e_i^{cell,\mathbb{1}}, e_i^{cell,\mathbb{2}}) \right|_{i=0..n} \\ e^{table,\mathbb{1}} := \texttt{VALUES}(e_1^{cell,\mathbb{1}}, \dots e_n^{cell,\mathbb{1}}) \\ e^{table,\mathbb{2}} := \texttt{VALUES}(e_1^{cell,\mathbb{2}}, \dots e_n^{cell,\mathbb{2}}) \end{array}}{\texttt{VALUES}(e_1^{cell}, \dots e_n^{cell}) \Mapsto^{table}_{detach} (e^{table,\mathbb{1}}, e^{table,\mathbb{2}})}\end{array}$$

stars𝟙

ρ	*name*
42	Sol

(a) **Input table.**

```
stars𝟙
UNION ALL
stars𝟙
```

(b) **Naive query rewrite.**

output𝟙

ρ	*name*
42	Sol
↯	Sol

(c) **Output table.**

Figure 8.12: **Example query: `UNION ALL` in phase 𝟙.**

The rule DETACH-VALUES is analogous to DETACH-ARRAY but on table level.

Correctness **C2** holds because no logging is carried out.

In order to satisfy **C1**, we restrict the $e_{\square}^{cell}$ to row expressions `ROW`. The `ROW` expression statically enforces fresh ρ values for each expression, thereby **C1** is satisfied.

8.6.3 `WITH`

$$\textsc{detach-with}\quad \frac{\left|\ e_i^{table} \mapsto_{detach}^{table} \left(e_i^{table,\mathbb{1}}, e_i^{table,\mathbb{2}}\right) \right|_{i=0..n} \qquad \begin{array}{l} e^{table,\mathbb{1}} := \texttt{WITH}\ (tab_1\mathbb{1}\ \texttt{AS}\ e_1^{table,\mathbb{1}}, \ldots tab_n\mathbb{1}\ \texttt{AS}\ e_n^{table,\mathbb{1}})\ e_0^{table,\mathbb{1}} \\ e^{table,\mathbb{2}} := \texttt{WITH}\ (tab_1\mathbb{2}\ \texttt{AS}\ e_1^{table,\mathbb{2}}, \ldots tab_n\mathbb{2}\ \texttt{AS}\ e_n^{table,\mathbb{2}})\ e_0^{table,\mathbb{2}} \end{array}}{\texttt{WITH}\ (tab_1\ \texttt{AS}\ e_1^{table}, \ldots tab_n\ \texttt{AS}\ e_n^{table})\ e_0^{table} \mapsto_{detach}^{table} \left(e^{table,\mathbb{1}}, e^{table,\mathbb{2}}\right)}$$

Rule DETACH-WITH uses the same renaming pattern as DETACH-TAB.

8.6.4 `UNION ALL`

The e_l^{table} `UNION ALL` e_r^{table} expression merges the result tables of its two subexpressions. All rows including duplicates are retained. This semantics may violate **C1**, as exemplified in Figure 8.12. In table *output𝟙*, a collision of row identifiers $\rho = 42$ occurs.

Rewrite rule DETACH-UNION-ALL of Figure 8.13 employs logging to avoid such conflicts. The idea is that any row identifier of e_r^{table} gets substituted with a fresh (i.e., globally unique) one. Depending on cardinalities, e_l^{table} can be a better choice (not covered in our work).

The schema is

logunion

$\underline{\ell}$	$\underline{\rho_{in}}$	$\underline{\rho_{f,1}}$	$\underline{\cdots}$	$\underline{\rho_{f,m}}$	ρ_{out}

DETACH-UNION-ALL

$$
\dfrac{
\begin{array}{c}
e_l^{table} \Mapsto^{table}_{detach} \left(e_l^{table,\mathbb{1}}, e_l^{table,\mathbb{2}}\right) \qquad e_r^{table} \Mapsto^{table}_{detach} \left(e_r^{table,\mathbb{1}}, e_r^{table,\mathbb{2}}\right) \\[1ex]
(col_1, \ldots col_n) \coloneqq \mathsf{colnames}(e_r^{table}) \qquad (f_1, \ldots f_m) \coloneqq \mathsf{freevariables}(e_r^{table}) \\[1ex]
e_{r*}^{table,\mathbb{1}} \;\coloneqq\; \begin{array}{rl}
\mathtt{FROM} & e_r^{table,\mathbb{1}}\ \mathtt{LATERAL\ ()\ AS\ v} \\
\mathtt{WHERE} & \mathtt{TRUE} \\
\mathtt{GROUP\ BY} & \mathtt{()} \\
\mathtt{AGGREGATES} & \mathtt{()} \\
\mathtt{HAVING} & \mathtt{TRUE} \\
\mathtt{WINDOWS} & \mathtt{()} \\
\mathtt{SELECT} & \mathtt{writeUnion(}\boldsymbol{\ell}\mathtt{,\ v.}\rho\mathtt{,\ }f_1\mathtt{.}\rho\mathtt{,\ }\ldots f_m\mathtt{.}\rho\mathtt{)\ AS\ }\rho\mathtt{,} \\
 & \mathtt{v.}col_1\ \mathtt{AS}\ col_1\mathtt{,} \\
 & \vdots \\
 & \mathtt{v.}col_n\ \mathtt{AS}\ col_n \\
\mathtt{ORDER\ BY} & \mathtt{()} \\
\mathtt{DISTINCT\ ON} & \mathtt{()} \\
\mathtt{OFFMIT} & \mathtt{NULL\ NULL}
\end{array} \\[1ex]
e_{r*}^{table,\mathbb{2}} \;\coloneqq\; \begin{array}{rl}
\mathtt{FROM} & e_r^{table,\mathbb{2}}\ \mathtt{LATERAL\ ()\ AS\ v} \\
\mathtt{WHERE} & \mathtt{TRUE} \\
\mathtt{GROUP\ BY} & \mathtt{()} \\
\mathtt{AGGREGATES} & \mathtt{()} \\
\mathtt{HAVING} & \mathtt{TRUE} \\
\mathtt{WINDOWS} & \mathtt{()} \\
\mathtt{SELECT} & \mathtt{readUnion(}\boldsymbol{\ell}\mathtt{,\ v.}\rho\mathtt{,\ }f_1\mathtt{.}\rho\mathtt{,\ }\ldots f_m\mathtt{.}\rho\mathtt{)\ AS\ }\rho\mathtt{,} \\
 & \mathtt{v.}col_1\ \mathtt{AS}\ col_1\mathtt{,} \\
 & \vdots \\
 & \mathtt{v.}col_n\ \mathtt{AS}\ col_n \\
\mathtt{ORDER\ BY} & \mathtt{()} \\
\mathtt{DISTINCT\ ON} & \mathtt{()} \\
\mathtt{OFFMIT} & \mathtt{NULL\ NULL}
\end{array} \\[1ex]
e^{table,\mathbb{1}} \coloneqq e_l^{table,\mathbb{1}}\ \mathtt{UNION\ ALL}\ e_{r*}^{table,\mathbb{1}} \qquad e^{table,\mathbb{2}} \coloneqq e_l^{table,\mathbb{2}}\ \mathtt{UNION\ ALL}\ e_{r*}^{table,\mathbb{2}}
\end{array}
}{
e_l^{table}\ \mathtt{UNION\ ALL}\ e_r^{table} \Mapsto^{table}_{detach} \left(e^{table,\mathbb{1}}, e^{table,\mathbb{2}}\right)
}
$$

Figure 8.13: **Rewrite of UNION ALL.**

where

- $\boldsymbol{\ell}$ is the log identifier,
- ρ_{in} is the row identifier of the input row,
- $\rho_{f,1}, \dots \rho_{f,m}$ are the row identifiers of correlated tuple variables and
- ρ_{out} is a fresh and unique row identifier.

The auxiliary function $\mathsf{colnames}(e^{table}_{\square})$ yields the list of column names in table expression $e^{table}_{\square}$ (not formalized).

Correctness Through the logging procedure formalized above, **C1** is satisfied. **C2** is satisfied through including the row identifiers of all correlated (i.e., free) variables. The auxiliary function $\mathsf{freevariables}(\cdot)$ carries out the search for free variables (not formalized).

8.7 Rewrite of the SFW Expression

The query normalization (described in Chapter 7) yields five types of SFW expressions which can be distinguished through simple pattern matching. In this section, we provide the five corresponding rewrite rules.

8.7.1 Normalized `SELECT`

The rewrite rule DETACH-SELECT from Figure 8.14 has a single entry in its `FROM` clause (due to normalization) and specifies n columns in its `SELECT` clause. The rewrite adds the ρ column to the result columns and goes into recursion for all expressions in the `SELECT` clause.

8.7.2 Normalized Join

Rule DETACH-JOIN of Figure 8.15 has a non–trivial `WHERE` predicate and multiple entries in its `FROM` clause. In the detached approach, this rule requires logging for two reasons. First, the `WHERE`–predicate filters away certain combinations of join partners. Second, joining n rows rows together requires a consistent strategy for how to merge their n individual row identifiers $\rho_{\square}$. Our approach is to generate fresh row identifiers at runtime and to record the mapping between the old identifiers and the new one. For $n = 2$, the schema of the log is

DETACH-SELECT

$$
\begin{array}{c}
e^{table}_{from} \Mapsto^{table}_{detach} (e^{table,\mathbb{1}}_{from}, e^{table,\mathbb{2}}_{from}) \qquad \Big| \; e^{cell}_{i} \Mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}_{i}, e^{cell,\mathbb{2}}_{i}) \Big|_{i=1..n} \\
e^{table,\mathbb{1}} := \begin{array}{rl}
\texttt{FROM} & e^{table,\mathbb{1}}_{from} \; \texttt{LATERAL ()} \; \texttt{AS} \; \texttt{v} \\
\texttt{WHERE} & \texttt{TRUE} \\
\texttt{GROUP BY} & \texttt{()} \\
\texttt{AGGREGATES} & \texttt{()} \\
\texttt{HAVING} & \texttt{TRUE} \\
\texttt{WINDOWS} & \texttt{()} \\
\texttt{SELECT} & \texttt{v.}\rho \; \texttt{AS} \; \rho, \\
 & e^{cell,\mathbb{1}}_{1} \; \texttt{AS} \; col_1, \ldots \; e^{cell,\mathbb{1}}_{n} \; \texttt{AS} \; col_n \\
\texttt{ORDER BY} & \texttt{()} \\
\texttt{DISTINCT ON} & \texttt{()} \\
\texttt{OFFMIT} & \texttt{NULL NULL}
\end{array}
\qquad
e^{table,\mathbb{2}} := \begin{array}{rl}
\texttt{FROM} & e^{table,\mathbb{2}}_{from} \; \texttt{LATERAL ()} \; \texttt{AS} \; \texttt{v} \\
\texttt{WHERE} & \texttt{TRUE} \\
\texttt{GROUP BY} & \texttt{()} \\
\texttt{AGGREGATES} & \texttt{()} \\
\texttt{HAVING} & \texttt{TRUE} \\
\texttt{WINDOWS} & \texttt{()} \\
\texttt{SELECT} & \texttt{v.}\rho \; \texttt{AS} \; \rho, \\
 & e^{cell,\mathbb{2}}_{1} \; \texttt{AS} \; col_1, \ldots \; e^{cell,\mathbb{2}}_{n} \; \texttt{AS} \; col_n \\
\texttt{ORDER BY} & \texttt{()} \\
\texttt{DISTINCT ON} & \texttt{()} \\
\texttt{OFFMIT} & \texttt{NULL NULL}
\end{array} \\
\hline
\begin{array}{rl}
\texttt{FROM} & e^{table}_{from} \; \texttt{LATERAL ()} \; \texttt{AS} \; \texttt{v} \\
\texttt{WHERE} & \texttt{TRUE} \\
\texttt{GROUP BY} & \texttt{()} \\
\texttt{AGGREGATES} & \texttt{()} \\
\texttt{HAVING} & \texttt{TRUE} \\
\texttt{WINDOWS} & \texttt{()} \\
\texttt{SELECT} & e^{cell}_{1} \; \texttt{AS} \; col_1, \ldots \; e^{cell}_{n} \; \texttt{AS} \; col_n \\
\texttt{ORDER BY} & \texttt{()} \\
\texttt{DISTINCT ON} & \texttt{()} \\
\texttt{OFFMIT} & \texttt{NULL NULL}
\end{array}
\Mapsto^{table}_{detach} (e^{table,\mathbb{1}}, e^{table,\mathbb{2}})
\end{array}
$$

Figure 8.14: **Rewrite rule: normalized SELECT.**

DETACH-JOIN

$$\ell \coloneqq \mathsf{fresh}^{log}()$$

$$\left| \; e_i^{table} \Mapsto^{table}_{detach} \left(e_i^{table,\mathbb{1}}, e_i^{table,\mathbb{2}}\right) \right|_{i=1..n} \qquad e^{cell} \Mapsto^{cell}_{detach} \left(e^{cell,\mathbb{1}}, e^{cell,\mathbb{2}}\right)$$

$e^{table,\mathbb{1}} \coloneqq$

```
          FROM e_1^{table,𝟙} LATERAL vs_1 AS var_1,
               ⋮
               e_n^{table,𝟙} LATERAL vs_n AS var_n
         WHERE e^{cell,𝟙}
      GROUP BY ()
    AGGREGATES ()
        HAVING TRUE
       WINDOWS ()
        SELECT writejoin(ℓ, var_1.ρ, … var_n.ρ) AS ρ,
               var_{sel,1}.col_1 AS col_1, …
      ORDER BY ()
   DISTINCT ON ()
        OFFMIT NULL NULL
```

$e^{table,\mathbb{2}} \coloneqq$

```
          FROM e_1^{table,𝟚} LATERAL vs_1 AS var_1,
               ⋮
               e_n^{table,𝟚} LATERAL vs_n AS var_n,
               readjoin(ℓ, var_1.ρ, … var_n.ρ)
                      LATERAL (var_1, … var_n) AS 1
         WHERE TRUE
      GROUP BY ()
    AGGREGATES ()
        HAVING TRUE
       WINDOWS ()
        SELECT 1.ρ AS ρ,
               Ψ^𝟚(var_{sel,1}.col_1, Υ(e^{cell,𝟚})) AS col_{s,1}, …
      ORDER BY ()
   DISTINCT ON ()
        OFFMIT NULL NULL
```

```
          FROM e_1^{table} LATERAL vs_1 AS var_1,
               ⋮
               e_n^{table} LATERAL vs_n AS var_n
         WHERE e^{cell}
      GROUP BY ()
    AGGREGATES ()
        HAVING TRUE
       WINDOWS ()
        SELECT var_{sel,1}.col_{s,1} AS col_1, …
      ORDER BY ()
   DISTINCT ON ()
        OFFMIT NULL NULL
```

$$\Mapsto^{table}_{detach} \left(e^{table,\mathbb{1}}, e^{table,\mathbb{2}}\right)$$

Figure 8.15: **Rewrite rule: normalized join.**

```
       FROM logjoin2 LATERAL () AS l
      WHERE a0=ℓ AND a1=l.ρ1 AND a2=l.ρ2
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT l.ρout AS ρ
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

Figure 8.16: **UDF body of `readJoin(·)` for a 2–way join, using positional parameters (`a0`, `a1`, `...`).**

logjoin2

$\underline{\ell}$	$\underline{\rho_1}$	$\underline{\rho_2}$	ρ_{out}

. ρ_{out} is the resulting row identifier and ℓ the logging location. For $n > 2$, the schema grows accordingly.

Looking at $e^{table,\mathbb{1}}$ from DETACH-JOIN, the log–writing UDF is placed in the `SELECT` clause. Its call references all $\rho_\square$ values which in turn get written to the log. For each write, a new ρ_{out} is produced which in turn becomes the new ρ value in the result table. The remaining entries of the `SELECT` clause can be copied literally from the input expression (see rules DETACH-VAR and DETACH-COL which have trivial rewrites — other subexpressions do not appear in the `SELECT` clause due to normalization).

In phase $\mathbb{2}$ (see $e^{table,\mathbb{2}}$), the log–reading UDF is placed in the `FROM` clause. Using `LATERAL`, all other variables become visible to the UDF. An implementation of the log–reading function for 2–way joins is provided in Figure 8.16. It can only yield zero rows or a single row (due to the primary key of the log).

In the provenance definition (see DEF-WHERE), Why–provenance gets added to the current row. In DETACH-JOIN, this semantics is achieved through adding it to each individual result column (see line 12 of $e^{table,\mathbb{2}}$ in DETACH-JOIN).

Correctness **C1** is satisfied: for each output row, a fresh row identifier is generated dynamically. **C2** is satisfied: all correlated tuple variables are listed in the `FROM` expression (due to normalization, see Section 7.3).

8.7.3 Normalized `ORDER BY`

Rule DETACH-ORDER-BY (provided in Figure 8.17) rewrites the non–trivial `ORDER BY`, `DISTINCT ON` and `OFFMIT` clauses. The two latter ones depend on logging (i.e., synchronize the removal of rows between the two phases), however `ORDER BY` does not (i.e., the row ordering between phases $\mathbb{1}$ and $\mathbb{2}$ is generally different). Put in other words, rows in phase $\mathbb{2}$ can always stay unordered.

The query expression $e^{table,\mathbb{1}}$ of phase $\mathbb{1}$ (see DETACH-ORDER-BY) consists of two nested SFW expressions. The basic issue here is that logging must be carried out *after* the `OFFMIT` clause has been processed. Therefore, the inner expression carries out the sort operation, duplicate elimination and removes leading/trailing rows. The outer expression carries out the logging (i.e, any rows being left are considered qualified) and sorts again. We assume that query optimizers exploit the redundancy in the two `ORDER BY` clauses (they are identical) and the according query plan only triggers a single, physical sort operation. The schema of the log is

logfilter

$\underline{\ell}$	$\underline{\rho}$

with ℓ being the logging location and ρ the row identifier of a qualified row. Put in other words, our detached provenance analysis models the tripel (`ORDER BY`, `DISTINCT ON` and `OFFMIT`) as a filter operation. The log does not contain information about the row order.

In phase $\mathbb{2}$ (see $e^{table,\mathbb{2}}$ of DETACH-ORDER-BY), we employ `readfilter(·)` in an additional `FROM` entry to replay the filter semantics of `DISTINCT ON` and `OFFMIT`. The row order is not explicitly synchronized but implicitly through the (omnipresent) row identifiers which connect the rows of phase $\mathbb{2}$ to the ordered rows of phase $\mathbb{1}$. Why–provenance is generated on lines 9-10 from the subexpressions of `ORDER BY` and `DISTINCT ON`. The Why–provenance is added to all output columns.

For simpler SFW expressions (with trivial `DISTINCT ON` and `OFFMIT`), the logging can be skipped entirely. We do not formalize these corner cases.

Correctness **C1** is satisfied: the SFW expression discussed above exclusively removes rows. Therefore, ambiguous row identifiers cannot be added.

C2 is satisfied through transitivity: the `v.`ρ identifiers have been generated according to rule DETACH-JOIN with all (correlated) tuple variables taken into account.

8.7.4 Normalized `AGGREGATES`

DETACH-ORDER-BY

$$\boldsymbol{\ell} \coloneqq \mathsf{fresh}^{log}()$$

$$e^{table} \overset{table}{\underset{detach}{\Longmapsto}} (e^{table,\mathbb{1}}, e^{table,\mathbb{2}})$$

$e^{table,\mathbb{1}}$:=

```
FROM (    FROM $e^{table,\mathbb{1}}$ LATERAL () AS v
         WHERE TRUE
      GROUP BY ()
    AGGREGATES ()
        HAVING TRUE
       WINDOWS ()
        SELECT v.$\rho$ AS $\rho$,
               v.$col_{sel,1}$ AS $col_{sel,1}$, ...
      ORDER BY v.$col_{ord,1}$ $asc_1$, ...
   DISTINCT ON v.$col_{dis,1}$, ...
        OFFMIT $\ell_{off}$ $\ell_{lim}$) AS v
   WHERE TRUE
GROUP BY ()
AGGREGATES ()
  HAVING TRUE
 WINDOWS ()
  SELECT writefilter($\boldsymbol{\ell}$, v.$\rho$) AS $\rho$,
         v.$col_{sel,1}$ AS $col_{sel,1}$, ...
ORDER BY v.$col_{ord,1}$ $asc_1$, ...
DISTINCT ON ()
  OFFMIT NULL NULL
```

$e^{table,\mathbb{2}}$:=

```
    FROM $e^{table,\mathbb{2}}$ LATERAL () AS v,
         readfilter($\boldsymbol{\ell}$, v.$\rho$) LATERAL (v) AS l
   WHERE TRUE
GROUP BY ()
AGGREGATES ()
  HAVING TRUE
 WINDOWS ()
  SELECT l.$\rho$ AS $\rho$,
         $\Psi^{\mathbb{2}}$(v.$col_{sel,1}$,$\Upsilon$(v.$col_{ord,1}$ $\cup$ ...
                              v.$col_{dis,1}$ $\cup$ ...))
           AS $col_{sel,1}$,
         ⋮
ORDER BY ()
DISTINCT ON ()
  OFFMIT NULL NULL
```

```
     FROM $e^{table}$ LATERAL () AS v
    WHERE TRUE
 GROUP BY ()
AGGREGATES ()
   HAVING TRUE
  WINDOWS ()
   SELECT v.$col_{sel,1}$ AS $col_{sel,1}$, ...
 ORDER BY v.$col_{ord,1}$ $asc_1$, ...
DISTINCT ON v.$col_{dis,1}$, ...
   OFFMIT $\ell_{off}$ $\ell_{lim}$
```

$$\overset{table}{\underset{detach}{\Longmapsto}} (e^{table,\mathbb{1}}, e^{table,\mathbb{2}})$$

Figure 8.17: **Rewrite rule: normalized ORDER BY.**

DETACH-AGGREGATES

$$\ell \mathrel{:=} \mathsf{fresh}^{log}()$$

$$e^{table} \Longmapsto^{table}_{detach} (e^{table,\mathbb{1}}, e^{table,\mathbb{2}}) \quad \Big|\ e_i^{agg} \Longmapsto^{agg}_{detach} (e_i^{agg,\mathbb{1}}, e_i^{agg,\mathbb{2}}) \Big|_{i=1..n}$$

$$e_{hav}^{cell} \Longmapsto^{cell}_{detach} (e_{hav}^{cell,\mathbb{1}}, e_{hav}^{cell,\mathbb{2}}) \quad \Big|\ e_k^{cell} \Longmapsto^{cell}_{detach} (e_k^{cell,\mathbb{1}}, e_k^{cell,\mathbb{2}}) \Big|_{k=1..m}$$

$e^{table,\mathbb{1}} \mathrel{:=}$

```
       FROM $e^{table,\mathbb{1}}$ LATERAL () AS v
      WHERE TRUE
   GROUP BY $e_1^{cell,\mathbb{1}}$, ... $e_m^{cell,\mathbb{2}}$
 AGGREGATES array_agg(v.$\rho$) AS $\rho_{group}$,
            $e_1^{agg,\mathbb{1}}$ AS $col_1$, ... $e_n^{agg,\mathbb{1}}$ AS $col_n$
     HAVING $e_{hav}^{cell,\mathbb{1}}$
    WINDOWS ()
     SELECT writeagg($\ell$, aggs.$\rho_{group}$) AS $\rho$,
            aggs.$col_1$ AS $col_1$,
            ⋮
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

$e^{table,\mathbb{2}} \mathrel{:=}$

```
       FROM $e^{table,\mathbb{2}}$ LATERAL () AS v,
            readagg($\ell$, v.$\rho$) LATERAL (v) AS l
      WHERE TRUE
   GROUP BY l.$\rho$
 AGGREGATES the(l.$\rho$) AS $\rho$
            $e_1^{agg,\mathbb{2}}$ AS $col_1$, ... $e_n^{agg,\mathbb{2}}$ AS $col_n$
     HAVING TRUE
    WINDOWS ()
     SELECT aggs.$\rho$ AS $\rho$,
            $\Psi^{\mathbb{2}}$(aggs.$col_1$,$\Upsilon$($e_{hav}^{cell,\mathbb{2}} \cup \bigcup_{i=1}^{m} e_i^{cell,\mathbb{2}}$))
                AS $col_1$,
            ⋮
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

```
       FROM $e^{table}$ LATERAL () AS v
      WHERE TRUE
   GROUP BY $e_1^{cell}$, ... $e_m^{cell}$
 AGGREGATES $e_1^{agg}$ AS $col_1$, ... $e_n^{agg}$ AS $col_n$
     HAVING $e_{hav}^{cell}$
    WINDOWS ()
     SELECT aggs.$col_1$ AS $col_1$,
            ⋮
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

$$\Longmapsto^{table}_{detach} (e^{table,\mathbb{1}}, e^{table,\mathbb{2}})$$

Figure 8.18: **Rewrite rule: normalized AGGREGATES.**

```
       FROM logagg LATERAL () AS l
      WHERE a0=l.$\ell$ AND a1=l.$\rho_{in}$
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS ()
     SELECT l.$\rho_{out}$ AS $\rho$
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

Figure 8.19: **UDF body of `readagg(·)`.**

The rule DETACH-AGGREGATES is similar to DETACH-JOIN in the aspect of combining multiple ρ identifiers into a single one. Only in this case, the ρ values getting merged share the same group. The schema of the log is

logagg		
$\underline{\ell}$	$\underline{\rho_{in}}$	ρ_{out}

with ρ_{out} being identical for the ρ_{in} sharing the same group. The `HAVING` predicate can remove certain groups from the result. If a group gets filtered, the log does not contain the according entries.

Looking at the rewrite implementation in DETACH-AGGREGATES, the predicate `HAVING` e^{cell}_{hav} is handled similar to the `WHERE`–predicate in DETACH-JOIN (i.e., kept in phase $\mathbb{1}$ and turned into Why–provenance in $\mathbb{2}$).

We employ an additional aggregation function `ARRAY_AGG`(e^{cell}) (not introduced in the backend dialect) which has the semantics to evaluate e^{cell} for all rows and put the values in an array value. Using `ARRAY_AGG`, all row identifiers constituting the current group get collected (see line 4 of $e^{table,\mathbb{1}}$). The resulting array value gets bound to `aggs.`ρ and becomes an argument of `writeagg()` (on line 8) which implements the log write and returns a fresh (single) ρ to be the representative for the entire group. In parallel, all n user–specified aggregation functions ($e_1^{agg,\mathbb{1}}$) get evaluated.

In phase $\mathbb{2}$, the aggregation log is inspected in order to re–create the same groups from phase $\mathbb{1}$). (The UDF body for `readagg(a0, a1)` is listed in Figure 8.19.) Using `GROUP BY (l.`ρ`)`, the same groups from phase $\mathbb{1}$ get re–assembled. Then, the evaluation of the (rewritten) aggregate expressions is carried out. Hereby, the aggregation function `THE()` yields *the* shared identifier of the current group — which becomes the row identifier of the result row.

$$\textsc{detach-sum}\quad\frac{e^{cell} \Mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}, e^{cell,\mathbb{2}})}{\texttt{SUM}(e^{cell}) \Mapsto^{agg}_{detach} (\texttt{SUM}(e^{cell,\mathbb{1}}), \bigcup e^{cell,\mathbb{2}})}$$

$$\textsc{detach-countstar}\quad \texttt{COUNTSTAR}() \Mapsto^{agg}_{detach} (\texttt{COUNTSTAR}(), \varnothing)$$

Figure 8.20: **Rewrite of aggregations.**

Correctness **C1** is satisfied: the SFW expression discussed above creates fresh row identifiers.

C2 is satisfied through transitivity: the $\texttt{v.}\rho$ identifiers have been generated according to rule DETACH-JOIN with all (correlated) tuple variables taken into account.

8.7.4.1 Rewrite of Aggregation Functions

The rewrite of aggregations is formalized using conclusions

$$\boxed{e^{agg} \Mapsto^{agg}_{detach} (e^{agg,\mathbb{1}}, e^{agg,\mathbb{2}})}$$

where

- e^{agg} is the aggregation expression to be rewritten,
- $e^{agg,\mathbb{1}}$ is the rewritten expression for phase $\mathbb{1}$ and
- $e^{agg,\mathbb{2}}$ is the rewritten expression for phase $\mathbb{2}$.

The concrete rewrite rules are listed in Figure 8.20. Rule DETACH-SUM represents the generic case and covers AVG, MIN, MAX, COUNT and THE as well. DETACH-COUNTSTAR yields the empty data provenance. We keep this discussion short, since the rewrite rules obey the provenance definition for aggregates (see Section 6.5).

The provenance analysis of ARRAY_AGG is not supported. The issue is that the arrays created in phases $\mathbb{1}$ and $\mathbb{2}$ generally differ in their order of elements. Some SQL dialects support an additional ORDER BY specifier for aggregates, however our backend dialect does not.

8.7.5 Normalized WINDOWS

As introduced in Section 6.6, a window functions employs a sliding window. While the sliding window is being moved over the input table, an aggregation functions gets evaluated repeatedly. The detached provenance analysis can grasp the sliding window and its movement. Since input rows get partitioned and ordered (both depending on runtime values), logging is required.

8.7.5.1 Logging

We reuse the example query from Section 6.6.1 to introduce our approach to logging of window functions. An excerpt of the original query is listed in Figure 8.21(a) and the input table (for phase 𝟙) is listed in Figure 8.21(c). For now, we just focus on the definition of the window frame and what information is needed in the log. The resulting log (see Figure 8.21(d)) consists of four columns. The information stored there does not explicitly store the four window frames (that would require more data) but enough information to determine these frames.

- ℓ holds the logging identifier which is unique per window function in the query.
- ρ contains the row identifiers of the input table. These are not necessarily ordered. For each of those row identifiers, one window frame is to be determined.
- *part* stores an identifier denoting the partition of ρ. Shared identifiers means that the corresponding rows share the same partition. In our (deliberately simple)

```
OVER (PARTITION BY ()
          ORDER BY v.month
           ROWS 1 1        )
```

(a) **Example OVER clause.**

```
OVER (PARTITION BY l.part
          ORDER BY v.month ASC
              ROWS 1 1         )
```

(b) **Example OVER clause in phase 𝟚.**

sales𝟙

ρ	*month*	*units*
t_1	1	20
t_2	2	30
t_3	3	25
t_4	4	25

(c) **Phase 𝟙 input table.**

logwindow

$\underline{\ell}$	$\underline{\rho}$	*part*	*rank*
ℓ_1	t_1	t_1	1
ℓ_1	t_2	t_1	2
ℓ_1	t_3	t_1	3
ℓ_1	t_4	t_1	4

(d) **The resulting log.**

Figure 8.21: **Logging example for a window function.**

example, only one partition exists.

- *rank* determines the position of the row in its partition, after sorting.

What phase 𝟚 does is to reference columns *part* and *rank* directly in the corresponding `OVER` clause. Figure 8.21(b) lists the according query fragment. Input table and *logwindow* got joined on ρ and *logwindow* gets bound to variable `l` (`FROM` clause not listed). Hence,

- `PARTITION BY l.part` yields the same partitions,
- `ORDER BY l.rank ASC` yields the same order of rows and
- `ROWS 1 1` yields the same window width

in phase 𝟚.

8.7.5.2 Rewrite Rules

The rewrite rule DETACH-WINDOWS of Figures 8.22 and 8.23 is restricted to the rewrite of a single window function (for its size and complexity). If multiple window functions are to be rewritten, we are confident that an additional normalization step could be used (not formalized in this work). The idea of such normalization is based on the principle of window functions that they do not change the row count. Therefore, SFW expressions (each evaluating a single window function) can be nested into each other (using their `FROM` clauses). Each SFW expression would then evaluate one window function and add one additional column.

Phase 𝟙 The input expression of rule DETACH-WINDOWS contains a single window function `w1` (defined on lines 6-9). The basic idea of phase 𝟙 is to keep `w1` (see $e^{table,\mathbb{1}}$ on lines 6-9) and only rewrite its subexpressions. On top of that, we employ two additional window functions which are responsible for collecting the data for the log.

- `w2` uses the window function `FIRST_VALUE` which yields the first value of the current partition, ordered according to the `ORDER BY` clause. We employ this value (which is the same for all tuples of the same partition) as the partition identifier in the log (see column *part* in Figure 8.21(d)).
- `w3` uses the window function `ROW_NUMBER` which yields the (dense and ascending) number of the row in the current partition, ordered according to the `ORDER BY` clause. We store this value in the log (see column *rank* in Figure 8.21(d)) and use it as the sole sort criterion in phase 𝟚.

DETACH-WINDOWS

$$\ell := \mathsf{fresh}^{log}() \qquad e^{table}_{from} \mapsto^{table}_{detach} (e^{table,\mathbb{1}}_{from}, e^{table,\mathbb{2}}_{from})$$

$$e^{win} \mapsto^{win}_{detach} (e^{win,\mathbb{1}}, e^{win,\mathbb{2}})$$

$$\Big|\, e^{cell}_{par,k} \mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}_{par,k}, e^{cell,\mathbb{2}}_{par,k}) \,\Big|_{k=1..m} \qquad \Big|\, e^{cell}_{ord,i} \mapsto^{cell}_{detach} (e^{cell,\mathbb{1}}_{ord,i}, e^{cell,\mathbb{2}}_{ord,i}) \,\Big|_{i=1..n}$$

$e^{table,\mathbb{1}}$:=

```
       FROM $e^{table,\mathbb{1}}_{from}$ LATERAL () AS v
      WHERE TRUE
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS $e^{win,\mathbb{1}}$ OVER (
                 PARTITION BY $e^{cell,\mathbb{1}}_{par,1}$, ...
                     ORDER BY $e^{cell,\mathbb{1}}_{ord,1}$ $asc_1$, ...
                         ROWS $\ell_{prec}$ $\ell_{foll}$) AS w1,
                FIRST_VALUE(v.$\rho$) OVER (
                 PARTITION BY $e^{cell,\mathbb{1}}_{par,1}$, ...
                     ORDER BY $e^{cell,\mathbb{1}}_{ord,1}$ $asc_1$, ...
                         ROWS NULL 0) AS w2,
                ROW_NUMBER() OVER (
                 PARTITION BY $e^{cell,\mathbb{1}}_{par,1}$, ...)
                     ORDER BY $e^{cell,\mathbb{1}}_{ord,1}$ $asc_1$, ...
                         ROWS NULL 0) AS w3
     SELECT writewindow($\ell$, v.$\rho$, wins.w2, wins.w3) AS $\rho$,
            wins.w1 AS w1, ...
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

```
       FROM $e^{table}_{from}$ LATERAL () AS v
      WHERE TRUE
   GROUP BY ()
 AGGREGATES ()
     HAVING TRUE
    WINDOWS $e^{win}$ OVER (
                PARTITION BY ($e^{cell}_{par,1}$, ...)
                    ORDER BY ($e^{cell}_{ord,1}$ $asc_1$, ...)
                        ROWS $\ell_{prec}$ $\ell_{foll}$) AS w1
     SELECT wins.w1 AS w1, ...
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

$$\mapsto^{table}_{detach} (e^{table,\mathbb{1}}, e^{table,\mathbb{2}})$$

Figure 8.22: **Rewrite rule: normalized `WINDOWS`. Continued in Figure 8.23.**

```
$ov$ :=  PARTITION BY l.part
           ORDER BY l.rank ASC
             ROWS $\ell_{prec}$ $\ell_{foll}$
```

$$ps \;:=\; \curlyvee^{\mathbb{2}}(e^{cell,\mathbb{2}}_{par,1}) \cup \cdots \cup \curlyvee^{\mathbb{2}}(e^{cell}_{ord,1}) \cup \cdots$$

```
$e^{table,\mathbb{2}}$ :=
 1            FROM $e^{table,\mathbb{2}}_{from}$ LATERAL () AS v,
 2                 readwindow($\boldsymbol{\ell}$, v.$\rho$) LATERAL (v) AS l
 3           WHERE TRUE
 4        GROUP BY ()
 5      AGGREGATES ()
 6          HAVING TRUE
 7         WINDOWS $e^{win,\mathbb{2}}$ OVER $ov$ AS w1,
 8                 $\bigcup ps$ OVER $ov$ AS w2
 9          SELECT v.$\rho$ AS $\rho$,
10                 $\Psi^{\mathbb{2}}$(wins.w1, $Y$(wins.w2)) AS w1
11        ORDER BY ()
12     DISTINCT ON ()
13          OFFMIT NULL NULL
```

Figure 8.23: **Continuation: additional premises of the WINDOWS rewrite rule.**

Lines 9 (w1), 13 (w2) and 17 (w3) specify the window width for the three window functions. The width differs on purpose.

- w1 keeps the values of the input expression (as w1 is supposed to keep the original query semantics).

- w2 and w3 however denote ROWS NULL 0 which defines a window stretching from the beginning of the current partition to the current row. This setting is important since the beginning of the partition is used to determine the offset of the current row and the identifier used to represent the current partition.

The log–writing UDF is located on line 18. It writes its arguments directly to table *logwindow*. Its schema is as in the example of Figure 8.21(d).

Phase 𝟚 Phase 𝟚 employs the UDF readwindow() on line 2 of $e^{table,\mathbb{2}}$ in Figure 8.23. This log–reading UDF is very similar to previous UDFs and not listed (it joins v.ρ with the log and filters for $\boldsymbol{\ell}$). The log yields the columns l.part and l.rank which get referenced in ov (see Figure 8.23). These values re–create the window frames from phase 𝟙 in phase 𝟚 (discussed above). Again, w1 denotes the window function directly assembled from the input expression and computes Where–provenance.

Why–Provenance First, the cell expression ps is assembled from all PARTITION BY and ORDER BY criteria. We employ $\curlyvee^{2}(\cdot)$ in order to yield flat sets. On line 8 (see Figure 8.23), the $\bigcup$ aggregate is used to union the data provenance of all criteria together. The Why operator is applied on line 10 and the Why–provenance is distributed over the (possibly nested) `wins.w1`.

Correctness **C1** is satisfied: the output rows (i.e., $e^{table,\mathbb{1}}$ and $e^{table,\mathbb{2}}$) retain the row identifiers from the input rows (i.e., e^{table}_{from}).

C2 is satisfied through transitivity: the $\mathtt{v}.\rho$ identifiers have been generated according to rule DETACH-JOIN with all (correlated) tuple variables taken into account.

8.7.5.3 Rewrite of Window Expressions

The rewrites of window expressions is formalized using conclusion

$$\boxed{e^{win} \mapsto^{win}_{detach} \left(e^{win,\mathbb{1}}, e^{win,\mathbb{2}}\right)}$$

where

- e^{win} is the expression being rewritten,
- $e^{win,\mathbb{1}}$ is the expression for phase $\mathbb{1}$ and
- $e^{win,\mathbb{2}}$ is the expression for phase $\mathbb{2}$.

In the context of this work, all window functions are created from aggregation functions. Rule DETACH-WINDOW-FUNCTION is listed below.

$$\text{DETACH-WINDOW-FUNCTION}\quad \frac{e^{agg} \mapsto^{agg}_{detach} \left(e^{agg,\mathbb{1}}, e^{agg,\mathbb{2}}\right)}{e^{agg} \mapsto^{win}_{detach} \left(e^{agg,\mathbb{1}}, e^{agg,\mathbb{2}}\right)}$$

The provenance definition and analysis of true window functions (for example, FIRST_VALUE) are considered future work.

9 Experimental Evaluation

An experimental evaluation was carried out in order to determine the runtime overhead of the detached provenance analysis. We determined the slowdown factors

- 1.00 for query normalization,
- 3.56 for phase $\mathbb{1}$,
- 5.48 for phase $\mathbb{2}$ and
- 12.5 in total.

According to these (aggregated and rounded) numbers, phase $\mathbb{2}$ is the most expensive computation step and the query normalization has no performance impact at all. In total, the runtime of a provenance analysis is 12.5 times the runtime of the original query. We employed PostgreSQL as DMBS backend and successfully evaluated the 22 queries of the TPC-H benchmark (with scale factor 1.0). The details are presented in the body of this chapter.

9.1 Implementation in PostgreSQL

We chose a PostgreSQL DBMS backend for the experiments. The implementation of the detached provenance analysis was low–invasive and basically consisted of additional tables and UDFs. A precompiled PostgreSQL binary was used.

9.1.1 Query Rewrites

A manual rewrite of the 22 TPC-H queries was carried out. That means, for each of the i original queries (denoted Q_i^{user}), we produced the normalized query (Q_i^{norm}), the query for phase $\mathbb{1}$ ($Q_i^{\mathbb{1}}$) and for phase $\mathbb{2}$ ($Q_i^{\mathbb{2}}$). An overview of noteworthy differences (compared to the formalization of Chapter 8) is provided below.

- Q13 employed a *left outer join*. Outer joins are not in the scope of our formalization. An outer join potentially produces null values which can be cast to 0 (for the purpose of logging) or to ∅ (the empty provenance annotation). Besides the

```
db=> SELECT ARRAY[1,-2]::int[] AS example;
 example
---------
 {1,-2}
(1 row)
```

Figure 9.1: **Example: a provenance annotation.**

treatment of null values, the provenance semantics and logging are very similar to joins DETACH-JOIN (see Section 8.7.2). We skip the details.

- Q15 involved the view definition `revenue0`. We added view definitions for phases $\mathbb{1}$ and $\mathbb{2}$, i.e. the views `revenue0_𝟙` and `revenue0_𝟚` and applied the rewrite rules recursively for the body of `revenue0`.
- Q16 uses an additional `DISTINCT` flag in its aggregation, i.e. `COUNT(DISTINCT ...)` which is not supported by the SQL backend dialect. We turned the `DISTINCT` flag into a subquery.

9.1.2 Implementation of Provenance Annotations

Provenance annotations according to Definition 5.1 are (flat) sets. Using the unmodified PostgreSQL DBMS, we implemented them as arrays of signed 4–byte integers. The provenance annotation $\{1^e, 2^y\}$ is exemplified in Figure 9.1.

Where– and Why–provenance get mapped to the positive and negative integer domain, respectively. The zero value is forbidden for ambiguity. There are two limitations to keep in mind.

- Integer arrays can contain a maximum number of ≈ 65 million entries (due to a hard–coded memory limitation of PostgreSQL).
- The (4 byte) integer domain ranges between ≈ -2 and $\approx +2$ billion.

The biggest table in the TPC-H benchmark is `lineitem` and has ≈ 6 million tuples (on scale factor 1.0) and 16 columns, i.e. $6 * 16 = 96$ million provenance identifiers[1]. Therefore, aggregating the `lineitem` table would already exceed the maximum array size (96 > 65 million). However, the 22 queries of the benchmark are selective enough to stay within the limit. Regarding to the second limitation (4 byte integer domain), a scale factor of up to ≈ 20 can be supported. If necessary, the 8 byte integers of PostgreSQL could be employed.

[1] If Where– and Why– provenance are both considered, this makes an additional factor of 2.

```
CREATE AGGREGATE ⋃(anyarray) (
    INITCOND='{}',
    STYPE=anyarray,
    SFUNC=array_cat,
    FINALFUNC=array_unique
)
```

(a) **Definition as custom aggregate.**

```
-- phase 𝟙
SELECT
    ...
    SUM(e),
    ...
...
-- phase 𝟚
SELECT
    ...
    ⋃(e),
    ...
...
```

(b) **Idiomatic usage.**

Figure 9.2: **$\bigcup$ in PostgreSQL.**

Array elements are in a fixed sequence (not relevant for the representation of data provenance). Arrays may contain duplicate elements which is against Definition 5.1. We employ explicit duplicate elimination. In the context of this work, we did not evaluate different implementations of provenance annotations. The optimization of phase 𝟚 is generally considered future work, elaborated in Chapter 13.

9.1.2.1 `NULL`

In phase 𝟚, `NULL`–valued provenance annotations may occur. For example, the outer join of Q13 emits cells with `NULL::int[]`. Through implicit casting, these get turned into empty arrays.

9.1.2.2 Binary Union $\cup$

The (small) union combines two sets (i.e., provenance annotations) a and b into a resulting set $a \cup b$.

We implement $\cup$ with the PostgreSQL builtin `array_cat(a, b)` (infix notation: `a || b`) which concatenates two arrays `a` and `b`. Duplicate elimination is not carried out immediately.

9.1.2.3 n–Ary Union $\bigcup$

Our primary method for the implementation of $\bigcup$ is inlining using $\cup$. This is possible as long as the number of operands is statically known.

```
CREATE FUNCTION array_unique(a anyarray)
    RETURNS anyarray AS
$$
    SELECT ARRAY(SELECT DISTINCT v.unnested
                   FROM unnest(a) AS v(unnested))
$$ LANGUAGE SQL IMMUTABLE;
```

Figure 9.3: **Implementation of duplicate removal.**

However for aggregations (see the example in Figure 9.2(b)), we implemented a custom aggregation function in PostgreSQL, listed in Figure 9.2(a). The aggregate definition specifies `INITCOND` (start value: the empty array), `STYPE` (the accumulator type), `SFUNC` (accumulation function: `array_cat()` from above) and `FINALFUNC` (duplicate removal, discussed below).

Optimization Despite of its conciseness, the $\bigcup$ definition from Figure 9.2(a) turned out as a major bottleneck when utilized for the provenance analysis of the TPC-H queries. Therefore, all experimental results presented in this work are based on an optimized version of $\bigcup$. Both implementations (–concise and –optimized) are PostgreSQL aggregates and can be exchanged easily. The idea behind the optimized version is sketched below.

1. Nest each array into a row (with a single attribute).
2. Collect the rows into an array, using `ARRAY_AGG` (which is fast).
3. Carry out a double unnest and thereby yield a single, unnested array.

Step 1. is necessary to avoid constructing two–dimensional arrays. We skip the details.

9.1.2.4 Duplicate Removal

The UDF `array_unique(a)` (listed in Figure 9.3) detects and removes duplicates in array `a`.

9.1.2.5 Why Operator $\mathsf{Y}(\cdot)$

The Why operator is defined in Definition 5.2 and is responsible for generating Why–provenance from Where–provenance. The experiments were carried out using the PostgreSQL implementation listed in Figure 9.4. The basic idea is to

- turn the array `a` into a table with a single column,

```
CREATE OR REPLACE FUNCTION toY(a int[])
RETURNS int[] AS
$$
    SELECT array(
                 SELECT -abs(var.col)
                   FROM unnest(a) AS var(col))
$$ LANGUAGE SQL IMMUTABLE;
```

Figure 9.4: **Implementation of $\mathsf{Y}(\cdot)$.**

- make all integers negative (see line 5) and
- turn the table back into an array.

Duplicate removal is deferred.

Semantics Switch A local change to the implementation of $\mathsf{Y}(\cdot)$ can be used to switch between different provenance semantics.

- The computation of Why–provenance is skipped if `toY` is implemented as the empty array `ARRAY[]`. Where–provenance stays unaffected.
- Where–and Why–provenance get merged together if `toY` is implemented as the identity $\mathsf{id}(\mathtt{a})$.

9.1.2.6 Recursive Set Operators

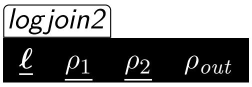
We did not implement the recursive set operators (i.e., $\curlyvee^{\mathbb{1}}(\cdot)$, $\curlyvee^{\mathbb{2}}(\cdot)$ and $\Psi^{\mathbb{2}}(\cdot,\cdot)$) but used inlining where needed.

9.1.3 Implementation of Logging: Overview

Relational logs were used in the experiments (evaluating other storage options is considered future work). For TPC-H, we defined 10 logging tables with their respective read/write UDFs. The individual TPC-H queries had between 1 and 4 logging locations.

9.1.4 Implementation of Logging: Example

The implementation of logging for 2–way joins is exemplified. The schema from Section 8.7.2 is reproduced in Figure 9.5(a). We implemented the log according to Figure 9.5(b). It defines the compositional primary key $(\boldsymbol{\ell},\ \rho1,\ \rho2)$. One essential task



(a) **Log schema.**

```
-- sequence definition: shared between logs
CREATE SEQUENCE $\rho_{seq}$;

-- definition of log.join2
CREATE TABLE log.join2 ($\ell$ integer NOT NULL,
                        $\rho_1$ integer NOT NULL,
                        $\rho_2$ integer NOT NULL,
                        $\rho_{out}$ integer NOT NULL);
ALTER TABLE log.join2 ADD PRIMARY KEY ($\ell$, $\rho_1$, $\rho_2$);
ALTER TABLE log.join2 ALTER COLUMN $\rho_{out}$
                      SET DEFAULT NEXTVAL('$\rho_{seq}$');
```

(b) **Table definition in PostgreSQL.**

Figure 9.5: **Example: 2–way joins and log implementation.**

for the primary key is to block duplicate entries (elaborated below). The `NEXTVAL(.)` expression generates unique ρ values.

9.1.4.1 UDF Placement

Figure 9.6 exemplifies the placement of logging UDFs. Be aware that the example query is normalized, i.e. there are no additional SFW clauses.

In phase 𝟙, the ρ values of the contributing rows got written to the according log. The first argument (here: 42) is to distinguish between different logging locations. It must be unique and match between the two phases.

In phase 𝟚, the log gets read. Basically, the log replaced the `WHERE`–clause. Due to implicit `LATERAL` semantics, the variables `row1` and `row2` are visible to the UDF. The resulting values in column `log.`ρ match with those from phase 𝟙.

9.1.4.2 UDF Implementations

Continuing the example from above, the UDF implementations for the 2–way join are listed in Figure 9.7. The implementation of `readjoin()` is straightforward. The UDF can be declared `STABLE` which is beneficial for query optimization. The implementation of

```
SELECT
    writeJoin(42, row1.ρ,
                  row2.ρ) AS ρ,
    ...
FROM
    (...) AS row1,
    (...) AS row2
WHERE
    ...
```

(a) **Phase 𝟙–query.**

```
SELECT
    log.ρ
            AS ρ,
    ...
FROM
    (...) AS row1,
    (...) AS row2,
    readJoin(42, row1.ρ,
                 row2.ρ) AS log(ρ)
```

(b) **Phase 𝟚–query.**

Figure 9.6: **Placement of logging UDFs.**

`writejoin()` basically writes its arguments to the log, returning a fresh row identifier ρ_{out}. On top of that, lines 13-15 specify an `EXCEPTION` handling (discussed below).

9.1.4.3 Duplicate Log Writes

The necessity of detecting / avoiding duplicate log entries is induced by the limitations of query optimizers. Duplicate log writes can be triggered when the query executor repeats the same evaluation multiple times. Of course, this is inefficient and would not happen in a perfect world. An example for when repeated evaluation occurs is failed query decorrelation (a concrete example is Q17, discussed in this chapter). For regular query evaluation, the consequence of surplus evaluations is decreased performance. In context of the detached provenance analysis, the consequences are worse. By default, any log write yields a unique ρ_{out}. If the same log write is carried out twice, two different ρ_{out} are produced (for the same combination of $\rho_1, \dots$). This is an ambiguity and in phase 𝟚 it is no longer clear which input/output ρ values belong to each other. To avoid such ambiguity beforehand, we employed primary keys and exception handling.

The terminal listing of Figure 9.8 exemplifies how our implementation transparently deals with duplicate log writes. Calling `writeJoin(1, 10, 51)` twice yields the same ρ value `101`. Under the hood, the primary key stops duplicate log entries from being written. The program logic on lines 13-15 (see Figure 9.7(a)) handles any `UNIQUE_VIOLATION` being raised and looks up the already existing ρ value through calling `readjoin()`. For reasons not further elaborated, we recommend to inline `readjoin()` which significantly increases the performance.

```
CREATE FUNCTION writejoin($\boldsymbol{\ell}$ integer,
                          $\rho_1$ integer,
                          $\rho_2$ integer)
    RETURNS integer AS
$$
DECLARE
    res integer;
BEGIN
    INSERT INTO log.join2 ($\boldsymbol{\ell}$, $\rho_1$, $\rho_2$)
        VALUES ($\boldsymbol{\ell}$, $\rho_1$, $\rho_2$)
        RETURNING $\rho_{out}$ INTO res;
        RETURN res;
EXCEPTION
    WHEN UNIQUE_VIOLATION THEN
        RETURN readJoin($\boldsymbol{\ell}$, $\rho_1$, $\rho_2$);
END;
$$ LANGUAGE PLPGSQL VOLATILE;
```

(a) **Log–writing UDF.**

```
CREATE FUNCTION readjoin($\boldsymbol{\ell}$ integer,
                         $\rho_1$ integer,
                         $\rho_2$ integer)
    RETURNS TABLE($\rho$ integer) AS
$$
    SELECT j.$\rho_{out}$
      FROM log.join2 AS j
     WHERE j.$\boldsymbol{\ell}$=$\boldsymbol{\ell}$
       AND j.$\rho_1$=$\rho_1$
       AND j.$\rho_2$=$\rho_2$
$$ LANGUAGE SQL STABLE;
```

(b) **Log–reading UDF.**

Figure 9.7: **UDF logging implementation for 2–way joins.**

```
-- start sequence at 101
db => ALTER SEQUENCE $\rho_{seq}$ RESTART WITH 101;
ALTER SEQUENCE

-- (1): first log write
db => SELECT writejoin(1, 10, 51);
 writejoin
-----------
       101
(1 row)

-- (2): second log write
db => SELECT writejoin(1, 10, 52);
 writejoin
-----------
       102
(1 row)

-- (3): first log write (repeated)
db => SELECT writejoin(1, 10, 51);
 writejoin
-----------
       101
(1 row)

-- final log
db => TABLE log.join2;
 $\boldsymbol{\ell}$ | $\rho_1$ | $\rho_2$ | $\rho_{out}$
---+----+----+-----
 1 | 10 | 51 | 101
 1 | 10 | 52 | 102
(2 rows)
```

Figure 9.8: **Example: unambiguous log entries (i.e., $(\boldsymbol{\ell}, \rho_1, \rho_2) \mapsto \rho_{out}$) are enforced.**

9.2 Setup

9.2.1 Hard– and Software

Hardware The experiments were carried out on a machine with two Intel Xeon 5570 CPUs[2]. The machine had 72 GB of main memory installed and plenty of HDD storage. Due to caching, HDD overheads were supposedly not significant.

Operating System The machine ran an Ubuntu 16.04 operating system with a Linux 4.4 kernel. Swap space and cron daemon were disabled during the experiments, however the machine was connected to the network. To compensate for network events and other pseudo–random effects, all time–sensitive experiments were carried out five times. We ignored the minimum and maximum value and took the arithmetic average of the rest.

DBMS We picked PostgreSQL [Pos] as the DBMS backend for our experiments. It is a very accessible system (well documented and open source) and has been modified in the context of a number of research projects. For example, there is *PERM* and we carried out a direct comparison with our approach (subject of Chapter 11).

We installed the precompiled PostgreSQL 9.5 package delivered by the distribution's package manager. This is possible because our low–invasive approach to provenance analysis does not require any core modification. More recent PostgreSQL versions (as of September 2019, version 11.5) have built–in support for parallel query evaluation. A topic we consider future work.

9.2.2 Database

The TPC-H Benchmark We used the TPC-H benchmark in version 2.18.0. TPC-H is a well–established benchmark in the DB field for OLAP workloads. It consists of eight tables and 22 query templates. The benchmark comes with a data / query generator tool. For our experiments, we

- created a database with scale factor 1 ($\approx$ 1 GB of relational data) and
- generated a single set of 22 concrete queries. Using the rewrite rules from Chapter 8, the queries for provenance analysis have been derived manually.

One drawback of TPC-H is its high age of $\approx$ 20 years. More recent SQL features (like window functions, row values, arrays) are not covered.

[2] The CPUs provided 16 logical threads, however the experiments did not exploit parallelization.

Keys and Indices All primary keys according to [TPC][p.18, sec.1.4.2.2] and foreign keys according to [TPC][p.18, sec.1.4.2.3] have been created. PostgreSQL implicitly adds (composite) indices for primary keys. On top of that, we added

- (composite) indices for (composite) foreign keys and
- single–column indices for the remaining columns.

Populating the Database The TPC-H benchmark comes with a data generator. We used the scale factor $s = 1.0$ to generate the payload data (emitted in CSV format). This produces ≈ 1GB of data (files) and the biggest table `lineitem` contains ≈ 6 million rows.

Loading the data into the database takes ≈ 30 minutes. I.e.,

- read the CSV files and populate the respective database tables,
- create the derived input tables for phases 𝟙 and 𝟚 (discussed next),
- create indices and
- create statistics (using the command: `VACUUM (FULL, ANALYZE)`).

The steps discussed next are specific to the detached provenance analysis. We are about to discuss how to create the input tables for phases 𝟙 and 𝟚. For example, we derived tables `lineitem𝟙` and `lineitem𝟚` from `lineitem`. In the formalization according to Definition 8.5, this is the detachrm(liftrm(·)) step.

Tables for Phase 𝟙 The phase 𝟙–specific tables are a copy of the original tables with an additional ρ column. The sequence defined in Figure 9.9 is used to generate the required unique values for the ρ column. Right below, the table `lineitem𝟙` is being defined. The $*$ on line 8 basically makes a copy of schema and rows of the already existing table `lineitem`.

Finally, the same keys and indices (already created for the original tables) are to be recreated for the tables of phase 𝟙 (not exemplified). This step is **very** important for a fair comparison of the query runtimes.

Tables for Phase 𝟚 The tables of phase 𝟚 got populated with provenance annotations.

As a concrete example, the DDL snippet listed in Figure 9.10 defines the table `lineitem𝟚`. The ρ column (see line 7) is copied from `lineitem𝟙` in order to enforce symmetry between both tables. The sequence A_{seq} is used to create unique annotations.

```
-- once per instance: initialize a PostgeSQL sequence
CREATE SEQUENCE ρ_seq;

-- phase 𝟙 table definition and population
CREATE TABLE lineitem𝟙 AS
    SELECT
        nextval('ρ_seq')::int AS ρ,
        *
    FROM
        lineitem;
```

Figure 9.9: **Excerpt: definition and population of the phase 𝟙–tables.**

```
 1  -- initialize a sequence for provenance annotations
 2  CREATE SEQUENCE A_seq;
 3
 4  -- phase 𝟚 tables
 5  CREATE TABLE lineitem𝟚 AS
 6      SELECT
 7          ρ AS ρ,
 8          ARRAY[nextval('A_seq)::int] AS L_ORDERKEY,
 9          ...
10      FROM
11          lineitem𝟙;
12
13  -- phase 𝟚 indices
14  CREATE UNIQUE INDEX ON lineitem𝟚 (ρ);
```

Figure 9.10: **Excerpt: definition and population of the phase 𝟚–tables.**

Indices for tables in phase 𝟚 are entirely different. It makes no sense to create an index on a column filled with provenance annotations (the provenance annotations are never referenced in any predicate). Only the ρ columns (one column per table) is inspected in predicates. We define them as primary keys, exemplified on line 14 of Figure 9.10.

Database Size In the end, there are three sets of tables:

- original TPC-H: 1.3 GB (with 3.4 GB of indices),
- phase 𝟙: 1.3 GB (with 3.4 GB of indices) and
- phase 𝟚: 3.1 GB (with 0.18 GB of indices).

In total, the tables of phase 𝟙 and 𝟚 consume 4.4 GB of disk space (with 3.6 GB of indices). In the context of our experiments, we did not care about disk space but about simplicity (all columns indexed) and performance (no time overhead for on–the–fly table fabrication).

The tables of phase 𝟚 are quite space demanding with 3.1 GB. This is because the singleton arrays of PostgreSQL have a bad overhead / payload ratio (improving with array length).

9.3 Normalization Overhead

The purpose of query normalization (subject of Chapter 7) is to simplify the queries for the provenance analysis. The TPC-H benchmark has a normalization overhead of 1.00 (rounded to two digits). This number is the geometric mean calculated from the quotients

$$\frac{T_i^{norm}}{T_i^{user}}$$

for each of the $i = 1, \ldots 22$ queries. T_i^{user} is the wall clock time for the evaluation of the original i–th TPC-H query and T_i^{norm} is its normalized counterpart. Figure 9.11 shows the quotient for each pair of queries. The unitless slowdown (if < 1.0, it is a speedup) is plotted on the y–axis. A perfect query optimizer would generate the same plans for the semantics–preserving query normalization. Below, we carry out a short discussion of some queries. The achievement of our normalization is that it simplifies the queries without performance impact.

9.3.1 Speedup of Q9

The slowdown of Q9 was ≈ 0.94 which is a small performance gain. The inner query of Q9 consists of a 6–way join and an accordingly complex conjunctive `WHERE` predicate (not listed). The PostgreSQL query optimizer underestimated the row cardinality by the factor $\approx 7,000$ (see Figure 9.12(a): 47 vs. 319404 rows). This misprediction happens in both queries: normalized and not normalized. However, due to normalization the new plan employed a `HashAggregate` node (originally, this was a `GroupAggregate`). `HashAggregate` is more performant for the $(7,000$ times bigger) row cardinality which explains the improved query runtime. But why does the query optimizer employ different operators? For both queries, the cardinality estimate is way off.

Looking at Figure 9.13 (Q9 before and after normalization), the rewritten query has dedicated SFW expressions for *sorting* and *aggregation*. We assume that the query optimizer

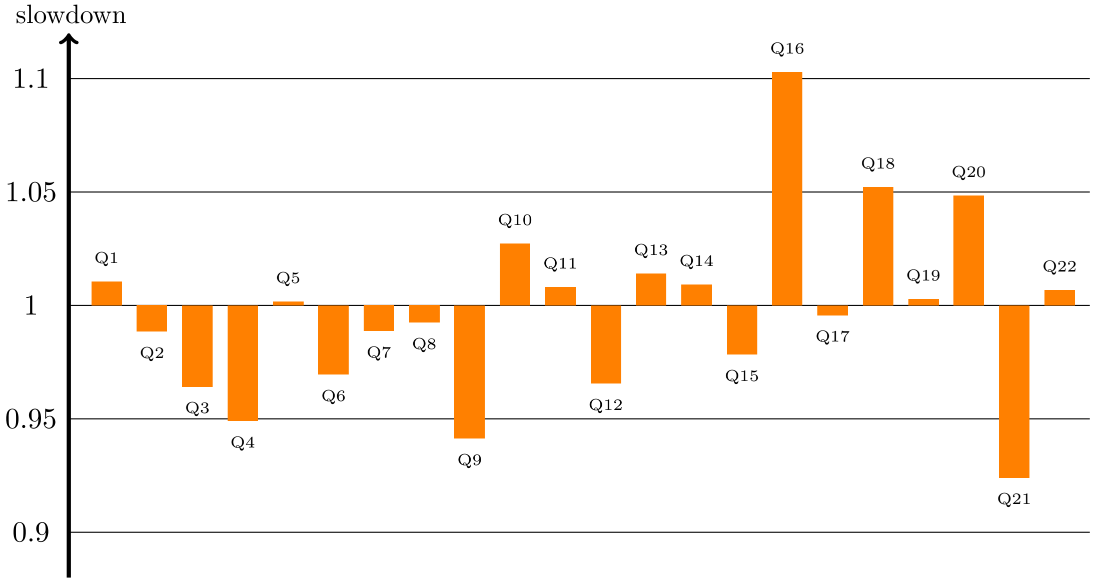


Figure 9.11: **Query speedup and slowdown through normalization.**

was not aware of the nesting and just planned the two SFW expressions individually. For the not normalized Q9, the query optimizer exploited similarities between sorting and grouping. Due to cardinality underestimation, this strategy backfired.

9.3.2 Speedup of Q21

We found an improved query performance for Q21 (slowdown: ≈ 0.92). The main difference according to the query plans is how the correlated subqueries of Q21 get optimized. In the original query, PostgreSQL plans the correlated `EXISTS` and `NOT EXISTS` subqueries as a semi join and an anti join, respectively. In the normalized query, PostgreSQL employs more general subplan nodes instead. This resulted in different join orders. We assume that the different optimizer decisions are related to the additional tuple variables our rewrite introduces. Figure 9.14 lists an excerpt of the normalized Q21. The variable `l1` gets bound outside and is inspected on line 10. Our normalization procedure introduces line 7. Re–binding `l1` might be the reason that the PostgreSQL optimizer did not recognize an anti join pattern. Instead, the more general optimization employing a subplan was carried out. In this specific case, the more general optimization turned out to evaluate faster.

```
Sort  (cost=49240.58..49240.70 rows=47 width=66)
      (actual time=4668.951..4668.964 rows=175 loops=1)
  ->  HashAggregate  (cost=49237.98..49238.80 rows=47 width=53)
                     (actual time=4668.640..4668.684 rows=175 loops=1)
        ->  Nested Loop  (cost=1.57..49237.16 rows=47 width=53)
                         (actual time=0.150..4178.316 rows=319404 loops=1)
              -> ...
              -> ...
```

(a) **Plan of the normalized query.**

```
GroupAggregate  (cost=49238.46..49240.23 rows=47 width=53)
                (actual time=4875.346..5240.587 rows=175 loops=1)
  ->  Sort  (cost=49238.46..49238.58 rows=47 width=53)
            (actual time=4873.974..4924.113 rows=319404 loops=1)
        ->  Nested Loop  (cost=1.57..49237.16 rows=47 width=53)
                         (actual time=0.159..4396.589 rows=319404 loops=1)
              -> ...
              -> ...
```

(b) **Plan of the original query.**

Figure 9.12: **Excerpts from the execution plans for Q9.**

9.3.3 Slowdown of Q16

The slowdown of Q16 is not directly associated with the normalization rules. Instead, it is caused by a substitution of

```
COUNT(DISTINCT ps_suppkey)
```

with an according subquery implementing the `DISTINCT` semantics separately. The slowdown was ≈ 1.1. Details are omitted.

9.4 Phase 𝟙 Overhead

Producing logs is the essential task of phase 𝟙. As an example for a phase 𝟙–rewrite, Figure 9.15 shows an excerpt of Q2. The normalized query (on the left) contains a multi–way join between base tables and the correlated tuple variable `p`. For the phase 𝟙–query (on the right), the base tables get renamed (i.e., suffixed with 𝟙) and the logging call **`writejoin(·)`** gets added. The logging call has a statically unique identifier (**3**) and references all ρ columns. For Q2, we employed three logging calls in total (not listed).

```
SELECT
    nation,
    o_year,
    SUM(amount) AS sum_profit
FROM
    (
        ...
    ) AS profit
GROUP BY
    nation,
    o_year
ORDER BY
    nation ASC,
    o_year DESC
```

(a) **Before normalization.**

```
SELECT
    v.nation AS nation,
    v.o_year AS o_year,
    v.sum_profit AS sum_profit
FROM
    (SELECT
         nation AS nation,
         o_year AS o_year,
         SUM(amount) AS sum_profit
     FROM
         (
             ...
         ) AS profit
     GROUP BY
         nation,
         o_year
     ) AS v
ORDER BY
    v.nation ASC,
    v.o_year DESC
```

(b) **After normalization.**

Figure 9.13: **Excerpts of Q9.**

```
WHERE
    ...
    AND NOT EXISTS (
        SELECT
            ...
        FROM
            (SELECT l1.*) AS l1, --normalization
            ...
        WHERE
            ... --references to l1
    )
    ...
```

Figure 9.14: **Excerpt: Q21 after normalization.**

```
SELECT




    ps.ps_supplycost
FROM
    (SELECT p.*) AS p,
    partsupp AS ps,
    ...
    region AS r
WHERE
    p.p_partkey = ...
```

(a) **Normalized query.**

```
SELECT
    writejoin(3, p.ρ,
                 ps.ρ,
                 ...
                 r.ρ) AS ρ,
    ps.ps_supplycost
FROM
    (SELECT p.*) AS p,
    partsupp𝟙 AS ps,
    ...
    region𝟙 AS r
WHERE
    p.p_partkey = ...
```

(b) **Phase 𝟙–query.**

Figure 9.15: **Example: rewrite of Q2 for phase 𝟙.**

9.4.1 Results

According to our experiments, the slowdown for log writing was 3.56. This is the geometric mean from the quotients

$$\frac{T_i^{\mathbb{1}}}{T_i^{norm}}$$

for the i–th TPC-H query. Our discussion of phase 𝟙 is however based on

$$\frac{T_i^{\mathbb{1}} - T_i^{norm}}{|log_i|} \tag{9.1}$$

, where $|log_i|$ is the log cardinality for query i. Put in plain words, Equation (9.1) yields the absolute time for writing a single log row.

Runtime Overheads Figure 9.16(a) shows the runtime overheads in terms of Equation (9.1). The three outliers Q2, Q17 and Q18 are to be discussed below. The remaining 19 overheads are in the value range between $0.005ms$ (Q4) and $0.11ms$ (Q19).

Log Sizes Figure 9.16(b) shows the log sizes in row counts (and kilobytes). The smallest log had 121 rows (Q19) and the biggest one had ≈ 6 million rows (Q1). Log sizes are specific to the query and input tables. For Q19 and Q1, selectivities of their `WHERE` predicates were essential.

- Q19 specified a join of `lineitem` and `part` which yielded 121 rows (selectivity: $1.0 \cdot 10^{-10}$). The result table had a single row due to additional aggregation.
- Q1 applied a filter to `lineitem` and yielded $\approx$ 6 million rows (selectivity: 0.99). The result table had four rows due to additional aggregation.

Optimizing the row count in logs is difficult, because the resulting ρ columns are generally needed in subsequent evaluation steps. It is therefore not an option to invert the logging, i.e. only record the unqualified row identifiers. However, redundancy could be exploited. In case of Q1, there are only four output rows (with four different row identifiers ρ_{out}). In the current (normalized) log format, these four identifiers get stored literally for all of the 6 million log entries.

9.4.2 Speedup of Q2

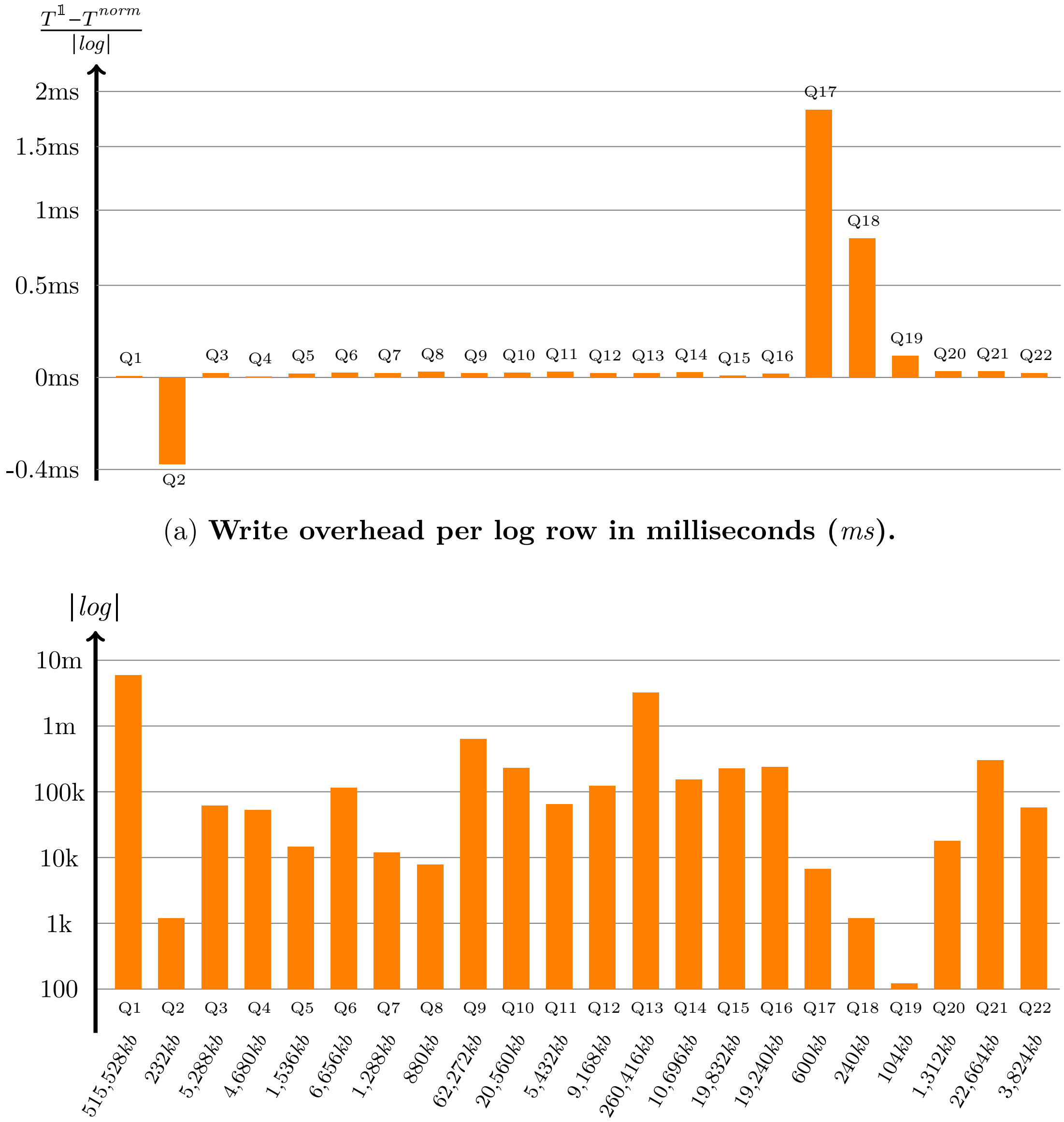
The logging overhead of Q2 is $-0.38ms$ per row, i.e. writing logs makes the query run faster. This result is strange at first sight but finds an explanation in optimizer decisions. We provide the relevant query plans in Figure 9.17. The plan of Q_2^{norm} employs a sequential scan while $Q_2^{\mathbb{1}}$ profits from a predicate pushdown and an index scan (both scans repeated 460 times). The underlying reasons for the (different) optimizer decisions are not evident.

We carried out an additional sanity check and disabled sequential scans. After a re–evaluation of the normalized query, its plan had changed as expected (i.e., index scan with predicate pushdown) and the runtime had improved by a factor of $\approx$ 19. The unmodified (not normalized) Q2 shared the same optimization issue with the normalized Q2.

9.4.3 Slowdown of Q18

We observed a considerable slowdown for Q18. On average, writing a single log row took $0.80ms$ (with $|log| = 1197$). Looking at Figure 9.16(a), this makes Q18 one of the three outliers in phase $\mathbb{1}$.

The issue can be narrowed down to Q18's (non–correlated) subquery listed in Figure 9.18 which produced 399 log rows and expended $2.83ms$ per row. The according query plan is however not detailed enough to pinpoint where the time goes. We presume (based on running derivative queries) that the aggregate expression on line 3 (**`ARRAY_AGG`**) got



(a) **Write overhead per log row in milliseconds (*ms*).**

(b) **Log sizes in cardinality and kilobytes (*kb*).**

Figure 9.16: **Phase 𝟙 results.**

```
->  Nested Loop  (cost=116.82..63853.00 rows=1 width=170)
                 (actual time=2.894..790.444 rows=460 loops=1)
      Join Filter: (ps.ps_suppkey = s.s_suppkey)
      Rows Removed by Join Filter: 4599540
      ->  ... -- 460 rows (expected 1)
      ->  Seq Scan on supplier s
            (cost=0.00..322.00 rows=10000 width=144)
            (actual time=0.002..0.730 rows=10000 loops=460)
```

(a) **Plan for the normalized query.**

```
->  Nested Loop  (cost=124.72..67360.50 rows=1 width=182)
                 (actual time=2.404..125.446 rows=460 loops=1)
      ->  ... --  460 rows (expected 1)
      ->  Index Scan using supplier_1_pkey on supplier_1 s
            (cost=0.29..0.30 rows=1 width=148)
            (actual time=0.002..0.003 rows=1 loops=460)
            Index Cond: (s_suppkey = ps.ps_suppkey)
```

(b) **Plan for phase $\mathbb{1}$.**

Figure 9.17: **Query plans: Q2 (excerpts).**

```
...
    SELECT
        writeagg(4, ARRAY_AGG(l.ρ)) AS ρ,
        l.l_orderkey AS l_orderkey
    FROM
        lineitem_1 AS l
    GROUP BY
        l.l_orderkey
    HAVING
        SUM(l.l_quantity) > 300
...
```

Figure 9.18: **Excerpt of Q18: the relevant subquery causing the significant overhead.**

```
SELECT
    writeJoin(2, p.ρ, l.ρ) AS ρ,
    l.l_quantity AS l_quantity,
    l.l_partkey AS l_partkey
FROM
    (SELECT p.*) AS p,
    lineitem_1 AS l
WHERE
    l.l_partkey = p.p_partkey
```

(a) **Subquery with correlated tuple variable `p`.**

```
FOR p IN part:
    -- here: 204 iterations
    ...
    FOR l IN lineitem:
        -- here:
        --   6088 iterations
        --   with 204 unique bindings of p
        ...
        subquery(p)
```

(b) **High–level execution plan of Q17 in pseudocode.**

Figure 9.19: **Q17 in phase 𝟙.**

evaluated *in parallel* with the (highly selective) `HAVING` predicate (rather than *after* the predicate). In consequence, the result of **`ARRAY_AGG`**

- got written to the log in 57 cases[3] and
- got thrown away in ≈ 1.5 million cases (`HAVING` selectivity: ≈ 0.004%).

This means that most (i.e. ≈ 99.996%) of the computational effort invested in evaluating **`ARRAY_AGG`** was wasted. Manipulation of the selectivity (new selectivity: 100%; new log cardinality: ≈ 6.0 million) reduced the average time to ≈ 0.01ms per log row.

9.4.4 Slowdown of Q17

Q17 experiences an overhead of 1.8ms per log row (with $|log| = 6675$). This was the biggest overhead of all phase 𝟙–queries.

[3] The 57 arrays written to the log yielded 399 rows. This is because arrays generally got unnested and hence one array became multiple rows in the log.

The reason for the performance loss is that ≈ 180, 000 redundant log writes got triggered. These had to be detected and skipped (explained in Section 9.1.4.3) and made the actual 6675 log entries appear expensive. Without duplicate removal, the overhead would have dropped to ≈ $0.02ms$ per log row (and $|log| = 187,625$). The main question therefore is: why did Q17 trigger superfluous log writes?

The PostgreSQL query optimizer failed to decorrelate the subquery of Figure 9.19(a). This has nothing to do with the provenance analysis of phase 𝟙 or the normalization but already occurred with the unaltered Q17. However, in phase 𝟙 and through the logging call placed on line 2 (see Figure 9.19(a)), the negative consequences got amplified. The pseudocode listed in Figure 9.19(b) sketches the PostgreSQL query plan:

- In an outer query (line 1), `p` gets bound (cardinality: 204).
- In the same (outer) query (line 4), a join with `lineitem` is carried out. In total, 6088 intermediate rows are being produced. The 204 unique valuations of `p` get replicated accordingly.
- The subquery (called on line 9 and SQL code in Figure 9.19(a)) only references `p` (ignoring `l`) and hence, suffers from redundant valuations.
- Inside of the subquery, another join with `lineitem` is carried out and yields the ≈ 180, 000 redundant logging calls.

Future versions of PostgreSQL might feature improved query decorrelation and massively reduce the overhead of Q17 in phase 𝟙. HyPer [KN10] by A. Kemper and T. Neumann and its advanced decorrelation algorithm [NK15] may provide an alternative DBMS backend for provenance analysis.

9.4.5 Optimization

A simple logging optimization can reduce the log size (and runtime overhead) considerably. Figure 9.20 exemplifies this for Q1. There are

- two logging locations (lines 2 and 8) on the left and
- only one logging location on the right (line 2) — through optimization.

The main idea of this optimization is that `writeFilter()` only forwards the ρ values, i.e. it implements the identity (with a side–effect). It does not merge ρ values or create new ones (like it happens for logging of joins). From value perspective, the UDF call can be replaced with `l.`ρ (see line 8). But why is the log not needed?

The optimization is restricted to `writeFilter()` sitting directly inside of another log–writing subquery (`writeagg()` on line 2). The semantics of `writeagg()` is to log any

```
SELECT
    writeagg(1, ARRAY_AGG(v.ρ))
        AS ρ,
    SUM(...),
    ...
FROM
    (SELECT
         writefilter(2, l.ρ)
             AS ρ,
         ...
     FROM
         lineitem𝟙 AS l
     WHERE
         ...
     ) AS v
GROUP BY
    ...
```

(a) **Before optimization.**

```
SELECT
    writeagg(1, ARRAY_AGG(v.ρ))
        AS ρ,
    SUM(...),
    ...
FROM
    (SELECT
         l.ρ
             AS ρ,
         ...
     FROM
         lineitem𝟙 AS l
     WHERE
         ...
     ) AS v
GROUP BY
    ...
```

(b) **After optimization.**

Figure 9.20: **Example: logging optimization of Q1.**

pair of input tuple and group. If nothing gets logged, the according input tuple is filtered. Hence, both logging UDFs implement filter semantics and this redundancy can be exploited.

We applied this optimization to queries Q1, Q4 and Q15 (the unoptimized queries were not part of our performance evaluation). Without this optimization, the log of Q1 (already the biggest log with ≈ 6 million rows) would have been even bigger. We carried out a comparable optimization for Q16, but the details are omitted.

9.5 Phase 𝟚 Overhead

In phase 𝟚, provenance annotations get propagated. Figure 9.21(b) lists an excerpt of Q2 as a typical example for a phase 𝟚–query. For reference, the according phase 𝟙–query is listed in Figure 9.21(a). These queries have been created through application of the rewrite rules presented in Chapter 8 — with adaptions to the SQL dialect of PostgreSQL.

- In phase 𝟙, the query utilizes multiple `FROM` entries and a **`WHERE`** predicate to realize a join. The **`writejoin()`** call logs row identifiers of the join partners which have qualified against the predicate.

```
 1 SELECT
 2     writejoin(3, p.ρ,
 3                  ps.ρ,
 4                  ...
 5                  r.ρ) AS ρ,
 6     ps.ps_supplycost
 7 FROM
 8     (SELECT p.*) AS p,
 9     partsupp𝟙 AS ps,
10     ...
11     region𝟙 AS r
12 WHERE
13     p.p_partkey = ...
14
15
16
```

(a) **Phase 𝟙–query.**

```
 1 SELECT
 2     log.ρ AS ρ,
 3
 4
 5
 6     y.y | ps.ps_supplycost
 7 FROM
 8     (SELECT p.*) AS p,
 9     partsupp𝟚 AS ps,
10     ...
11     region𝟚 AS r,
12     readjoin(3, p.ρ,
13                 ps.ρ,
14                 ...
15                 r.ρ) AS log(ρ),
16     toWhy(p.p_partkey | ...) AS y(y)
```

(b) **Phase 𝟚–query.**

Figure 9.21: **Example: excerpt of Q2.**

- In phase 𝟚, the predicate is basically replaced with an inspection of the log (see lines 12-15). The columns inspected in the **WHERE** predicate constitute the Why–provenance of phase 𝟚, assembled on line 16.

9.5.1 Provenance Size

The most important factor regarding to the phase 𝟚 runtime is the size of the data provenance being computed. We denote the provenance size with $|prov_i|$, i.e. the cardinality of all provenance annotations. For example, the table

output

ρ	*col1*	*col2*
42	$\{1^e, 2^y\}$	$\{1^e, 2^y\}$

has a provenance size of $|\{1^e, 2^y\}| + |\{1^e, 2^y\}| = 4$.

The provenance size for each query is provided in Figure 9.22.

- Q1 had the biggest provenance size of ≈ 250 million. The query aggregated (nearly) the entire `lineitem` table. Also, we found the single biggest provenance annotation of the benchmark with ≈ 18 million entries for Q1.
- Q11 had a provenance size of ≈ 200 million. Its `HAVING` predicate had an SFW

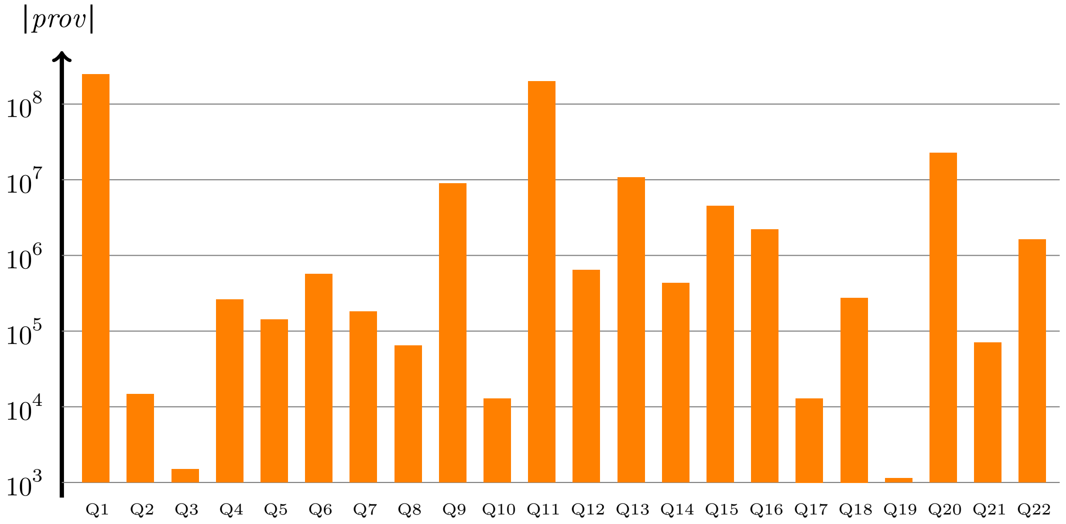


Figure 9.22: **Provenance size.**

subexpression which aggregated $\approx 30,000$ rows.

- Q3 had a `LIMIT 10` clause and yielded one of the smallest provenance cardinalities.
- Q19 was highly selective. Its provenance size of $\approx 1,000$ was the smallest of the benchmark.

9.5.2 Absolute Performance

Our experimental evaluation found a total slowdown for phase 2 of 5.48. This is the geometric mean from the quotients

$$\frac{T_i^{2}}{T_i^{norm}}$$

for the i–th TPC-H query. The individual slowdowns are represented as the wide bars of Figure 9.23. The range of values is between

- 0.065 for Q18 (which makes it a speedup) and
- 996 for Q11.

The narrow bars denote the total overhead for the entire provenance analysis. The formula for the total slowdown is

$$\frac{T_i^{\mathbb{1}} + T_i^{\mathbb{2}}}{T_i^{user}}$$

which integrates all slowdowns (or speedups) of normalization, phase 𝟙 and phase 𝟚. This formula constitutes our measure for the total performance of the detached provenance analysis. Compared to Definition 8.5, the formula omits the query rewrites, database preparation and the final merge step. The total slowdowns are represented as narrow bars in Figure 9.23. Their geometric mean is 12.5. We state some general observations.

- Q11, Q1 and Q20 had the biggest slowdowns of phase 𝟚. This result is consistent with their comparably big provenance sizes (see Figure 9.22).
- Q2, Q18, Q19 and Q21 showed a speedup in phase 𝟚. Q19 is discussed below.
- Q2 was by far the best–performing query in the benchmark. Its total slowdown was 0.39 which makes it a speedup. Put in other words, computing the data provenance of Q2 is faster than evaluating the original Q2.
- Q17's performance in phase 𝟙 was lower than in phase 𝟚. A discussion of Q17's phase 𝟙 can be found in Section 9.4.4.

Next, we factor in the provenance size and discuss selected queries.

9.5.3 Performance Relative to Provenance Size

In this part of the discussion, we focus on

$$\frac{T_i^{\mathbb{2}} - T_i^{norm}}{|prov_i|} \tag{9.2}$$

which denotes the time expended (in phase 𝟚) for computing a single provenance identifier. The results are presented individually for any i–th TPC-H query in Figure 9.24. Most notably, there are four queries which have a negative overhead, i.e. they evaluated faster in phase 𝟚 than the normalized TPC-H query got evaluated. We discuss

- Q19 having the smallest overhead of $-69\mu s$ and
- Q10 having the biggest overhead of $2918\mu s$.

Compared to Figure 9.23, the three most expensive queries Q11, Q20 and Q1 are no longer remarkable after their provenance sizes have been factored in.

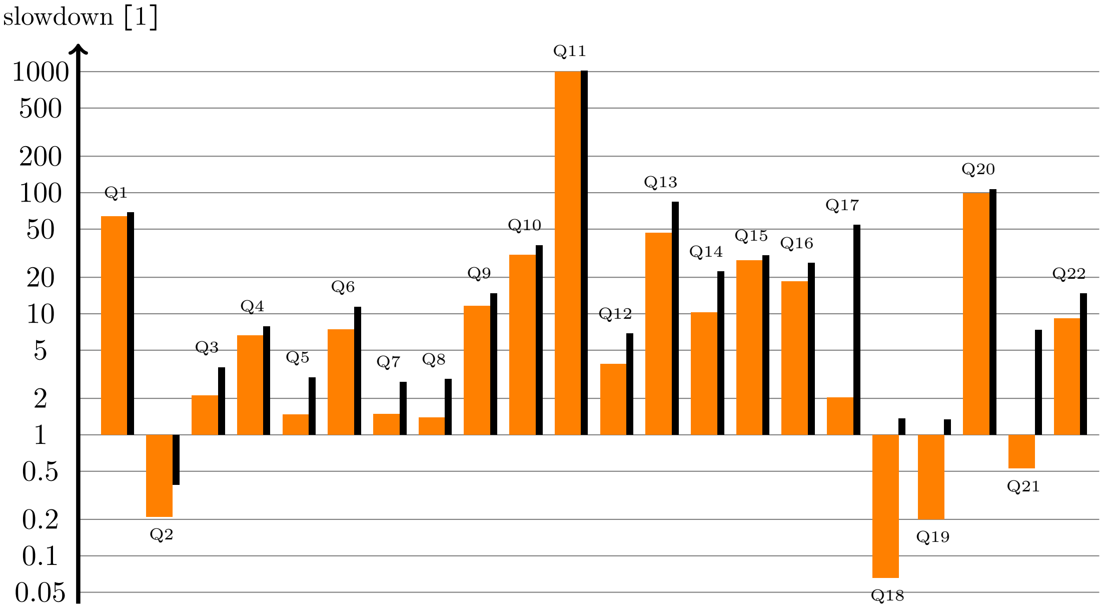


Figure 9.23: **Wide bars denote the slowdown of phase 2. Narrow bars denote the total slowdown for the detached provenance analysis.**

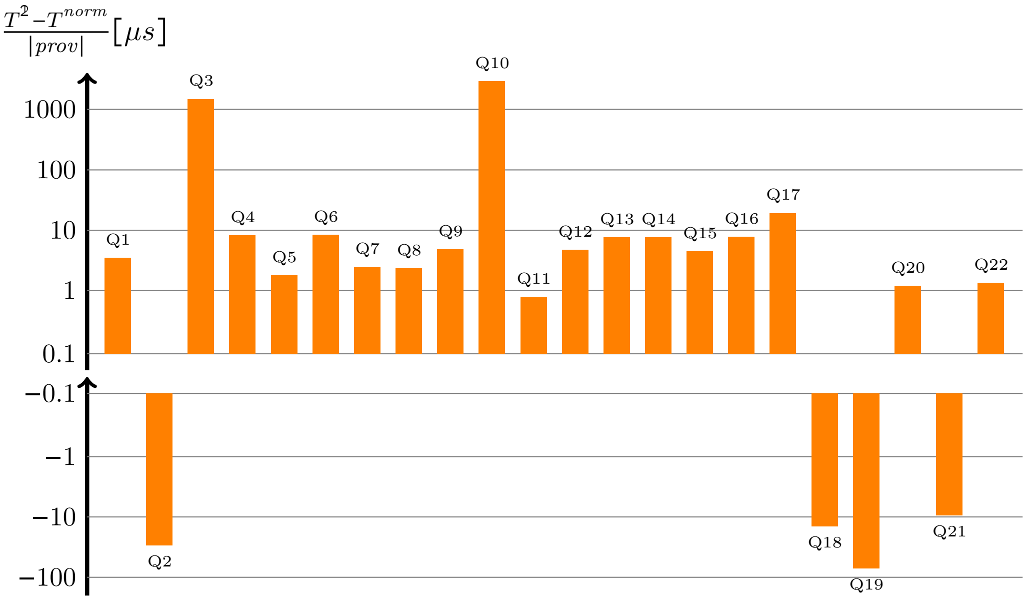


Figure 9.24: **Expended time in phase 2 for evaluating a single provenance identifier. The unit of time is microseconds (μs).**

```
Aggregate  (cost=2004.22..2004.73 rows=1 width=82)
           (actual time=7.326..7.326 rows=1 loops=1)
  ->  Nested Loop  (cost=1.12..2003.01 rows=121 width=82)
                   (actual time=0.184..5.040 rows=121 loops=1)
        ->  Nested Loop  (cost=0.85..2000.32 rows=121 width=250)
                         (actual time=0.042..2.841 rows=121 loops=1)
              ->  Nested Loop  (cost=0.43..1026.17 rows=121 width=154)
                               (actual time=0.031..1.752 rows=121 loops=1)
                    ->  Seq Scan on join2 j
                          (cost=0.00..2.51 rows=121 width=8)
                          (actual time=0.015..0.069 rows=121 loops=1)
                          Filter: ((ℓ)::integer = (1)::integer)
                    ->  Index Scan using lineitem_2_pkey on lineitem_2 l
                          (cost=0.43..8.45 rows=1 width=154)
                          (actual time=0.012..0.013 rows=1 loops=121)
                          Index Cond: (ρ = j.ρ1)
              ->  Index Scan using part_2_pkey on part_2 p
                    (cost=0.42..8.04 rows=1 width=104)
                    (actual time=0.007..0.008 rows=1 loops=121)
                    Index Cond: (ρ = j.ρ2)
        ->  Function Scan on toy _y
              (cost=0.27..0.28 rows=1 width=32)
              (actual time=0.017..0.017 rows=1 loops=121)
```

Figure 9.25: **Query plan of Q19 in phase 2.**

9.5.4 Speedup of Q19

Q19 was a comparably simple query joining the tables part and lineitem together, using a conjunctive WHERE–predicate (≈ 30 lines of SQL code) with many conditions being checked. We provide the full query plan for phase 𝟚 in Figure 9.25. The innermost table getting scanned is **join2** (on line 9) which basically contains the evaluated WHERE predicate. Afterwards, the joins with **lineitem_2** and **part_2** are carried out based on the $\boldsymbol{\rho}$ columns and indices are exploited. This example shows that the query optimizer is fully able to push down the log access (which replaces predicate evaluation). This option does not exist for single–pass query evaluation, since the outcome of predicates is typically not known in advance. The slowdown for Q19 in phase 𝟚 was ≈ 0.20, i.e. it evaluated ≈ 5 times faster than the normalized query.

Line 21 of Figure 9.25 is the plan node for evaluating Why–provenance. If we omitted the derivation of Why–provenance for Q19, the query optimizer would have employed an Index Only Scan for table **part_2**.

9.5.5 Slowdown of Q10

According to Figure 9.24, Q10 has the highest overhead of all queries in phase 𝟚. The issue is specific to the LIMIT clause and occurs for Q3 and Q2 as well. (However for Q2, the quantitative impact is faint.) Directly compared to the discussion of Q19 from above (where the predicate can be pushed downwards), the LIMIT filter does not get pushed in any of the queries, including phase 𝟚.

For phase 𝟚, the query plan of Q10 is listed in Figure 9.26. The GroupAggregate node yields 30780 rows (see line 7) while the LIMIT clause (in phase 𝟚, implemented through log inspection and the Merge Join on lines 3-4) yields only 20 rows. Denoted in selectivity, only 20/30780 = 0.06% of the intermediate rows qualify against LIMIT. (In comparison, Q3 yields 10/10713 = 0.09% and Q2 yields 100/460 = 22%.)

For confirmation, we evaluated a modified set of Q10 queries with LIMIT removed in each of them. Using Equation (9.2), the new overhead is

$$\frac{49.7s - 1.41s}{8076035} = 5.98\mu s$$

. Looking at Figure 9.24, the updated value closes the gap between Q10 and the majority of queries.

```
Nested Loop  (cost=553367.19..693381.18 rows=3078 width=292)
             (actual time=9127.706..38610.508 rows=20 loops=1)
  ->  Merge Join  (cost=553366.94..687101.81 rows=3078 width=260)
                  (actual time=9126.352..38597.259 rows=20 loops=1)
                  Merge Cond: (l_1.ρ_out = j.ρ_out)
        ->  GroupAggregate  (cost=553365.26..686669.21 rows=30780 width=293)
                            (actual time=8948.160..38582.901 rows=37488 loops=1)
        ->  ...
  ->  ...
```

Figure 9.26: **Excerpt: query plan of Q10 in phase $\mathbb{2}$.**

9.6 Optimization of Phase $\mathbb{2}$

The queries in phase $\mathbb{2}$ are dominated by union ($\cup$ and $\bigcup$) operations, processing provenance annotations represented as sets. With PostgreSQL as backend DBMS, we utilized arrays and according concat operations (`||` and `ARRAY_agg()`) instead of set operations. This is a mismatch to some degree. One consequence is that PostgreSQL is unaware of the optimization potential of the union operator. For example,

$$\mathbb{p}_x \cup \mathbb{p}_x = \mathbb{p}_x \tag{9.3}$$

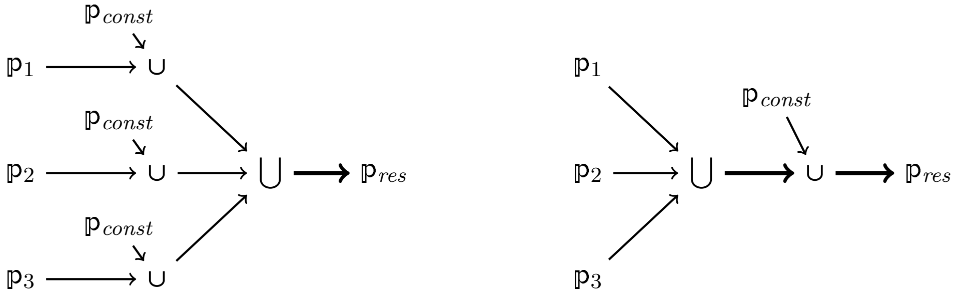
holds for any $\mathbb{p}_x$ but would not hold if translated into array concatenation. The union operator is commutative and associative which offers a lot of optimization potential. The PostgreSQL query optimizer however does not exploit this advantage. In our experiments, we have exploited the following optimization strategy through query rewriting.

9.6.1 $\bigcup$ Optimization

We discuss the poorly optimized phase $\mathbb{2}$–example of Figure 9.27(a). The provenance annotation $\mathbb{p}_{const}$ represents the provenance of a sub–expression in a `WHERE`–predicate. The three input rows $\mathbb{p}_1$, $\mathbb{p}_2$ and $\mathbb{p}_3$ (for presentation reasons, let each row consist of a single provenance annotation) survive the `WHERE`–predicate and get aggregated using $\bigcup$, yielding $\mathbb{p}_{res}$. Take note that the very same $\mathbb{p}_{const}$ contributes to all three individual rows, however two of them turn out as duplicates when $\bigcup$ is applied. The transformation

$$(\mathbb{p}_1 \cup \mathbb{p}_{const}) \cup (\mathbb{p}_2 \cup \mathbb{p}_{const}) \cup (\mathbb{p}_3 \cup \mathbb{p}_{const}) = \mathbb{p}_1 \cup \mathbb{p}_{const} \cup \mathbb{p}_2 \cup \mathbb{p}_3$$

exploits associativity and commutativity of $\cup$ and Equation (9.3). In Figure 9.27(b), the rewritten operator tree is shown. We applied this optimization to the TPC-H queries (on



(a) **Input query.** (b) **Optimized query.**

Figure 9.27: **Example: $\bigcup$ optimization of a phase 𝟚–query.**

SQL level), i.e. relocated entire subqueries (details omitted). This optimization is however not possible if correlated subqueries are involved. Our reported performance results include this very optimization, where applicable. We consider the (serious) optimization of phase 𝟚 future work.

Part IV

Conclusion

10 Related Work

Related work is selected according to criteria provided below.

Language and Data Model In our work, we have contributed a provenance analysis for a feature–rich SQL dialect and the relational model with bag semantics. For example, we cover correlated subqueries and window functions. A number of related works is considerably less expressive. To the best of our knowledge, window functions are not supported by any other work.

Provenance Granularity Generally spoken, provenance analysis tracks pieces of data involved in computations. *Provenance granularity* specifies the size of such pieces. Well–known examples are

- row granularity (very common in RA–based analysis approaches) or
- cell granularity.

We have contributed a provenance analysis in sub–cell–level granularity. An early work of A. Woodruff and M. Stonebraker [WS97] also pursues this goal, discussed below.

Provenance Semantics A *provenance semantics* (also called *provenance notion* or *contribution semantics*) is a set of rules which maps single (abstract) evaluation steps to their corresponding provenance results. For the entire research field, an important contribution is [BKT01] by P. Buneman, S. Khanna and W. Tan. They distinguish between Where–provenance and Why–provenance. Our approach adopts their vocabulary but uses a different semantics, elaborated below.

Implementation Approach In context of the databases field, we observe two entirely different kinds of implementations of provenance analyses: *with* or *without* DBMS support. A modern, fully–fledged DBMS encompasses the result of decades of research in query optimization, indexing, table statistics, caching and so on. If the provenance analysis does not exploit DBMS capabilities, its practical relevance is probably limited.

Implementation Invasiveness Approaches leveraging an existing DBMS (see directly above) may require modifications to the DBMS. These can be more or less invasive. For example,

- modification of the DBMS core (like, physical operators) or
- plug–in installation or
- definition of additional tables and UDFs. Our detached approach belongs to this category.

Presupposed that the source code of the desired DBMS is available, the typical advantage of invasive provenance implementations is improved performance. The typical disadvantage is that the provenance implementation will not be maintained in future DBMS releases and will be outdated sooner or later. P. Senellart, L. Jachiet et al. list examples and argument that such modification

> results in code that cannot be easily maintained or even compiled on modern operating systems. [SJMR18, Sec. 1]

10.1 Fine–Grained Data Lineage

[WS97] by A. Woodruff and M. Stonebreaker is an early work on data provenance (before the term *data provenance* got established). One distinct objective of their work is to derive fine–grained provenance for complex (nested) data structures.

> [...] POSTGRES supports a variety of complex attributes, e.g., arrays, tuple types (in which an attribute may be broken down into a number of other attributes), and user–defined types [...]. [WS97, Sec. 3.1.2]

In our work, we pursue a very similar goal, termed provenance analysis in *sub–cell granularity.*

An essential difference between [WS97] and our work is that their approach for provenance computation is based on *query inversion.* For a query f, they evaluate f^{-1} which yields the original inputs to f, i.e. the data provenance. They consider their approach *lazy* because f^{-1} only gets evaluated if the provenance is actually requested (alternatively, f had to compute and store additional metadata *eagerly*). Of course, f^{-1} does not always exist. For these cases, [WS97] contributes a method to approximate f^{-1}. The considerable downside of this method is that query–specific helper functions have to be provided by the user. In our approach, the user does not have to provide such helper functions nor do we employ query inversion. However, our phase $\mathbb{1}$ may be considered an eager component, slowing down the overall performance of the DBMS.

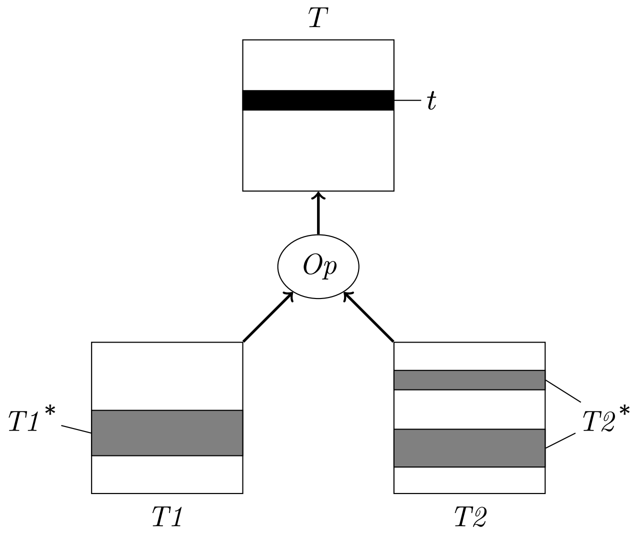


Figure 10.1: **Lineage example with two input tables, reproduced from [CWW00, p.187, Fig. 9].**

10.2 Data Lineage

The work of Y. Cui, J. Widom and J. Wiener [CWW00] is similar to [WS97] (discussed directly above) in employing inverse queries. However, it is more focused and defines concrete Op^{-1} for a number of RA operators *Op*. Also, [CWW00] restricts the provenance analysis to row granularity.

We use the abstract evaluation model from Figure 10.1 to sketch their perspective. Let T be a materialized view, *T1* and *T2* be two intermediate tables and *Op* be an operator of the relational algebra. For a tuple $t \in T$, the data lineage consists of the subsets $T1^* \subseteq T1$ and $T2^* \subseteq T2$. In [CWW00], the notation $Op^{-1}(t)$ is employed to denote the data lineage of t (not to be mixed up with the notation for the inverse function f^{-1} in the mathematical sense). In [CWW00, p.189, Theorem 4.4], they provide the Op^{-1} for selection, projection, theta join, aggregation, union and difference in set semantics. These can be extended for bag semantics ([CWW00, Section 8]).

Example We provide an example in order to directly compare between data lineage and our provenance semantics. The table *sales* (see Figure 10.2(a)) contains revenue data per sales region and article. The example query

$$Q_{lin} := \sigma_{\mathit{sum}>80}(\alpha_{\mathit{article},sum(\mathit{revenue})}(\mathit{sales}))$$

consists of two nested RA operators (selection σ and aggregation α[1]) and finds top–selling articles. The only result tuple is $t_{11} = \langle \mathsf{laptops}, 90\rangle$, listed in Figure 10.2(b). We sketch the lineage computation for t_{11}. In the first step, the (simplified) RA expression $\sigma^{-1}_{sum>80}\mathit{output}$ gets evaluated, yielding t_{12} (shown in Figure 10.2(c)). Take note that the lineage does not include $\langle \mathsf{smartphones}, 60\rangle$, because lineage is different from the inverse. Intuitively, it is clear that the smartphones data did not contribute to t_{11}. In the second step, t_{12} is fed into $\alpha^{-1}_{article,\ldots}$ and yields the lineage shown in Figure 10.2(d) (being a subset of *sales*).

Comparing the example directly with our work, we yield the SQL query and provenance result listed in Figure 10.3. The data provenance contributed through the `GROUP BY`– and `HAVING`–clauses is collected in variable gh and is not interesting for this discussion. The main observation we draw from this example is that t_2 and t_5 are in the results of both approaches.

In contrast, our approach is more fine–grained, i.e. the data provenance not just identifies rows but attributes of rows. For example, the data lineage of t_{11} is $\{t_2, t_5\}$ (see Figure 10.3(b)). Our approach yields references to cells, i.e.

$$\{t^e_{2.revenue}, t^e_{5.revenue}\}$$

for the output value 90 (see Figure 10.3(b)).

Beyond the scope of this example, the two approaches differ in their supported dialects (of RA and SQL). The SQL dialect of this work does not include set difference — we consider this future work (see Section 13.2). Apart from this exception, we consider our dialect considerably more expressive through features like correlated subqueries and window functions.

10.2.1 *WHIPS* Implementation

An implementation of data lineage is provided by Y. Cui and J. Widom [CW00]. The implementation uses the *WHIPS* data warehouse prototype by W. Labio, Y. Zhuge et al. [LZW⁺97] as its backend. *WHIPS* integrates data from multiple sources into the relational model, creating materialized views in the process. These views can be queried and lineage of those queries can be derived. The Op^{-1} are not evaluated directly (since they yield input tuples, no DBMS would implement them naturally) but are converted into so–called *tracing queries* (details omitted). To the best of our knowledge, there are no recent developments involving *WHIPS* and we consider it discontinued.

[1] $\alpha_{\ldots}$ specifies grouping criteria and aggregation functions next to each other. For example, $\alpha_{article}$ creates groups based on *article* and yields one representative of each group (column name implicit).

sales

region	*article*	*revenue*	
North	smartphones	40	t_1
North	laptops	40	t_2
East	TV sets	20	t_3
West	smartphones	20	t_4
West	laptops	50	t_5
South	TV sets	10	t_6

(a) **Input table.**

output

article	*sum*	
laptops	90	t_{11}

(b) **Result of query Q_{lin}.**

lineage-σ

article	*sum*	
laptops	90	t_{12}

(c) **Intermediate lineage (result of σ^{-1}).**

lineage

region	*article*	*revenue*	
North	laptops	40	t_2
West	laptops	50	t_5

(d) **Data lineage of Q_{lin}.**

Figure 10.2: **Example: data lineage according to [CWW00].**

```
       FROM sales AS s
      WHERE TRUE
   GROUP BY s.article
 AGGREGATES THE(s.article) AS article,
            SUM(s.revenue) AS sum
     HAVING aggs.sum > 80
    WINDOWS ()
     SELECT aggs.article AS article,
            aggs.sum AS sum
   ORDER BY ()
DISTINCT ON ()
     OFFMIT NULL NULL
```

(a) **Query Q_{lin} in SQL.**

detached⇧

article	*sum*
◀laptops, $\{t^e_{2.article}, t^e_{5.article}\} \cup gh$▶	◀90, $\{t^e_{2.revenue}, t^e_{5.revenue}\} \cup gh$▶

(b) **Data provenance of Q_{lin}.**

Figure 10.3: **The example from Figure 10.2 in the detached approach.**

sales

region	article	revenue	
North	laptops	40	t_1
West	laptops	50	t_2
South	TV sets	10	t_3

(a) **Database.**

output

article	
laptops	t_4
TV sets	t_5

(b) **Query result:** $\pi_{article}(sales)$**.**

Figure 10.4: **Example data for the discussion of Why–provenance.**

10.3 Why– and Where–Provenance

The contribution of P. Buneman, S. Khanna and W. Tan [BKT01] is a highly influential work for the research field. They distinguish between two provenance semantics (also called: *provenance notions*). One of them is called *Why–provenance* (discussed next) and the second is *Where–provenance* (discussed below). The formalism used in [BKT01] is based on a deterministic data model paired with a specific query language. We skip that formalism and instead refer to [CCT09] by J. Cheney, L. Chiticariu and W. Tan who redefine the contributions of [BKT01] using relational algebra.

10.3.1 Why–Provenance

Why–provenance can be considered a refinement of data lineage (see above). Why–provenance and data lineage both share row–level granularity but Why–provenance provides *multiple, alternative explanations* for a given output tuple. We are going to exemplify using the data (see Figure 10.4) and query $Q \coloneqq \pi_{article}(sales)$ (in set semantics). The focus is on output tuple t_4. t_4 is an interesting tuple, since it corresponds to the input tuples t_1 and t_2.

The **data lineage** (using set notation according to [CCT09, p. 20, Theorem 2.1]) is

$$\{t_1, t_2\}$$

, i.e. both laptops–valued input tuples are considered relevant. This means that there is some redundancy in tuples t_1 and t_2 (one of them in the input table would be sufficient to yield t_4). An important contribution of [BKT01] is to understand data provenance as a collection of witnesses, each witness being sufficient to yield a certain output row. In our example, the **witnesses** (according to [CCT09, p. 29, Definition 2.3]) are

$$\{\{t_1\}, \{t_1, t_2\}, \{t_1, t_2, t_3\}, \ldots\}$$

sales

region	*article*	*revenue*	
North$^{\square}$	laptops^{+}	40$^{\square}$	t_1
West$^{\square}$	laptops^{-}	50$^{\square}$	t_2
South$^{\square}$	TV sets*	10$^{\square}$	t_3

(a) **Database.**

output

article	
laptops$^{+,-}$	t_4
TV sets*	t_5

(b) **Query result: $\pi_{\mathit{article}}(\mathit{sales})$.**

Figure 10.5: **Example for Where–provenance: the annotations $(+, -, *)$ are being propagated to the output table. The $\square$ annotations are ignored.**

. One of the witnesses (=$\{t_1, t_2\}$) happens to be just the data lineage from above. Next to it, smaller witnesses exist (e.g., $\{t_1\}$) and less specific witnesses (e.g., $\{t_1, t_2, t_3\}$). A counter–example would be $\{t_3\}$ which is not a witness for t_4. For this example (according to [CCT09, p. 30, Definition 2.4])), the Why–provenance is

$$\{\{t_1\}, \{t_3\}\}$$

. In direct comparison to data lineage ($\{\{t_1\}, \{t_3\}\}$), Why–provenance provides more detail, i.e. identifies the two witnesses sufficient for computing t_4.

10.3.2 Where–Provenance

Where–provenance is the second provenance notion introduced in [BKT01] with the purpose to distinguish Where–provenance from Why–provenance. The main idea of Where–provenance is to determine where the data value of a certain output cell has been copied from. Where–provenance has cell granularity (different from Why–provenance). In subsequent work [BKT02, Sec.3] (of the same authors), annotation propagation rules for RA operators are provided. Figure 10.5 exemplifies this process for the π operator (in set semantics). The annotated output value laptops$^{+,-}$ indicates that two input values (with annotations + and −) have contributed.

10.3.3 Comparison

Next, a comparison of Where– and Why–provenance (according to [BKT01]) with the provenance semantics of this work is carried out. Since we have introduced different semantics for Where– and Why–provenance, there is a naming conflict. For the following discussion, sales⇧

region	article	revenue	
◀North, $\{t^{e}_{1.\mathit{reg}}\}$▶	◀laptops, $\{t^{e}_{1.\mathit{art}}\}$▶	◀40, $\{t^{e}_{1.\mathit{rev}}\}$▶	t_1
◀West, $\{t^{e}_{2.\mathit{reg}}\}$▶	◀laptops, $\{t^{e}_{2.\mathit{art}}\}$▶	◀50, $\{t^{e}_{2.\mathit{rev}}\}$▶	t_2
◀South, $\{t^{e}_{3.\mathit{reg}}\}$▶	◀TV sets, $\{t^{e}_{3.\mathit{art}}\}$▶	◀10, $\{t^{e}_{3.\mathit{rev}}\}$▶	t_3

(a) **Database.**

```
       FROM sales AS s
      WHERE TRUE
   GROUP BY ()
 AGGREGATES ()
     HAVING ()
    WINDOWS ()
     SELECT s.article AS article
   ORDER BY ()
DISTINCT ON s.article
     OFFMIT NULL NULL
```

(b) **SQL query.**

output⇧

article	
◀laptops, $\{t^{e}_{1.\mathit{art}}, t^{y}_{1.\mathit{art}}\}$▶	t_4
◀TV sets, $\{t^{e}_{3.\mathit{art}}, t^{y}_{3.\mathit{art}}\}$▶	t_5

(c) **Output table.**

Figure 10.6: **The running example in the detached approach.**

- Where– and Why–provenance denote the semantics according to [BKT01] and
- e– and y–provenance denotes our definitions.

One novel aspect of our approach is that e– and y–provenance are designed to fit together. Both notions share the same granularity. The examples from above are continued in Figure 10.6 (with RA translated into SQL) and data provenance according to the detached approach is derived. All provenance results for output value laptops are aggregated below.

Provenance Notion	Result
Where	$\{t_{1.\mathsf{art}}, t_{2.\mathsf{art}}\}$
Why	$\{\{t_1\}, \{t_2\}\}$
e and y	$\{t^e_{1.\mathsf{art}}, t^y_{1.\mathsf{art}}\}$

- According to [CCT09, Sec. 5.3], Where–provenance is contained in Why–provenance. The example confirms that. In contrast, e– and y–provenance are not.
- Where–provenance models copy operations of values. e–provenance is more general and covers value transformations.
- Why–provenance yields multiple explanations for an output value. Therefore, it is less susceptible to (value–equivalent) query reformulation. y–provenance yields a single explanation and is susceptible to query reformulation.
- y–provenance cannot be computed without e–provenance. Whenever a predicate is evaluated, the corresponding e–provenance is transformed into y–provenance. In contrast, Why–provenance is less specific: it has row granularity and identifies all rows which have been involved in a query evaluation.

10.3.4 *DBNotes* Implementation

D. Bhagwat, L. Chiticariu et al. contributed [BCTV05] which is basically an implementation of Where–provenance in cell–level granularity. They employ a dialect called pSQL which consists of SFW expressions

```
SELECT DISTINCT selectlist
           FROM fromlist
          WHERE wherelist
      PROPAGATE options
```

and can combine multiple SFW expressions using `UNION`. Aggregates, subqueries or bag semantics are not supported ([BCTV05, Definition 1]). The additional `PROPAGATE` clause allows the user to switch between different flavors of Where–provenance (not discussed here). Queries formulated in pSQL get translated into SQL (enriched with logic for annotation propagation) and delegated to a DBMS (Oracle 9i Enterprise Edition) for execution. Basically, the annotations sit in additional columns (one annotation column for each data column). A single annotation may consist of multiple items (comparable to our Definition 5.1) — implemented through duplicate rows (we use SQL arrays in our implementation).

Although not pointed out by the authors, *DBNotes* appears to be a non–invasive implementation regarding to the DBMS core — very much like our own contribution. There is a more detailed figure in [CTV05, Figure 3] which clearly suggests that the *DBNotes* implementation is wrapped around the DBMS, managing the inputs and outputs to/from the DBMS. Unlike to pSQL, the SQL dialect of our work basically allows nesting anywhere (subqueries and data) and supports more recent SQL features like window functions.

10.4 Semirings Approach

The approach of T. Green, G. Karvounarakis and V. Tannen [GKT07] annotates rows with elements from semirings (hence row–level granularity). The concrete semiring determines the provenance semantics. We are going to exemplify two concrete semirings. The abstract notation of a semiring is $(K, \oplus, \odot, \textcircled{0}, \textcircled{1})$ where

- K is a set (its elements representing annotations),
- $\oplus$ is a binary operation with neutral element $\textcircled{0}$ and
- $\odot$ is a binary operation with neutral element $\textcircled{1}$.

[GKT07, Definition 3.2] formally specifies how concrete RA operators translate into operations in the (abstract) semiring. The following two examples have been simplified. Let R be an intermediate table (in set semantics) with annotated rows, in symbols $\forall r \in R : \mathsf{annot}(r) \in K$.

- For projections $\pi_{\square}(R)$, and rows $r_{i=1..n} \in R$ equal under projection and yield r', the data provenance is $\mathsf{annot}(r') \coloneqq \textcircled{\Sigma}_{i=1..n}\mathsf{annot}(r_i)$. $\textcircled{\Sigma}$ denotes the n–ary kind of $\oplus$ from above.
- For natural joins $L \bowtie R$ and rows $l_{i=1..n} \in L$ $r_{k=1..m} \in R$, yielding (a subset of) the joined rows $lr_{i,k}$, the data provenance is $\mathsf{annot}(lr_{i,k}) \coloneqq \mathsf{annot}(l_i) \odot \mathsf{annot}(r_k)$.

The concrete semiring determines the (provenance) semantics. Directly below, we provide two simplified examples from [GKT07].

- The counting semiring $(\mathbb{N}, +, \cdot, 0, 1)$ implements bag semantics.
- The polynomial semiring $(\mathbb{N}[T], +, \cdot, 0, 1)$ implements so–called *How–provenance*; the elements of $\mathbb{N}[T]$ are polynoms and draw variables from T and coefficients from $\mathbb{N}$.

We use the example database from Figure 10.7(a) with two tables. The database contains planets of the solar system and visits (planetary flyby maneuvers) of the voyager space probes. Each row has (for compactness of presentation) two annotations in parallel, corresponding to the two semirings introduced above. The initial annotations of the counting semiring (column N) are row counts, i.e. the value 1. The polynomial semiring (column: $\mathbb{N}[T]$) has unique annotations $t_\square$ drawn from T.

The intermediate table *projected* (see Figure 10.7(b)) is the result of projection on *id*. The counting semiring keeps track of duplicate rows (the rows with *id*=5 and *id*=6 would appear twice in bag semantics). The provenance semiring (column $\mathbb{N}[T]$) lists the actual input tuples used in the projection.

The final table *joined* is the result of $\pi_{id}(visits) \bowtie planets$. The annotations of the counting semiring are not interesting (tuple counts did not change) but the provenance semiring yields real insight.

> Intuitively, "+" corresponds to alternative use of data in producing a query result, while " · " corresponds to joint use. [KG12, p. 5]

For the output row containing Jupiter, the data provenance is $(t_{11} + t_{13}) \cdot t_5$ which implies that t_5 is mandatory but one of t_{11} or t_{13} would be sufficient.

Hence, the polynom representation of data provenance not only lists the contributing input tuples but also provides insight regarding to *how* these input tuples have been combined. In [GKT07], the term *How–provenance* has been suggested to denote this new kind of data provenance. In [CCT09, Sec. 5.1] it has been shown that the semirings approach generalizes both data lineage ([CWW00]) and Why–provenance ([BKT01]). In [KG12, Sec. 4.1] G. Karvounarakis and T. Green relate even more provenance semantics to each other with the polynomial semiring sitting on top of the hierarchy.

10.4.1 Comparison of Data Provenance

Figure 10.8(a) lists the query from above in SQL and Figure 10.8(b) shows the output table with data provenance computed in the detached approach. In comparison,

- the detached approach yields data provenance in (sub–) cell–level granularity and is more insightful in this aspect. For example, the result ◀Jupiter, $\{t^e_{5.planet}, t^y_{13.id}, t^y_{5.id}\}$▶ exposes exactly the values $t^y_{13.id}$ and $t^y_{5.id}$ having been inspected in predicates.

planets

id	*planet*	$\mathbb{N}$	$\mathbb{N}[T]$
1	Mercury	1	t_1
2	Venus	1	t_2
3	Earth	1	t_3
4	Mars	1	t_4
5	Jupiter	1	t_5
6	Saturn	1	t_6
7	Uranus	1	t_7
8	Neptune	1	t_8

visits

id	*probe*	*year*	$\mathbb{N}$	$\mathbb{N}[T]$
5	Voyager 1	1979	1	t_{11}
6	Voyager 1	1980	1	t_{12}
5	Voyager 2	1979	1	t_{13}
6	Voyager 2	1981	1	t_{14}
7	Voyager 2	1986	1	t_{15}
8	Voyager 2	1989	1	t_{16}

(a) **Input tables.**

projected

id	$\mathbb{N}$	$\mathbb{N}[T]$
5	2	$t_{11} + t_{13}$
6	2	$t_{12} + t_{14}$
7	1	t_{15}
8	1	t_{16}

(b) $\pi_{id}(visits)$

joined

id	*planet*	$\mathbb{N}$	$\mathbb{N}[T]$
5	Jupiter	2	$(t_{11} + t_{13}) \cdot t_5$
6	Saturn	2	$(t_{12} + t_{14}) \cdot t_6$
7	Uranus	1	$t_{15} \cdot t_7$
8	Neptune	1	$t_{16} \cdot t_8$

(c) $\pi_{id}(visits) \bowtie planets$

Figure 10.7: **Example: space probes and the semirings approach.**

- On the other hand, the semirings approach provides insight in *how* tuples have been combined (i.e., $(t_{11} + t_{13}) \cdot t_5$) to produce the result. Accordingly, the semirings approach also provides more insight.

- Take note that t_{13} and t_5 appear in both results but t_{11} does not. This is because only one of the two tuples (t_{13} or t_{11}) is sufficient to produce the result. The provenance derived by the detached approach exposes the decision of the DBMS while the semirings unwraps all alternative ways of computation. If implemented using a DBMS backend, we expect a lower runtime overhead for the detached approach.

Comparison of the Language Dialects Our SQL dialect is considerably more expressive than the RA dialect of the semirings approach. The original contribution ([GKT07]) is based on an RA dialect called positive relational algebra (RA$^+$), basically consisting of union, projection, selection and natural join. For this dialect, the semirings approach

```
      FROM visits AS v,
           planets AS p
     WHERE v.id=p.id
  GROUP BY ()
AGGREGATES ()
    HAVING TRUE
   WINDOWS ()
    SELECT v.id AS id,
           v.planet AS planet
  ORDER BY ()
DISTINCT ON v.id
    OFFMIT NULL NULL
```

(a) **SQL query.**

joined⇧

id	*planet*
◀5, $\{t^{e}_{13.id}, t^{y}_{13.id}, t^{y}_{5.id}\}$▶	◀Jupiter, $\{t^{e}_{5.planet}, t^{y}_{13.id}, t^{y}_{5.id}\}$▶
◀6, $\{t^{e}_{14.id}, t^{y}_{14.id}, t^{y}_{6.id}\}$▶	◀Saturn, $\{t^{e}_{6.planet}, t^{y}_{14.id}, t^{y}_{6.id}\}$▶
◀7, $\{t^{e}_{15.id}, t^{y}_{15.id}, t^{y}_{7.id}\}$▶	◀Uranus, $\{t^{e}_{7.planet}, t^{y}_{15.id}, t^{y}_{7.id}\}$▶
◀8, $\{t^{e}_{16.id}, t^{y}_{16.id}, t^{y}_{8.id}\}$▶	◀Neptune, $\{t^{e}_{8.planet}, t^{y}_{16.id}, t^{y}_{8.id}\}$▶

(b) **Resulting data provenance.**

Figure 10.8: **Example: space probes and the detached approach.**

is very elegant (allowing to switch easily between different provenance notions) and can deliver very insightful results (exemplified above for the polynomial semiring). However, the expressiveness of RA^+ is very limited. In subsequent work, the boundaries of RA^+ got extended.

- In [GP10], F. Geerts and A. Poggi have introduced m–semirings (=semirings with monus $\ominus$) making it possible to add *difference* to the RA dialect. Not all semirings can be converted into m–semirings.
- In [ADT11], Y. Amsterdamer, D. Deutch and V. Tannen have concluded that row–level annotations are not sufficient for (sensible) support of *aggregation* and propose additional annotations for individual values.

10.4.2 ProvSQL

Recent work by P. Senellart et al. [SJMR18] implements the semirings approach and integrates additional features like Where–provenance and m–semirings. The SQL dialect is limited. For example, aggregations are not supported but `UNION` and `EXCEPT` are.

10.5 Traces Approach

J. Cheney, A. Ahmed and U. Acar [CAA14] introduce a two–step approach in deriving data provenance for NRC queries with bag semantics. In the first step, the query Q (accessing database Δ) is executed. In an augmented evaluation (defined for each expression of the NRC), Q yields an output table v and a trace T. T basically is a record of all evaluation steps and data being accessed for turning Δ into v. Accordingly, the size of T can be considerable since it may contain the entire database Δ. In the second step, a (user–defined) pattern p is applied to T. The pattern denotes the user's desire in which part of v is of interest. Applying p to T is similar to a filter operation. The result of this second step are a Δ' and T', capturing the relevant input data and relevant query expressions for evaluating v. The purpose of traces is considerably different from the logs we employ in the detached approach. Logs basically contain predicate decisions but traces contain payload data.

10.6 Smoke Approach

F. Psallidas and E. Wu recently contributed [PW18a] (and the technical report [PW18b]). *Smoke* is a prototype of a query processor with built–in support for provenance analysis. Their work provides a contrast to our own research focus.

```
xs = [-2, -1, 42]

res = []
foreach x in xs do
    if writeLog(x >= 0) then
        append(res, x)
    fi
od

// res: [42]
```

(a) **Phase 𝟙.**

```
xs = [𝕡1, 𝕡2, 𝕡3]

res = []
foreach x in xs do
    if readLog() then
        append(res, x)
    fi
od

// res: [𝕡3]
```

(b) **Phase 𝟚.**

Figure 10.9: **Example: detached provenance analysis for imperative programs.**

- In *Smoke*, provenance support is built directly into physical operators. If their approach should be adopted for an existing DBMS, it would require modifications to the DBMS kernel. The focus of our work is to enable provenance analysis for existing, unmodified DBMS implementations.
- The language features are limited. The focus of their optimization is on SPJA queries. More operators are provided in the technical report. Their experimental evaluation covers 4 uncorrelated and simplified TPC-H queries (omission of the clauses `ORDER BY` and `LIMIT`). The focus of our work is on a feature–rich SQL dialect.

One distinct feature of the *Smoke* approach is that it uses indices to speed up provenance computation. The fundamental idea is that each concrete RA operator (together with concrete payload data) maintains an index which maps the relationship of input rows to output rows. Conceptually, this makes the first evaluation of an operator more expensive but follow–up evaluations cheap. Hence, [PW18a] promotes interactive visualizations of data provenance as a use case. Also, these indices may be considered a generic form of data provenance and allow for a refinement in a second stage. [PW18b][Appendix E.] provides examples for how to switch between lineage, Why–provenance and How–provenance.

10.7 The Detached Approach and Imperative Languages

Originally, the detached provenance analysis has been developed for imperative languages. T. Müller and T. Grust contributed a demonstration [MG15] which involved a (very basic) prototype of a query–compiling DBMS[2]. First, the SQL query gets translated into an imperative program P and then, the two phases are carried out.

- Phase 𝟙 evaluates P and creates a log.
- Phase 𝟚 inspects the log and propagates provenance annotations.

On this high–level perspective, the two phases for analyzing an imperative program are identical with the detached provenance analysis on SQL level (see Chapter 8).

We are going to exemplify their approach for a simple, imperative program in Python–inspired [Pyt] pseudocode. The program listed in Figure 10.9(a) (phase 𝟙) iterates through a list and applies a filter (comparable to a simple SFW expression). Input data is provided on line 1 and the `append(res, x)` function inserts the qualified list element(s) `x` at the end of the resulting list `res`. On line 5, a logging function has been inserted. The predicate gets evaluated and `writeLog(x >= 0)` records the stream of booleans

```
[false, false, true]
```

required for phase 𝟚. After logging, the boolean is forwarded to the surrounding `if` (from `if` perspective, the logging function behaves like the identity).

In phase 𝟚 (see Figure 10.9(b)), the very same log gets inspected (see line 5) and is being used to filter the provenance annotations $\mathbb{p}_{\square}$ without looking at the original values. In comparison to the detached approach for SQL, the log is ordered and gets read/written from left to right. The example presented here is very basic and only logs a (compact) stream of bits. While very appealing at first sight, in our publications (also see [Mül16] which uses more formalism and a bigger dialect) the logs require other data types as well. In the end, $\mathbb{p}_3$ is found as the data provenance of list element 42 (see the lines 10 of both phases).

Comparison

- Implementing the provenance analysis on SQL level is more flexible. In theory, rewritten SQL queries can be fed to any DBMS which supports SQL. (In practice, each DBMS implements a slightly different SQL dialect.)
- Compiled queries are optimized already. Modifications (e.g. log reading and writing) would not be able to profit from optimization. Also, the modifications of phase 𝟚 (change of data types into provenance annotations) would interfere with planned cache sizes.

[2] Query compilation is a design principle for databases ([Neu11]) and the opposite of query interpretation.

- The research sketched above (detached provenance analysis using query compilation) is still at an early stage, involving prototypes without any query optimization. In comparison, the detached provenance analysis on SQL level just forwards the rewritten queries to a backend DBMS and gets the query optimization for free.

10.8 *GProM*

GProM (Generic Provenance Middleware) by B. Arab, S. Feng et al. [AFG+18] is termed middleware for its low–invasive implementation approach. *GProM* connects to a DBMS backend and carries out query rewriting in order to derive data provenance. In that consideration, *GProM* is very similar to our own approach. Additional features of *GProM* are provenance analysis for transactional updates and Why–not provenance.

In our experimental comparison, we focus on *PERM* which may be considered a predecessor of *GProM*. However, only *PERM* has support for correlated subqueries.

11 Comparison with *PERM*

This chapter consists of two parts. We discuss

- how to implement a provenance analysis in *row granularity* using the detached approach and
- present an experimental comparison with *PERM* [GA09a].

11.1 Implementation of Row–Level Granularity

In Section 9.1, we outlined an implementation of the detached provenance analysis in cell granularity. With a few changes, a provenance analysis in row granularity can be implemented. Phase 𝟙 stays entirely the same (logs can be used for *both* granularities) while phase 𝟚 requires changes to its base tables and queries. The main idea is to restrict any (intermediate) table to two columns.

11.1.1 Tables of Phase 𝟚

- In cell granularity, base tables have one ρ column and n columns with n provenance annotations.
- In row granularity, base tables have one ρ column and a single column with provenance annotations. This makes two columns per table.

The table definition for **`lineitem𝟚_row`** is exemplified in Figure 11.1. This definition is analogous to Figure 9.10. The original `lineitem` table (specified by the benchmark) has 16 columns. The new table **`lineitem𝟚_row`** has only two columns and saves an according amount of disk space. Table `lineitem𝟙` is kept.

11.1.2 Queries of Phase 𝟚

Consistent with the updated base tables, the phase 𝟚 queries produce result tables with two columns. As an example, Figure 11.2(a) lists a subquery of Q19 which specifies the columns $\boldsymbol{\rho}$ and **`prov`** on lines 2-5. Where– and Why–provenance share a common domain

```
-- initialize a sequence for provenance annotations
CREATE SEQUENCE $A_{seq}$;

-- phase 𝟚 tables
CREATE TABLE lineitem𝟚_row AS
    SELECT
        ρ AS ρ,
        ARRAY[nextval('$A_{seq}$')::int] AS prov
    FROM
        lineitem𝟙;

-- phase 𝟚 indices
ALTER TABLE lineitem𝟚_row ADD PRIMARY KEY(ρ);
```

Figure 11.1: **Excerpt: definition and population of tables for row granularity provenance analysis.**

(do not employ `towhy()`) and cannot be distinguished from each other in the final result. This makes the resulting data provenance better comparable to *PERM*.

11.1.3 Conclusion

Switching to row–level granularity is straightforward for phase 𝟚. It may appear wrong to just drop some columns of base tables but all predicates get evaluated in phase 𝟙. In phase 𝟚, evaluation of predicates is replaced with log inspection.

11.2 PERM

We put our work in the perspective of the *PERM* approach by B. Glavic and G. Alonso [GA09a, GA09b] . Its implementation (also called *PERM*) is available as a modified PostgreSQL server [Gla]. To the best of our knowledge, only *PERM* and the detached approach can analyze correlated SQL queries. The experimental evaluation covers all 22 queries of the TPC-H benchmark [TPC].

11.2.1 Introductory Example

Like our own approach, *PERM* employs annotation propagation and query rewriting in order to derive data provenance. Annotations are represented as additional columns and rows. Per convention, columns containing provenance are prefixed with `prov_`. The next example is based on table `part` from the TPC-H benchmark. The table `part`$^+$ (the

```
SELECT
    log.ρ AS ρ,
    l.prov ∪          -- Where-provenance
      l.prov ∪ p.prov -- Why-provenance
      AS prov
FROM
    lineitem𝟚_row AS l,
    part𝟚_row AS p,
    readjoin(1, l.ρ, p.ρ) AS log(ρ)
```

(a) **Row granularity.**

```
SELECT
    log.ρ AS ρ,
    y.why ∪ l.l_extendedprice AS l_extendedprice,
    y.why ∪ l.l_discount AS l_discount
FROM
    lineitem𝟚 AS l,
    part𝟚 AS p,
    readjoin(1, l.ρ, p.ρ) AS log,
    towhy(  p.p_partkey
          ∪ l.l_partkey
          ∪ p.p_brand
          ∪ p.p_container
          ∪ l.l_quantity
          ∪ p.p_size
          ∪ l.l_shipmode
          ∪ l.l_shipinstruct
          ) AS y(why)
```

(b) **Cell granularity.**

Figure 11.2: **Excerpts of Q19, phase 𝟚: provenance analysis in different granularities.**

part$^+$

p_partkey	*prov_p_partkey*	*p_name*	*prov_p_name*	...
1	1	goldenrod...	goldenrod...	

Figure 11.3: **Example: an annotated table in *PERM*.**

$^+$ denotes provenance derivation) listed in Figure 11.3 has the additional provenance columns prov_p_partkey and prov_p_name. Data provenance is represented literal and in the example (a base table) the provenance is just a copy of the existing columns. During query evaluation, the data provenance gets propagated towards the result table.

Q19 in *PERM* The *PERM* approach provides rewrite rules for relational algebra supporting both set and bag semantics. Q19's simplified RA expression is listed below. It consists of base tables, a theta join and an aggregation.

$$Q19 = \alpha_{sum(\dots)}(\texttt{lineitem} \bowtie_p \texttt{part})$$

Using the rewrite rules **R1** (base table), **R4** (cross join) and **R5** (aggregation) provided in [GA09a][III.C], the (simplified) RA expression

$$Q19^+ = (\alpha_{sum(\dots)}(\texttt{lineitem} \bowtie_p \texttt{part})) ⟕_{true} (\pi_{\mathsf{prov\_}*}(\texttt{lineitem}^+ \bowtie_p \texttt{part}^+))$$

is derived. The expression on the left–hand side of the left outer join ($⟕_{true}$) is a copy of the input expression Q19 and evaluates to the same (value–only) result. On the right–hand side, the data provenance is evaluated.

- $\texttt{lineitem}^+$ and $\texttt{part}^+$ are the provenance–augmented base tables. $\texttt{part}^+$ is exemplified in Figure 11.3.
- The theta join of Q19 is basically kept, however its subexpressions are rewritten. This yields the expression $(\texttt{lineitem}^+ \bowtie_p \texttt{part}^+)$.
- The right–hand side of the left outer join is supposed to produce nothing but data provenance. For this purpose, the projection $\pi_{\mathsf{prov\_}*}(\cdot)$ removes all non–provenance columns.

When the outer join is evaluated, the result value and its data provenance get joined together. Q19 produces a single result cell (see Figure 11.4(a)) while $Q19^+$ has 1 + 16 + 9 columns. These columns contain the value (1), the data provenance regarding to lineitem (16) and the data provenance regarding to part (9). A fragment of the output table is listed in Figure 11.4(b). The full table has 121 rows and 26 columns. The result value 3083843.0578 is repeated 120 times in order to yield a normalized table. The difference in cell counts between Q19 and $Q19^+$ is calculated in *rows · columns* and we find $(121 \cdot 26) - (1 \cdot 1) = 3145$. This is called the *provenance representation size* of *PERM*.

output

revenue
3083843.0578

(a) **The value–only result of $Q19$.**

output^+

revenue	*prov_p_name*	...
3083843.0578	□	...
3083843.0578	□	...
⋮	⋮	⋮

(b) **The provenance result of $Q19^+$ (excerpt).**

Figure 11.4: **Q19's result in the *PERM* approach.**

output𝟙

ρ	*revenue*
1	3083843.0578

(a) **Phase 𝟙.**

output𝟚

ρ	*prov*
1	$[\square, \square, \ldots]$

(b) **Phase 𝟚.**

Figure 11.5: **Q19's result in the detached approach.**

Q19 in the Detached Approach In the row granularity version of the detached approach (introduced in Section 11.1), the two output tables of Figure 11.5 are produced.

- The provenance result is aggregated in a single array value. *PERM* grows the output table in rows and columns instead.
- A single provenance identifier □ (we implemented it as 4–byte integer values) refers to an entire tuple in the input table. *PERM* literally copies the entire input tuple.
- For Q19, the array length (in elements) is 224. We call this the *provenance representation size* in the detached approach. (We do not integrate the ρ columns in the provenance representation size. For TPC-H, they are typically orders of magnitude smaller than the array sizes.)
- The table cardinality of $Q19^+$ in *PERM* is 121. Each row references two tuples, one of each base table (modeling a two–way–join). This corresponds to a total of $121 \cdot 2 = 242$ referenced tuples. The detached approach uses duplicate elimination, therefore duplicates in join partners are removed ($224 < 242$).

11.2.2 Provenance Semantics

Both approaches (*PERM* and the detached approach) support multiple provenance semantics. We configured *PERM* with the default *PI-CS* setting (short for: *PERM* influence contribution semantics).

> [...] we developed *Perm-Influence contributions semantics* (*PI-CS*), a new type of *contribution semantics* based on *Lineage*, that uses a different representation of provenance and solves the problems of *Lineage* concerning negation and nested sub-queries. [Gla10, Sec. 1.2.1]

In turn, the detached provenance analysis got adapted to row granularity (the same granularity as *PI-CS*) and the transformation of Where–provenance into Why–provenance is omitted (exemplified in Figure 11.2). We consider the two provenance semantics unequal but related. The example involving Q19 (see directly above) confirms the relationship between the two provenance semantics. More examples get discussed below. It is not feasible to carry out a formal comparison of the two semantics in the context of this work.

11.3 Experiments

11.3.1 Setup

Both experiments (detached approach and *PERM*) were carried out in the very same environment of hardware and operating system (described in Section 9.2). The source code of *PERM* was retrieved from [Gla] (last updated on December, 9th 2017) and compiled with optimization level `-O2`. *PERM* is based on a PostgreSQL 8.3 DBMS. We took care in a fair configuration of the PostgreSQL servers (despite different versions) and table schemas (with indices). *PERM*–specific optimization flags (suggested B. Glavic) were set.

11.3.2 Comparison of Provenance Representation Sizes

Figure 11.6 shows the results for the provenance representation sizes in both approaches. The detached approach in row granularity (●) consistently has a smaller representation size than perm (■). For example, the sizes of Q19 (discussed above) have the ratio $3,145/224 = 14.0$. The geometric mean for the ratios of all queries (except Q17 and Q20 which did not finish) is 37.2.

11.3.2.1 Discussion of Q22

The experiments identify the biggest gap between representation sizes for Q22. In absolute numbers, *PERM* yields a representation size of ≈ 7 billion while the detached approach only yields $\approx 300,000$. This constitutes a factor of $\approx 23,300$ between the two representation sizes. We have identified two reasons (i.e., $27 \cdot 850 \approxeq 23,300$) for this outcome. Factor 27 corresponds (roughly) to the 25 additional result columns used for

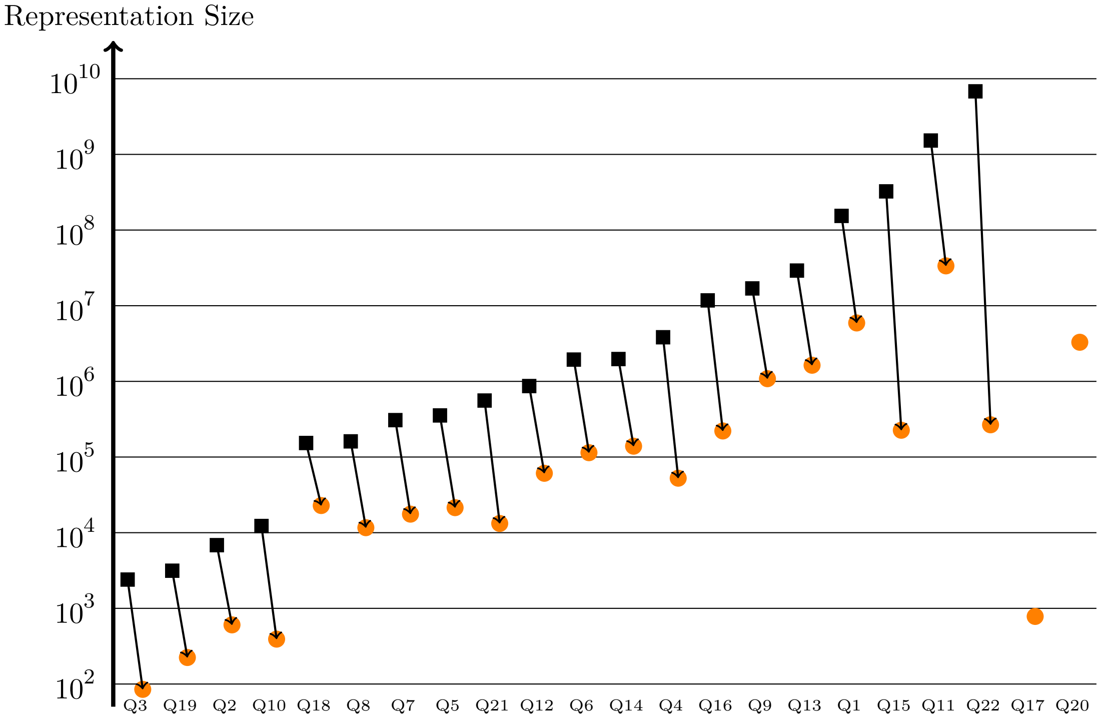


Figure 11.6: **Provenance representation sizes in the detached approach (●) and *PERM* (■).**

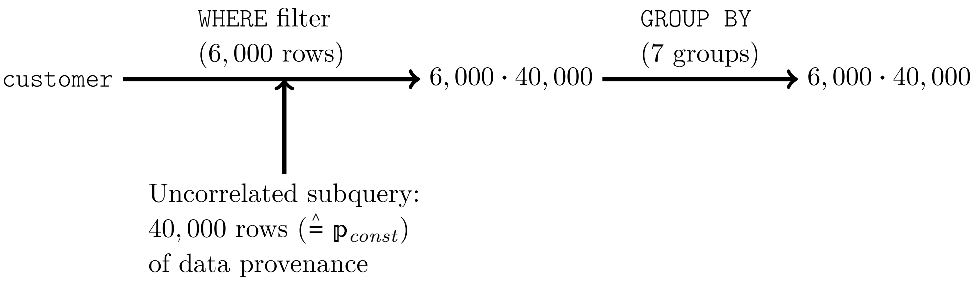


Figure 11.7: **Abstract perspective of provenance computation for Q22 in *PERM*.**

provenance representation in *PERM*'s result table. In contrast, the detached approach represents data provenance in a single column (see the example of Q19 from above).

The reasoning for factor 850 is sketched in Figure 11.7. Starting out from a scan of `customer` (cardinality: $1,500,000$), a `WHERE`–filter is applied. The relevant part of the predicate is an uncorrelated subquery which yields a data provenance of $\approx 40,000$ rows — a single annotation $\mathbb{p}_{const}$ in the detached approach. When the `WHERE`–filter is applied, each of the qualified $6,000$ tuples from `customer` gets annotated with the subquery's provenance, which makes $6,000 \cdot 40,000 = 240,000,000$ rows in total. In the last processing step, `GROUP BY` and aggregation functions are applied which affect the value semantics while the data provenance is just being forwarded. In the end, there are $\approx 240,000,000$ rows (= 7 billion representation size). In each of the 7 groups, the $40,000$ annotations are now represented

$$\frac{6,000}{7} = 850$$

times. The detached approach considers this as redundancy. Each group gets aggregated into a single provenance annotation and duplicates are eliminated. The 850 $\mathbb{p}_{const}$ (per group) are then combined according to

$$\bigcup_{i=1}^{850} \mathbb{p}_{const} = \mathbb{p}_{const}$$

because of duplicate elimination. Altogether, the representation size in the detached approach is $\approx 23,300$ times smaller for Q22. On top of that, our translation of Q22 implements the phase $\mathbb{2}$ optimization described in Section 9.6.1 and adds $\mathbb{p}_{const}$ only once to each group beforehand.

11.3.3 Comparison of Performance

Both approaches use PostgreSQL as a backend and derive their provenance through query execution by the DBMS. The performance comparison is based on the wall clock times $T^{\square}$ for query execution. The wall clock times are then turned into a *slowdown* factor using the following formulae.

Detached Provenance Analysis For the detached approach and TPC-H query i, the slowdown is

$$\frac{T_i^{\mathbb{1}} + T_i^{\mathbb{2}}}{T_i^{user}}$$

where T_i^{user} is the runtime for the unmodified TPC-H query. The two runtimes $T_i^{\mathbb{1}}$ and $T_i^{\mathbb{2}}$ correspond to the runtimes of the two phases of the detached approach. The log sizes (and runtimes) of phase $\mathbb{1}$ are the same as in Chapter 9. Be aware that *this* phase $\mathbb{2}$ derives data provenance in row granularity, resulting in an improved performance.

PERM For *PERM* and TPC-H query i, the slowdown is

$$\frac{T_i^{perm}}{T_i^{user.p}}$$

where $T_i^{user.p}$ is the runtime for the unmodified TPC-H query and T_i^{perm} is the runtime for the rewritten query.

Results The slowdowns of both approaches are plotted in Figure 11.8. The queries are grouped according to their nesting complexity. On the left–hand side (basic queries), most queries consist of a single SFW–expression. On the right–hand side, nesting–heavy queries are grouped together. The geometric means for each approach and group of queries are listed in Figure 11.9. The detached approach massively outperforms *PERM* when it comes to nested queries. These results are consistent with the working principles of the two approaches. *PERM* uses normalization to carry along provenance annotations. The more nested a query is, the more normalization overhead is induced. The detached approach however is based on two phases connected through logging. The associated write/read UDFs can be called directly from within any subquery without routing data through the surrounding queries. The experiments confirmed this intuition from the quantitative perspective. For the basic queries (on the left–hand side of Figure 11.8),

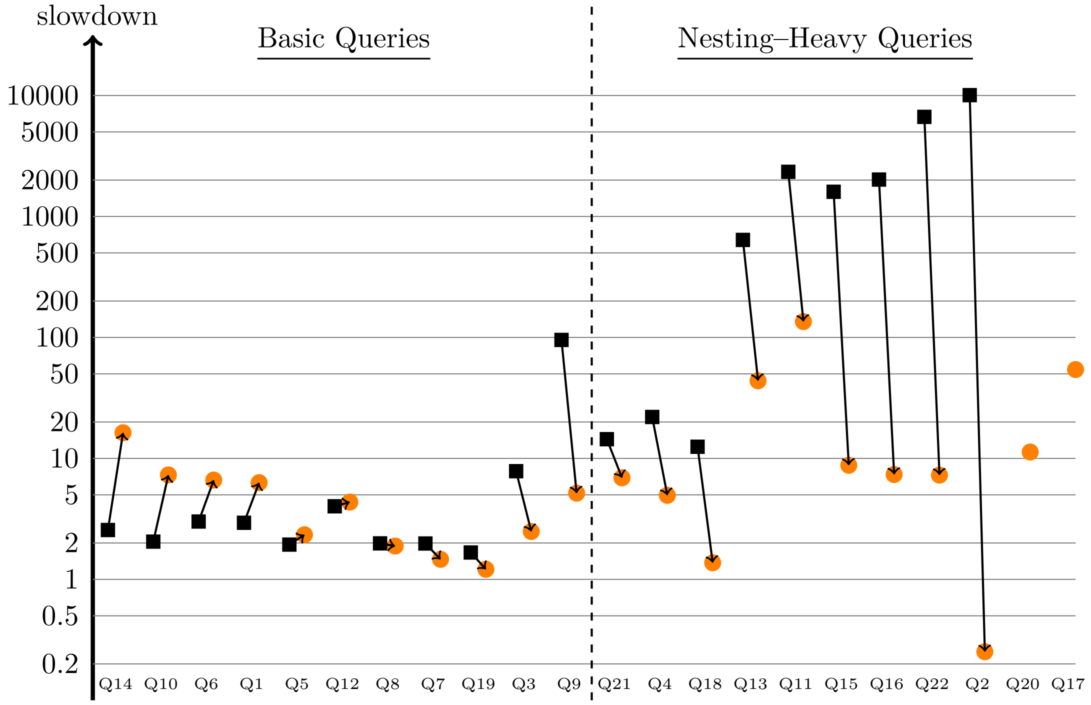


Figure 11.8: **Performance comparison: the detached approach and *PERM*.**

	Detached Approach	*PERM* Approach
Basic Queries	3.8	3.7
Nesting–Heavy Queries	6.8	477

Figure 11.9: **Geometric means of slowdowns for the two approaches and TPC-H.** Q20 **and** Q17 **are not included.**

PERM shows a slightly better performance. We presume that the logging overhead does not pay off for basic queries.

11.3.4 Basic Queries

The 11 basic queries are listed one the left–hand side of Figure 11.8. Most of them consist of a single SFW expression. Q7, Q8 and Q9 have another SFW expression nested in their respective `FROM`–clause. The slowdown ratios between the two approaches reside in between of Q14 and Q9.

- Q14 has a slowdown of 2.6 in *PERM* and 16 in the detached approach.
- Q9 has a slowdown of 95 in *PERM* and 5.2 in the detached approach.

The slowdown of 95 for Q9 in *PERM* is remarkably high. This result is consistent with the observation in [GA09a][Sec. V.A].

Q8, Q12 and Q14 contain `CASE` expressions. These expressions utilize logging in order to make provenance analysis of `CASE` in cell granularity possible. For row granularity however, this logging is superfluous and induces (useless) overhead. In fact, each query of the group of basic queries spends more time in phase 𝟙 than in phase 𝟚. If we would strip the logging of `CASE` expressions, the aforementioned queries might improve.

Looking closer at Q10, the detached approach has a bigger slowdown than *PERM* ($7.3 > 2.1$). In absolute numbers, log writing alone costs the detached approach $5.3s$ while *PERM* carries out the entire provenance analysis in $3.8s$. A comparison of log sizes, the time overhead per log row or an inspection of the query plans provided no additional insight. To the best of our knowledge, the evaluation of Q10 was reasonably fast in the detached approach. We assume that the *PERM* approach is better suited for this workload. *PERM* does not write logs and it is reasonable that this advantage shows in unnested queries like Q10.

11.3.5 Nesting–Heavy Queries

The nesting–heavy queries according to Figure 11.8 make (more) use of nested subqueries. Q2, Q4, Q17, Q20, Q21 and Q22 have correlated subqueries. Q13 has two nested SFW expressions, plus a left outer join. Q11, Q15, Q16 and Q18 have nested SFW–expressions in a `WHERE`– or `HAVING`–clause. Additionally, Q15 has a view definition which is referenced twice. We found the slowdown ratios between the two approaches reside in between of Q21 and Q2.

- Q21 had a slowdown of 14 in *PERM* and 7.0 in the detached approach.
- Q2 had a slowdown of $10,070$ in *PERM* and 0.25 in the detached approach.

All queries performed better in the detached approach and always terminated. The provenance analysis of Q20 and Q17 exceeded 15 hours using *PERM*. They are not considered in the comparison.

11.3.5.1 Discussion of Q2

We determined the single biggest difference in the slowdowns for Q2, i.e. $0.25 < 10,070$.

Speedup in the Detached Approach Q2 in the detached approach had a substantially better query runtime than the original query — the slowdown is 0.25 which corresponds

```
AND ps_supplycost = (
        SELECT
            MIN(ps_supplycost)
        FROM
            partsupp,
            supplier,
            nation,
            region
        WHERE
            p_partkey = ps_partkey
            and ...
    )
```

Figure 11.10: **Excerpt of Q2: `p_partkey` is correlated.**

to a speedup factor of 4.0. This observation is consistent with the performance of Q2 in cell–level granularity where Q2 experienced a speedup in both phases.

Slowdown in *PERM* The comparably high runtime in *PERM* is not due to provenance representation size. In numbers, the representation sizes are 605 in the detached approach and 6, 845 in *PERM* — a ratio of ≈ 11. Accordingly, Q2 is listed on the very left side of Figure 11.6. In fact, *PERM* yielded one of the smallest provenance results for Q2, both in absolute numbers and relative to the detached approach.

The main issue can be narrowed down to the correlated subquery of Q2, listed in Figure 11.10. According to the query plan, *PERM* employed the JA rewrite strategy which is a method for query decorrelation in context of aggregates.

> The rewrite rules of the JA strategy apply de-correlation and un-nesting to transform correlated sublink expressions into joins [...]. [Gla10, Sec. 5.2.6.2]

One of those joins is very expensive. The join predicate is not three–valued (i.e., `a=b` with `NULL=NULL` yields null) but two–valued (i.e., `NOT (a IS DISTINCT FROM b)` where `NULL` comparison yields true). According to a short experiment, the three–valued predicate gets planned as a hash join but the two–valued predicate gets planned as a nested loop join. The difference in query runtimes is ≈ 5, 000.

We conclude that the detached approach can deal with correlated subqueries in a more natural way. Basically, the rewritten queries for phases 𝟙 and 𝟚 are still correlated and the DBMS deals with the problem. However, the rewrite rules must be aware of correlated tuple variables and include them in the logging procedure (discussed in Section 8.3.2).

12 Summary

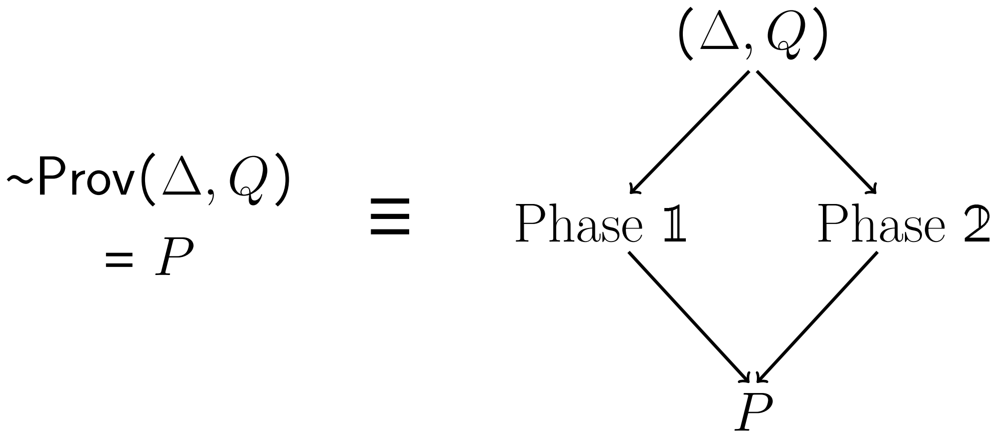


Figure 12.1: **Provenance analysis (left) turned into a *detached provenance analysis* (right).**

In this work, we have studied the problem of provenance analysis for SQL queries. First, we provided definitions for

- a relational model with embedded provenance annotations and
- the data provenance for a feature–rich SQL dialect. For example, the dialect covers correlated subqueries and window functions.

This thorough problem statement features arbitrarily deep nesting of data structures (namely, rows and arrays) and data provenance in sub–cell granularity. The provenance semantics is based on adapted notions of Where– and Why–provenance (originally defined in [BKT01]) with the computation of Why–provenance being optional. In symbols, we stated the problem $\sim\mathsf{Prov}(\Delta, Q) = P$ with Δ being a (read–only) database and Q a SQL query[1]. P is the according output table, consisting of output values annotated with data provenance.

The main part of this work put its focus on the provenance computation. In theory, an implementation of $\sim\mathsf{Prov}(\cdot, \cdot)$ from scratch is possible. However, any such prototype

[1] For presentation reasons, we approximate $\mathsf{Prov}(\cdot, \cdot, \cdot)$ with $\sim\mathsf{Prov}(\cdot, \cdot)$.

would have to face the issue of implementing a table storage, a query processor and other components. These components already exist in form of DBMS implementations. On top of that, a modern DBMS is a highly optimized software product, constituting the result of decades of research. In this work, we have contributed an approach called *detached provenance analysis* which leverages existing DBMS implementations for provenance analysis.

Figure 12.1 depicts how the provenance definition and the detached provenance analysis correspond to each other. The basic idea is to evaluate the two aspects of P (i.e., values and provenance annotations) in two separate computation steps, termed phase $\mathbb{1}$ and phase $\mathbb{2}$. We have provided a ruleset which carries out the translation $Q \rightsquigarrow (Q^{\mathbb{1}}, Q^{\mathbb{2}})$. The two resulting queries drive the two phases.

In the experimental part of this work, we have determined the overhead of provenance analysis in our approach (based on the TPC-H benchmark and the PostgreSQL DBMS). In a second set of experiments, we adapted the provenance granularity to row granularity and compared our approach to *PERM* [GA09a]. *PERM* is an extended PostgreSQL server with support for provenance analysis. Unlike to our approach, the provenance support is built into the DBMS itself. We distinguished between two sets of queries: (i) basic (for example, aggregating queries) and (ii) nesting–heavy (for example, correlated subqueries). For set (i), *PERM* showed a slightly better performance and for set (ii), the detached approach outperformed *PERM*.

13 Future Work

13.1 WITH RECURSIVE

```
WITH RECURSIVE
rtable AS (

  $e_b^{table}$
    UNION ALL
  $e_r^{table}$



)
...
```

(a) **User query.**

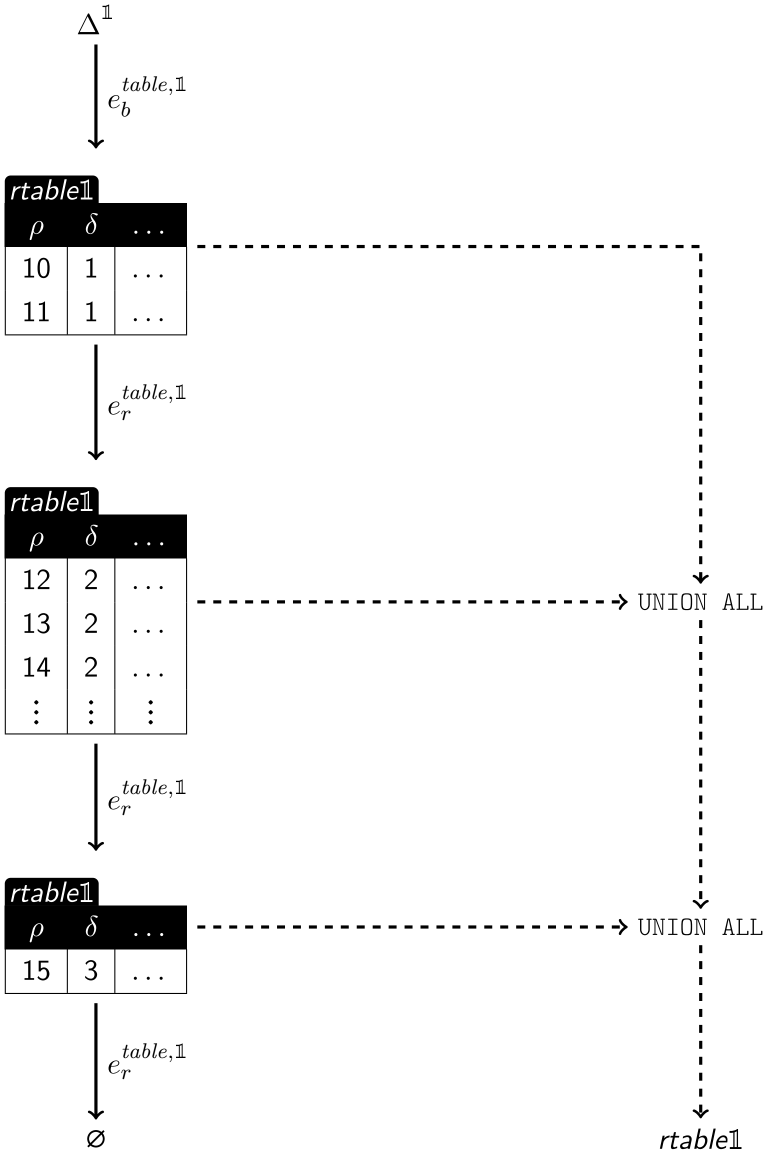
```
 1  WITH RECURSIVE
 2  rtable$\mathbb{1}$ AS (
 3
 4    $e_b^{table,\mathbb{1}}$
 5      UNION ALL
 6    SELECT ...
 7      FROM (...$\delta$+1...) AS v($\rho$),
 8           LATERAL $e_r^{table,\mathbb{1}}$ AS ...
 9  )
10  ...
```

(b) **Phase $\mathbb{1}$.**

Figure 13.1: **Provenance analysis for recursive queries (pseudocode).**

Let Figure 13.1(a) be a recursive query with recursive table definition *rtable*. The expression e_r^{table} gets evaluated repeatedly until it yields an empty table. The difficulty regarding to a detached provenance analysis is this repeated evaluation. Each evaluation is similar but different rows are involved. Any logging carried out within e_r^{table} needs to be aware of this repetition (as argued in Section 8.3.2).

Our idea is exemplified in Figure 13.2. After three evaluations of e_r^{table}, the empty table is found (denoted $\varnothing$). On the right–hand side of the figure, the *rtable*$\mathbb{1}$ get collected using UNION ALL and constitute the result table (also bound to *rtable*$\mathbb{1}$). Column ρ holds row identifiers (already known from Section 8.3.1). The new feature is column δ which is used as a counter for recursion depth. In Figure 13.1(b), we present the according query. On line 7, column $\boldsymbol{\delta}$ is being accessed and bound to `v(ρ)`. Using **LATERAL** on line 8, this makes `v` a tuple variable which can be referenced from anywhere within the recursive query $e_r^{table,\mathbb{1}}$. Then, `v` can be understood as a correlated tuple variable and included in the logging procedures carried out within $e_r^{table,\mathbb{1}}$. Thus, all logs can be qualified with the recursion depth `v(ρ)`. Log ambiguity (because of repeated evaluation of $e_r^{table,\mathbb{1}}$) is avoided.



rtable$\mathbb{1}$

ρ	δ	...
10	1	...
11	1	...

rtable$\mathbb{1}$

ρ	δ	...
12	2	...
13	2	...
14	2	...
⋮	⋮	⋮

rtable$\mathbb{1}$

ρ	δ	...
15	3	...

Figure 13.2: **Recursive evaluation at runtime.**

*A*𝟙

ρ	*mycol*
1	foo
2	bar

*B*𝟙

ρ	*mycol*
3	foo
4	baz

(a) **Input tables.**

*output*𝟙

ρ	*mycol*
2	bar

(b) `A EXCEPT B` **in phase 𝟙.**

```
SELECT
    a.ρ,
    a.mycol
FROM
    A𝟙 AS a,
    (
       SELECT a.mycol FROM A𝟙 AS a
       EXCEPT
       SELECT b.mycol FROM B𝟙 AS b
    ) AS diff
WHERE
    a.mycol = diff.mycol
```

(c) **Query in pseudocode.**

Figure 13.3: **Example:** `EXCEPT` **in phase 𝟙.**

The support of recursive `UNION DISTINCT` is more difficult. Additional columns would be considered in the process of duplicate elimination and therefore, would interfere with the query semantics.

13.2 Set Operations

The detached approach consistently augments all tables with an additional column of row identifiers. A straightforward duplicate elimination using the SQL `DISTINCT` keyword is not possible since the additional column would interfere with the query semantics. In the main part of our work, we employed `DISTINCT ON` instead (which allows to specify the columns being compared). Typical SQL dialects support additional set operations. In this discussion, we focus on `EXCEPT` (set difference \ in relational algebra).

Phase 𝟙 Figure 13.3(a) lists two example tables *A*𝟙 and *B*𝟙 (in phase 𝟙). Column *mycol* contains regular data (the payload). What we desire to evaluate is `A EXCEPT B` and yield the output table shown in Figure 13.3(b) (extended with *ρ* column). That means, the two ρ columns are to be ignored by the `EXCEPT` operator but nonetheless, an according ρ column hast to show up in the table. A possible query rewriting strategy is exemplified in Figure 13.3(c). The idea is to

- project the ρ columns away (lines 7,9),
- carry out the set operation (line 8) and
- re–attach the ρ column through a join on the payload column(s).

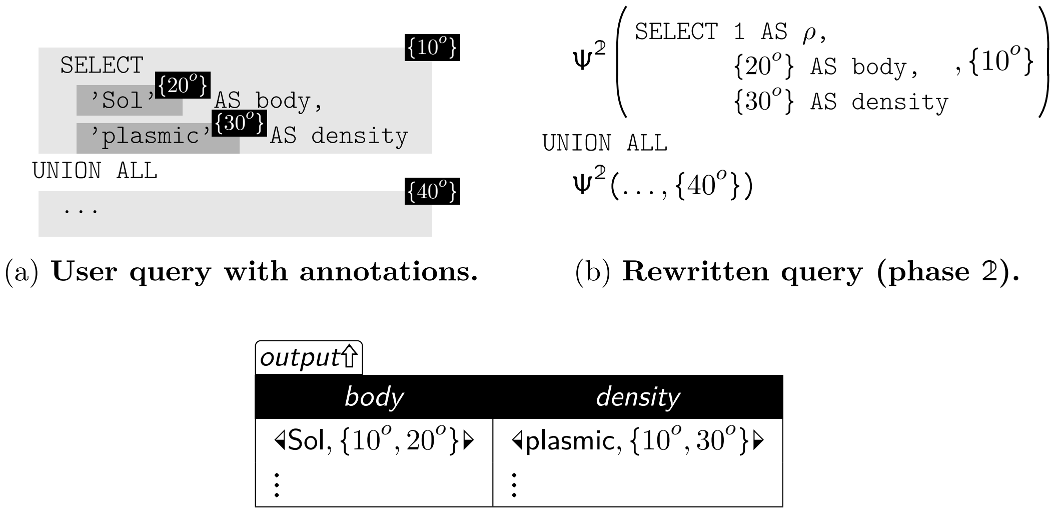
13.3 Slicing–Based How–Provenance

How–provenance introduced by T. Green, G. Karvounarakis and V. Tannen [GKT07] features a mapping of relational operators (e.g., join and union) to operations in a semiring (i.e., $\oplus$ and $\odot$). However, in the upcoming discussion, we ignore the semirings aspect. Instead, we sketch an alternative approach to How–provenance.

Figure 13.4(a) lists a short SQL query fragment which consists of `UNION ALL` and the constant expressions `'Sol'` and `'plasmic'`. The two subexpressions of the `UNION ALL` and both constants are annotated with (our adapted) How–provenance. The annotations are singleton sets with unique identifiers. Using the detached provenance analysis (we omit logging and phase $\mathbb{1}$), the rewrite could yield the phase $\mathbb{2}$–query provided in Figure 13.4(b). The rewrite of query constants is straight–forward (replacing them with their annotations) and the two subexpressions of `UNION ALL` make use of $\Psi^{2}(\cdot,\cdot)$ (see Definition 8.7), i.e. the according annotations get distributed. The How–provenance for this example is presented in Figure 13.4(c): the annotation $\{10^{o}, 20^{o}\}$ (for value: Sol) exhibits a contribution of the corresponding query constant and the specific branch of **`UNION ALL`** to that result value. The How–provenance exemplified here has cell granularity. In comparison to [GKT07], our adapted approach to How–provenance would be able to uniquely identify each single expression of the input query.

Related work by D. O'Grady, T. Müller and T. Grust [OMG18] has a very similar understanding of How–provenance and also employs the detached approach. However, the approach of [OMG18] utilizes query compilation instead of SQL rewrites.

Our adaption of How–provenance is related to *program slicing* [Wei84] by M. Weiser. The goal of program slicing is to identify the program fragment which has been involved in computing a certain output. Looking again at Figure 13.4(c), the How–provenance of output value Sol exactly identifies the expressions required for computing that value. Hence, query debugging could be a direct application. J. Cheney [Che07] already found a connection between program slicing and data provenance. However, the discussion in [Che07] focuses on the relationship of input data to output data (rather than query text



(a) **User query with annotations.** (b) **Rewritten query (phase 𝟚).**

(c) **Output table with How–provenance.**

Figure 13.4: **Example: How–provenance.**

to output data, as sketched above).

An important open question is if computation of (meaningful) How–provenance is possible in the context of query normalization (see Chapter 7).

13.4 Parallel Query Execution

Parallel query execution was not in the scope of this work. We expect that phase 𝟙 and the log–writing UDFs are an issue for parallelization. On the other hand, queries of phase 𝟚 are read–only and make heavy use of set union. The union operator is commutative and associative (as discussed in Section 9.6). The PostgreSQL DBMS we used in the experiments is however totally unaware of this optimization potential. In [MDG18b, Appendix D.], we present an additional set of experiments in which provenance annotations are not represented as arrays but as actual sets (using a handcrafted PostgreSQL plugin). That research could be continued and we may be able to teach PostgreSQL additional optimization rules for set union.

Bibliography

[ADT11] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for Aggregate Queries. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 153–164, 2011.

[AFG$^+$18] Bahareh Sadat Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. Gprom - A Swiss Army Knife for Your Provenance Needs. *IEEE Data Engineering Bulletin*, 41(1):51–62, 2018.

[BCTV05] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. An Annotation Management System for Relational Databases. *VLDB Journal*, 14(4):373–396, 2005.

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and Where: A Characterization of Data Provenance. In *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings.*, pages 316–330, 2001.

[BKT02] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. On Propagation of Deletions and Annotations Through Views. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 150–158, 2002.

[BMT07] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost In Translation: Formalizing Proposed Extensions to C$^\#$. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 479–498, 2007.

[CAA14] James Cheney, Amal Ahmed, and Umut A. Acar. Database Queries that Explain their Work. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 271–282, 2014.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A Structured English Query Language. In *Proceedings of 1974 ACM-SIGMOD Workshop* *on Data Description, Access and Control, Ann Arbor, Michigan, USA, May 1-3, 1974, 2 Volumes*, pages 249–264, 1974.

[CCT09] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[Che07] James Cheney. Program Slicing and Data Provenance. *IEEE Data Engineering Bulletin*, 30(4):22–28, 2007.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CTV05] Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. DBNotes: A Post-It System for Relational Databases based on Provenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 942–944, 2005.

[CW00] Yingwei Cui and Jennifer Widom. Lineage Tracing in a Data Warehousing System. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 683–684, 2000.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM TODS*, 25(2):179–227, 2000.

[DMG16] Benjamin Dietrich, Tobias Müller, and Torsten Grust. The Best Bang for Your Bu(ck)g. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016.*, pages 674–675, 2016.

[GA09a] Boris Glavic and Gustavo Alonso. PERM: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 174–185, 2009.

[GA09b] Boris Glavic and Gustavo Alonso. Provenance for Nested Subqueries. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, pages 982–993, 2009.

[GKT07] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance Semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART*

Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China, pages 31–40, 2007.

[Gla] Boris Glavic. *PERM.* `https://github.com/IITDBGroup/perm`.

[Gla10] Boris Glavic. *Perm: Efficient Provenance Support for Relational Databases.* PhD thesis, University of Zürich, 2010.

[GP10] Floris Geerts and Antonella Poggi. On Database Query Languages for k-Relations. *Journal of Applied Logic*, 8(2):173–185, 2010.

[JC94] C. Barry Jay and J. Robin B. Cockett. Shapely Types and Shape Polymorphism. In *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*, pages 302–316, 1994.

[Kah87] Gilles Kahn. Natural Semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, pages 22–39, 1987.

[KG12] Grigoris Karvounarakis and Todd J. Green. Semiring-Annotated Data: Queries and Provenance. *SIGMOD Record*, 41(3):5–14, 2012.

[KN10] Alfons Kemper and Thomas Neumann. One Size Fits all, Again! the Architecture of the Hybrid OLTP&OLAP Database Management System HyPer. In *Enabling Real-Time Business Intelligence - 4th International Workshop, BIRTE 2010, Held at the 36th International Conference on Very Large Databases, VLDB 2010, Singapore, September 13, 2010, Revised Selected Papers*, pages 7–23, 2010.

[LZW⁺97] Wilburt J. Labio, Yue Zhuge, Janet L. Wiener, Himanshu Gupta, Hector Garcia-Molina, and Jennifer Widom. The WHIPS Prototype for Data Warehouse Creation and Maintenance. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 557–559, 1997.

[MDG18a] Tobias Müller, Benjamin Dietrich, and Torsten Grust. You Say 'What', I Hear 'Where' and 'Why'? (Mis-)Interpreting SQL to Derive Fine-Grained Provenance. *PVLDB*, 11(11):1536–1549, 2018.

[MDG18b] Tobias Müller, Benjamin Dietrich, and Torsten Grust. You Say 'What', I Hear 'Where' and 'Why'? (Mis-)Interpreting SQL to Derive Fine-Grained Provenance. *arXiv e-prints*, page arXiv:1805.11517, May 2018.

[MG15] Tobias Müller and Torsten Grust. Provenance for SQL through Abstract Interpretation: Value-less, but Worthwhile. *PVLDB*, 8(12):1872–1875, 2015.

[Mül15] Tobias Müller. Where- und Why-Provenance für syntaktisch reiches SQL durch Kombination von Programmanalysetechniken. In *Proceedings of the 27th GI-Workshop Grundlagen von Datenbanken, Gommern, Germany, May 26-29, 2015.*, pages 84–89, 2015.

[Mül16] Tobias Müller. Have Your Cake and Eat it, Too: Data Provenance for Turing-Complete SQL Queries. In *Proceedings of the VLDB 2016 PhD Workshop co-located with the 42nd International Conference on Very Large Databases (VLDB 2016), New Delhi, India, September 9, 2016.*, 2016.

[Neu11] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[NK15] Thomas Neumann and Alfons Kemper. Unnesting Arbitrary Queries. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, pages 383–402, 2015.

[OMG18] Daniel O'Grady, Tobias Müller, and Torsten Grust. How "How" Explains What "What" Computes — How-Provenance for SQL and Query Compilers. In *10th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2018, London, UK, July 11-12, 2018.*, 2018.

[Pos] *The PostgreSQL Relational Database System.* postgresql.org.

[PW07] Simon Peyton Jones and Philip Wadler. Comprehensive Comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 61–72, 2007.

[PW18a] Fotis Psallidas and Eugene Wu. Smoke: Fine-grained Lineage at Interactive Speed. *PVLDB*, 11(6):719–732, 2018.

[PW18b] Fotis Psallidas and Eugene Wu. Smoke: Fine-grained Lineage at Interactive Speed. *arXiv e-prints*, page arXiv:1801.07237, January 2018.

[Pyt] *The Python Programming Language.* python.org.

[SJMR18] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. ProvSQL: Provenance and Probability Management in PostgreSQL. *PVLDB*, 11(12):2034–2037, 2018.

[SQL16] *Database Languages–SQL–Part 2: Foundation*, 12 2016. ISO/IEC 9075-2:2016.

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

[Wei84] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[WM90] Y. Richard Wang and Stuart E. Madnick. A Polygen Model for Heterogeneous Database Systems: The Source Tagging Perspective. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings.*, pages 519–538, 1990.

[WS97] Allison Woodruff and Michael Stonebraker. Supporting Fine-grained Data Lineage in a Database Visualization Environment. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, pages 91–102, 1997.

Acknowledgements

First of all, I want to thank my advisor Torsten Grust. He guided me throughout the entire process of being a doctoral candidate. His strong support started out with the bare essentials like funding and providing generous office space. I felt very welcome in Torsten's research group. Torsten carefully guided me through the research process but without pushing me towards results. Instead, he kept asking sensible questions which lead to valuable results.

Benjamin Dietrich and Daniel O'Grady were two other doctoral candidates of Torsten and the four of us published a number of papers together. I want to thank all of you for the enjoyable cooperation in the research and publication process. At time of writing, additional members of Torsten's group are Christian Duta and Denis Hirn. Former members I collaborated with are Alexander Ulrich, Dennis Butterstein, Jeroen Weijers, Noah Doersing and Tom Schreiber. Thank you for the pleasant collaboration and the fruitful discussions.

A substantial part of my funding came from the Wissenschaftscampus Tübingen (Bildung in Informationsumwelten). The interdisciplinary cooperation with scientists from the fields of economy and psychology has lead me to the topic of my dissertation. My thanks go to my direct collaborators Iassen Halatchliyski, Michael Kummer and Olga Slivkó and the organizers of the Wissenschaftscampus.

Finally, I want to thank my professors. Torsten Grust and Boris Glavic were the reviewers of my dissertation. The oral exam was conducted by Torsten, Boris, Wolfgang Küchlin and Klaus Ostermann. Thank you for giving me the opportunity to graduate.