Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät

Masterthesis Computer Science

# How to Optimize What Is Slow in Data Provenance and Why You Should Do It

*Daniel Täsch*
March 2, 2022

**Examiner**

Prof. Dr. Torsten Grust
Database Systems Research Group
University of Tübingen

**Co-Examiner**

Prof. Dr. Michael Menth
Chair for Communication Networks
University of Tübingen

**Supervisor**

Dr. Tobias Müller
University of Tübingen

**Täsch, Daniel:**
*How to Optimize What Is Slow in Data Provenance and Why You Should Do It*
Masterthesis Computer Science
Eberhard Karls Universität Tübingen
From 01.12.2021 to 01.06.2022

# Abstract

Data provenance is a useful tool in SQL query debugging and auditing.[ME22] Given an SQL query, it computes not the output of the query but the input cells involved in computing the real output (so called *provenance sets*).

The problem is, our approach of data provenance computes (in most cases huge) provenance sets for each cell of the result set. This results in a slowdown for most of the tried queries.[Mül20, Chapter 9.5.2]

In this work, multiple solutions to optimize what is slow in data provenance were considered. It is explained, how these solutions work and argued why these solutions are useful.

The first approach uses query rewriting to remove duplicates in provenance sets. For the optimization of why-provenance (our second optimization-approach), query rewriting is used, too, to move the computation to better suited places in the query. The last approach uses hash-tables, accelerating the needed logging of our approach.

# Contents


1. Introduction 1

2. Data Provenance 5
   - 2.1. What Is SQL? 5
   - 2.2. What Is Data Provenance? 7
   - 2.3. Why Is Data Provenance Useful? 8
   - 2.4. How Does Data Provenance Work? 9
     - 2.4.1. A New Set of Tables 9
     - 2.4.2. Query Normalization 9
     - 2.4.3. Phase 1 and 2 Queries 12
   - 2.5. Where to Find Optimization Potential? 15
     - 2.5.1. The Problem with Arrays 15
     - 2.5.2. Uncorrelated Subqueries in Why-Provenance 17
     - 2.5.3. Logging 18

3. Implementation 19
   - 3.1. Representing SQL in Haskell 19
     - 3.1.1. The `SQL`-Module 22
     - 3.1.2. The `SetLang`-Module for Phase 2 Queries 22
   - 3.2. Translation and Translation Rules 23
   - 3.3. Logging 34
   - 3.4. Optimization Implementation 34
     - 3.4.1. Set Union Optimization 34
     - 3.4.2. Why-Provenance Optimization 36
     - 3.4.3. Replacing Log Tables with Hash-Tables 37

4. Experiments 39
   - 4.1. TPC-H Benchmark 39
   - 4.2. Database Setup 39
   - 4.3. Experiment Setup 40
   - 4.4. Experiment Structure 41
   - 4.5. Optimization Results 42
     - 4.5.1. Results from Query Rewriting (Phase 2) 42
     - 4.5.2. Results from Logging with Hash-Tables (Phase 1) 44
     - 4.5.3. Cumulated Results from All Optimizations 45

5. Conclusion & Outlook 47
   - 5.1. Summary 47
   - 5.2. Future Work 47

Bibliography 53

List of Figures 56

Appendix 59
   - A. `SQL` Dialect 59
   - B. `SetLang` Dialect 61

# 1 INTRODUCTION

In times of exponential data growth (see e.g. [GH18, chapter 13]), there must be methods to explain where the information comes from that was used to compute a specific result. Having this possibility creates trust in the computed results and makes them easier to understand. These methods can hence be used for debugging or auditing purposes in application development.[ME22]

One of these methods for SQL queries is data provenance. The approach of the Research Group from the University of Tübingen is to take an SQL query and to rewrite it. They developed a two-phase approach for computing cell level data provenance.[OMG18] This two-phase approach lays the base of this work.

In figure 1.1, an overview of this approach is given. Keep this figure in mind, since we will come back to it throughout this work and modify it according to our knowledge gain.

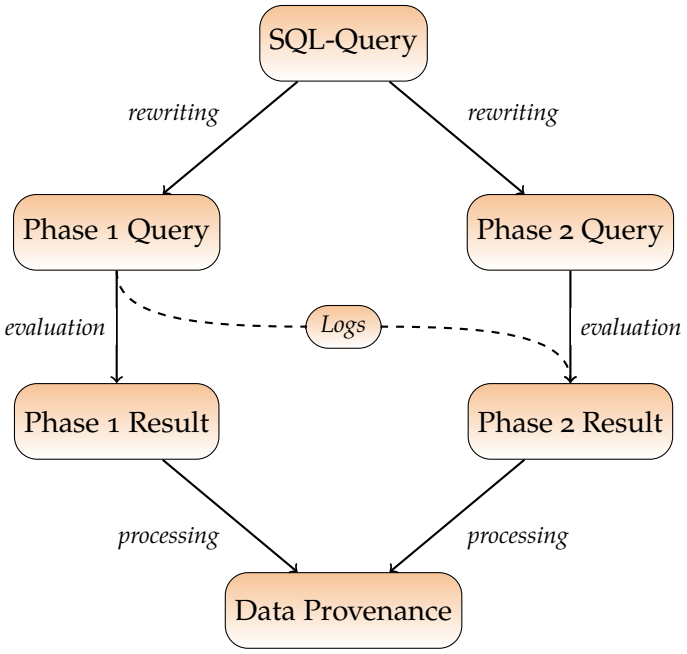



**Figure 1.1.:** Approach to data provenance of the Research Group

With this approach, where- and why-provenance are derived: The information where the data, that is part in the result, comes from and why this data took part in the computation of this result.[MDG18]

The problem with this two-phase approach is: it is slow.[Mül20, Chapter 9.5.2] Since fast feedback is essential for the use of data provenance as a query debugging tool, the goal of this work is to speed up provenance computation and get feedback faster.

Therefore, three optimization approaches were developed: one for phase 1 and two for phase 2. The two approaches for phase 2 are based on query rewriting (see the newly added *„**rewriting**“* arrow in figure 1.2). For this a Haskell module was developed that mixes SQL with set language to make this rewriting elegant and simple.

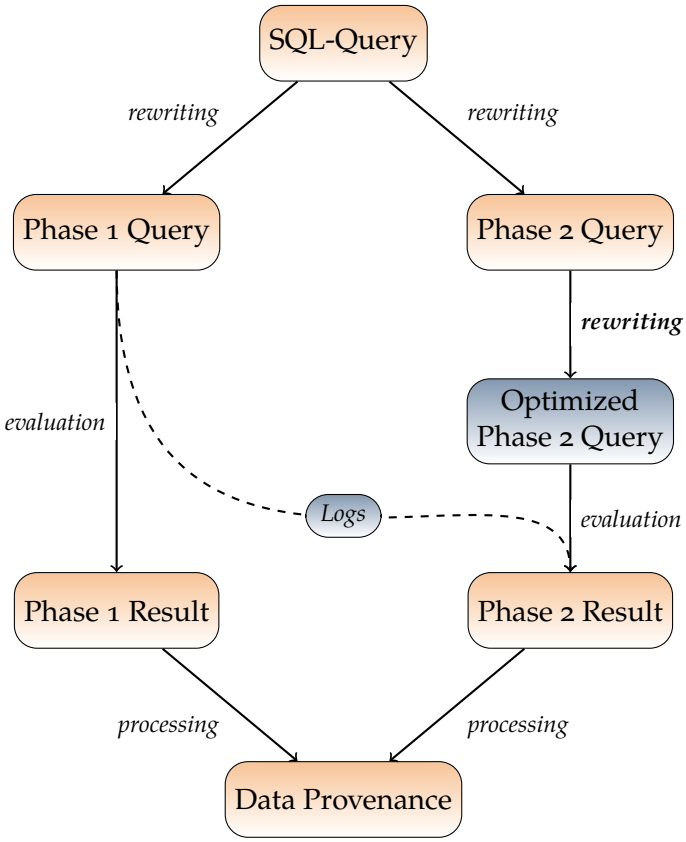



**Figure 1.2.:** Optimization locations in the approach from figure 1.1

The approach for phase 1 takes place in the data base with the PostgreSQL plugin `pg_hashtable` that replaces the Research Group's implementation for logging with database tables (see figure 1.2).[Bur21, chapter 4.1.2]

These three approaches are completely flexible in their usage and do not rely on up-front knowledge like Fotis Psallidas and Eugene Wu in their work [PW18]. Their approach prunes data that will not be used again and materializes data that will be used again to recompute result faster in the future.

The ideas of our approaches of query rewriting were outlined in the works from Tobias Müller, Benjamin Dietrich and Torsten Grust [MDG18] and specified in the dissertation of Tobias Müller [Mül20, chapter 9.6]. The goal of this work is to automatically optimize these approaches.

In this work, we will start with the idea behind data provenance in chapter 2. Therefore, a short summary of the needed SQL constructs is provided which an intuitive example for data provenance is based on. This example then is expanded to formal data provenance. Based on this, it will be discussed why data provenance is slow.

In chapter 3, the implementation of the SQL dialect and the translation to the two phases of data provenance (using *abstract syntax trees* in Haskell) is shown. The theoretical background for this was formalized by the Research Group and implemented by the author.

Starting with a simple SQL query, the corresponding Haskell data types are developed exemplary and the needed constructors are explained. After this, all used translation rules are formaly defined and explained. The last part of the chapter focuses on the implementation of the optimizations.

The next chapter 4 is about the executed experiments which are based on the implementation and the rules from chapter 3. All carried out experiments are explained and the results of these experiments are condensed in multiple figures to make the discussion of the results easier.

Since it is the base for this work, let's start by taking a look at data provenance.

# 2

# DATA PROVENANCE

Data provenance plays with the questions *what?*, *where?* and *why?*[1] This chapter starts with a short summary of *Structured Query Language* (SQL) which is the foundation of many relational database management systems. Next is an outline what data provenance is and how it works on SQL queries. After this explanation, we will move on to what data provenance can be used for in application development.

But unfortunately, data provenance comes with costs: The paper [MDG18] reported slowdowns up to a factor of 1000. In the end of this chapter, multiple reasons for this slowdown are stated and solution outlines for each of them are developed.

## 2.1. What Is SQL?

Interacting manually with large amounts of data structured in rows and columns of multiple tables gets very tedious very quickly. Take figure 2.1, a table of the planets in our solar system (along with their densities and their masses in earth masses), and the following simple question as an example:

"What is the total mass of all non-gaseous planets?"

planets

| name | density | mass |
|---|---|---|
| Mercury | rocky | 0.1 |
| Venus | rocky | 0.8 |
| Earth | rocky | 1.0 |
| Mars | rocky | 0.1 |
| Jupiter | gaseous | 318.0 |
| Saturn | gaseous | 95.0 |
| Uranus | icy | 15.0 |
| Neptune | icy | 17.0 |

**Figure 2.1.:** Table `planets`

[1] And *how?*, but how-provenance is not considered in this work.[OMG18]

You have to look in the density column to filter out Jupiter and Saturn from further processing. Now, you can take the values of the mass column of the remaining rows to compute the total mass of the remaining planets Mercury, Venus, Earth, Mars, Uranus, and Neptune:

$$\overbrace{0.1}^{\text{Venus}} + \overbrace{0.8}^{\text{Mercury}} + \overbrace{1.0}^{\text{Earth}} + \overbrace{0.1}^{\text{Mars}} + \overbrace{15.0}^{\text{Uranus}} + \overbrace{17.0}^{\text{Neptun}} = \overbrace{34.0}^{\text{mass\_sum}}$$

To organize this kind of structured data, you can use *relational database management systems* (RDBMS) like PostgreSQL.[psq] *Structured Query Language* (SQL) is used to interact with these RDBMSs.[CB74] SQL is divided in

- *data definition language* (DDL),
- *data manipulation language* (DML),
- *data control language* (DCL),
- *transaction control language* (TCL),
- ***data query language* (DQL)**.[sql]

With **DDL** you can create, drop and alter tables (e.g. add or remove columns to/from tables), you can define and declare new types for those columns of your tables, set indices, and add comments.

You can insert, update and delete data, and interact with locks with **DML**, and grant or revoke access privileges with **DCL** commands.

**TCL** is used to manage transactions: this allows to savely manipulate data with DML without the risk of leaving inconsistent data when errors occur. When all steps in the transaction block are done, you commit your changes and persist them, or you can roll them back and return to the point before you began the data manipulation.

```
SELECT    e_1 AS c_1, ..., e_n AS c_n   -- columns and computations included in result
FROM      t_1 AS v_1, ..., t_m AS v_m   -- tables and subqueries to search for data
WHERE     p                             -- filter criterion
GROUP BY  g_1, ..., g_k                 -- columns to group by
HAVING    q                             -- group filter criterion
ORDER BY  o_1, ..., o_j                 -- columns to order by
OFFSET    l_1                           -- ignore the first l_1 elements
LIMIT     l_2                           -- only include l_2 elements in result
```

**Figure 2.2.:** General SQL query

```
SELECT sum(p.mass) AS mass_sum
FROM   planets p
WHERE  p.density <> 'gaseous'
```

**(a)** Query Q1

| Output |
| --- |
| mass_sum |
| 34.0 |

**(b)** Result of Q1

**Figure 2.3.:** "What is the total mass of all non-gaseous planets?"

The for this work important subset of expressions (*queries*) are those from **DQL** which retrieve data from existing tables and compute new result tables from this data. A general SQL query can be seen in figure 2.2.

To answer the question from above, "What is the total mass of all non-gaseous planets?", we can use query Q1 from figure 2.3a. It consists of the **SELECT**, **FROM** and **WHERE** statements from figure 2.2, computing the sum of all mass es from all rows in the table planets that have a density other than `'gaseous'`.

So, an SQL query takes data from a table, filters and manipulates this data, and creates a new output table as seen in figure 2.3b. In summary: "SQL [...] specifies what the desired output of a query is."[MDG18]

## 2.2. What Is Data Provenance?

While SQL determines *what* the result looks like (see 2.3b), data provenance describes *where* the input data that gave this result (34.0) came from, and *why* it had an impact on the result:[MDG18]

| planets | | |
| --- | --- | --- |
| name | density | mass |
| Mercury | rocky | 0.1 |
| Venus | rocky | 0.8 |
| Earth | rocky | 1.0 |
| Mars | rocky | 0.1 |
| Jupiter | gaseous | 318.0 |
| Saturn | gaseous | 95.0 |
| Uranus | icy | 15.0 |
| Neptune | icy | 17.0 |

**Figure 2.4.:** Intuitive where- and why-provenance for query Q1 (figure 2.3a)

In this case, 34.0 clearly is the sum of the marked values in the mass column. The marked values in the density column are those values matching the filter criterion.

In our approach, data provenance is computed on cell level: For each cell $o$ in the output table $t$ (query Q1 has exactly one cell as output: see figure 2.3b) results a separate *dependency set* of cells that were involved in the computation of this output cell $o$.

A *dependency set* is the (possibly empty) set $\{i_1, \ldots, i_n\}$ of input table cells that were copied, transformed, or inspected to compute the value of a given output cell $o$.[MDG18] $o$ and $i_1, \ldots, i_n$ are identifiers (later referred to as tuid s) that represent the cells themselves not the values from these cells.

## 2.3. Why Is Data Provenance Useful?

With a simple query like Q1 from figure 2.3a, it is easy to see where the data that created the result 34.0 came from. It is pretty easy to explain the query to other people, too. But what about more complex queries? In those complex queries, it is easy for developers to make tiny bugs that deliver erroneous results without triggering static or dynamic errors.[ME22]

Data provenance makes it possible to verify the origins of every output value. With this data, it is easier to understand and explain complex queries. It helps building trust in the result of those queries.[MDG18]

planets[1]

| tuid | name | density | mass |
|---|---|---|---|
| 1 | Mercury | rocky | 0.1 |
| 2 | Venus | rocky | 0.8 |
| 3 | Earth | rocky | 1.0 |
| 4 | Mars | rocky | 0.1 |
| 5 | Jupiter | gaseous | 318.0 |
| 6 | Saturn | gaseous | 95.0 |
| 7 | Uranus | icy | 15.0 |
| 8 | Neptune | icy | 17.0 |

**(a)** Phase 1 table

planets[2]

| tuid | name | density | mass |
|---|---|---|---|
| 1 | {101} | {102} | {103} |
| 2 | {201} | {202} | {203} |
| 3 | {301} | {302} | {303} |
| 4 | {401} | {402} | {403} |
| 5 | {501} | {502} | {503} |
| 6 | {601} | {602} | {603} |
| 7 | {701} | {702} | {703} |
| 8 | {801} | {802} | {803} |

**(b)** Phase 2 table with intuitive where- and why-provenance from table 2.4

**Figure 2.5.:** Automatically generated phase 1 and phase 2 tables

It is also possible to debug queries with data provenance: Looking at the dependency set for an unlikely result can help finding errors in the query quickly. E.g. when in Q1 the predicate was to wrongfully be `WHERE p.density = 'gaseous'`, the result would be 413.0. The provenance set containing only cells from Jupiter and Saturn would very quickly lead to this wrong predicate as fault. Especially, when you also consider how-provenance.[ME22]

In summary: Application development benefits from data provenance, because queries a programmer developed can be better explained to colleagues and managers, so fewer errors happen. In addition, dealing with errors when they occur is easier with the tools of data provenance, so they can be fixed faster and the developer can come back to developing new features.

## 2.4. How Does Data Provenance Work?

Our approach to data provenance is a two step approach. The phase 1 query evaluates the given query and writes logs, listing all cells involved in computing the result. The phase 2 query reads from those logs and returns the provenance set of unique cell identifiers (tuid s) for each cell in the result.

### 2.4.1. A New Set of Tables

To work with these tuid s, two new tables are generated in advance: One table for phase 1 ($\text{planets}^1$) which adds a new column with a unique tuid to the former table planets and one for phase 2 ($\text{planets}^2$), filled only with arrays of tuid s instead of real values. These tables share the same tuid s for every row (see figure 2.5 for the phase 1 and 2 tables for the table planets from figure 2.1).

Now it becomes obvious, why we need two phases for this approach: Table $\text{planets}^2$, on which the provenance sets are computed, does not contain any values that can be summed or used as filter criteria.

To make the phase 2 table work with arrays more smoothly, the columns are filled with single valued arrays of tuid s, instead of integers.

### 2.4.2. Query Normalization

Our approach of data provenance works only with normalized queries: Before we can translate an SQL query to its corresponding phase 1 and 2 queries, it has to be normalized like seen in figure 2.6. That serves the purpose of clause

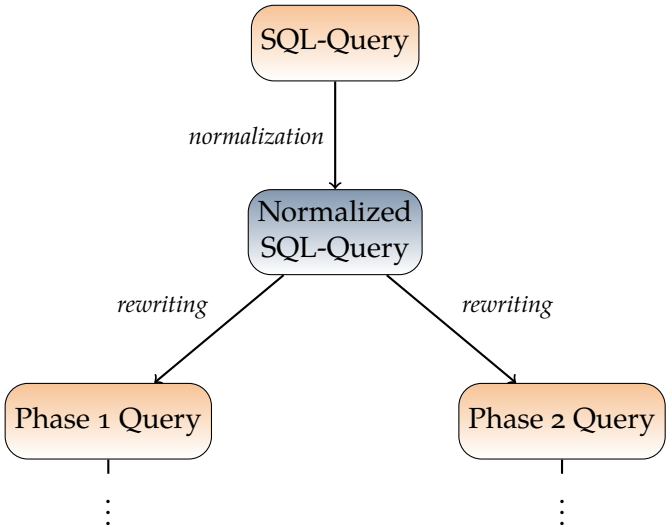



**Figure 2.6.:** Add normalization to the approach

```
SELECT    ...
FROM      (SELECT   ... AGG(...) ...
          FROM      (SELECT ...
                     FROM    q_1, ..., q_n
                     WHERE   p_1) AS t_1
          GROUP BY  g_1, ..., g_m
          HAVING    p_2) AS t_2
ORDER BY  o_1, ..., o_k
LIMIT     l
```

**Figure 2.7.:** A generic normalized query

isolation. The result is an "onion-style" uncorrelated nesting of queries in the `FROM` clause (see figure 2.7). "Normalization preserves query semantics as well as data provanence."[MDG18]

Automatic normalization was not in the scope of this work, so every query that was used in this work was normalized manually before the automatic translation to its corresponding phase 1 and 2 queries.

Only the innermost subquery (■) deals with multiple tables being joined. Every other query can assume that it has only one `FROM` clause. The second layer (■) deals with grouping, aggregation and filtering these groups. Only this layer has to deal with aggregates.

In the third layer (□), the result is ordered and limited to $l$ rows.[2] Now we have three possible query forms:

```
(1) SELECT ... FROM ..., ... WHERE ...;
(2) SELECT ... FROM ... GROUP BY ...;
(3) SELECT ... FROM ... ORDER BY ... LIMIT ...;
```

**Figure 2.8.:** Possible query forms after normalization

For the normalization of Q1, only two layers, the join (■) and the group (■) layers are needed (see figure 2.9). The results are filtered in the inner subquery for non-gaseous planets and the mass is summed in the outer query (with a generic `GROUP BY True`) to sum all results from the inner query. With `HAVING True`, no rows from the result set are filtered out.

For the sake of simplicity, from now on we will assume that all input queries are already normalized.

```
SELECT   sum(t.mass) AS mass_sum
FROM     (SELECT p.mass AS mass
          FROM   planets p
          WHERE  p.density <> 'gaseous') t
GROUP BY True
HAVING True
```

**Figure 2.9.:** Normalized version of Q1

[2] `OFFSET` could be implemented here, too, but was not needed for the measured queries.

### 2.4.3. Phase 1 and 2 Queries

The normalized query from figure 2.9 now can be automatically rewritten to its corresponding phase 1 and 2 queries (see figure 2.11). You will see in chapter 3.2 how this rewriting works in detail. In this chapter, the focus is on how the rewritten queries interact with the phase 1 and phase 2 tables, and with the logs to compute data provenance. This will deepen the understanding of the two phases, before diving into detail in chapter 3.

In the first step (see figure 2.10 on the left), the phase 1 query can be executed on `planets_1`, the SQL version of planets[1], instead of the original table `planets`. In addition to computing the result of the given query, it writes logs of tuid s involved in the computation with `writelog` (see the inner **SELECT** in figure 2.11a) and `writeagg` (see the outer **SELECT** in figure 2.11a). Query Q1[1] produces the output and logs seen in figure 2.12.

You can see that the tuid s of the planets Jupiter (5) and Saturn (6) are missing in log1. Only tuid s of planets fulfilling the predicate `density <> 'gaseous'` are listed in table 2.12a. These planets get a new unique tuid tuidout for further processing of log1.

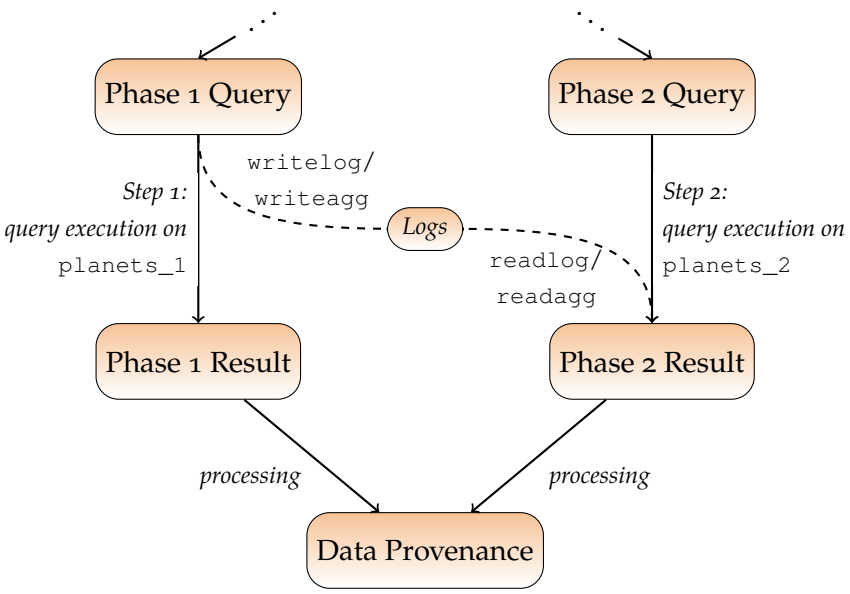



**Figure 2.10.:** Query execution in phases 1 and 2

```
SELECT   writeagg(0, array_agg(t.tuid)) AS tuid,
         SUM(t.mass) AS mass_sum
FROM     (SELECT writelog(1, p.tuid) AS tuid,
                 p.mass AS mass
          FROM   planets_1 AS p
          WHERE  p.density <> 'gaseous') AS t
GROUP BY True
    HAVING   True
```

**(a)** Query $Q1^{\mathbb{1}}$

```
SELECT   l0.tuid AS tuid,
         array_concat(t.mass) || array_concat(wh.y) AS mass_sum
FROM     (SELECT v1.tuid, p.mass || wh.y AS mass
          FROM   planets_2 AS p,
                 LATERAL readlog(1, p.tuid) AS v1,
                 LATERAL toY(p.density) AS wh(y)
          WHERE  True) AS t,
         LATERAL readAgg(0, t.tuid) AS l0,
         LATERAL toY(Array[]) AS wh(y)
GROUP BY l0.tuid
HAVING   True
```

**(b)** Query $Q1^{\mathbb{2}}$[a][b]

[a] duplicate and empty expressions are omitted for simplicity

[b] `array_concat(...)` is a shorthand for `concat(array_agg(array_to_json(...)))`

**Figure 2.11.:** Query Q1 and its corresponding phase 1 and phase 2 queries

Output

| tuid | mass_sum |
|---|---|
| 86814 | 34.0 |

**(a)** Result of $Q1^{\mathbb{1}}$

log1

| location | tuid1 | tuidout |
|---|---|---|
| 1 | 4 | 86811 |
| 1 | 7 | 86812 |
| 1 | 3 | 86810 |
| 1 | 2 | 86809 |
| 1 | 1 | 86808 |
| 1 | 8 | 86813 |

**(b)** Log generated by `writelog(1,...)`

logagg1

| location | tuid1 | tuidout |
|---|---|---|
| 0 | 86812 | 86814 |
| 0 | 86811 | 86814 |
| 0 | 86809 | 86814 |
| 0 | 86813 | 86814 |
| 0 | 86808 | 86814 |
| 0 | 86810 | 86814 |

**(c)** Log generated by `writeagg(0,...)`

**Figure 2.12.:** Logs generated by $Q1^{\mathbb{1}}$

The function `writeagg` takes those new tuid s, and logs which tuid s are involved in computing the **sum**. Since the result only has one row, there is only one new tuidout. All tuid s from log1 get collected under this new tuidout in figure 2.12c.

The phase 2 query $Q1^{2}$ from figure 2.11b now computes the provenance sets for each non-tuid cell in the output and links them with their unique tuid using the logs computed in phase 1. Since there are no values to compare or to sum in the phase 2 table, but only arrays of tuid s, the phase 2 query $Q1^{2}$ needs those logs. In this case, there is only one cell which needs a provenance set with tuid 86814:

Output

| tuid | mass_sum |
|---|---|
| 86814 | {−702, 803, −302, −202, 703, −102, −802, −402, 403, 203, 303, 103} |

**Figure 2.13.:** Output of query $Q1^{2}$ using the logs from figure 2.12 and the tuids from figure 2.5b

Positive values denote where-provenance, negative values why-provenance. Recalling figure 2.5b, and comparing it with figure 2.4, you can see that the intuitive understanding of data provenance and the result as seen here match.

## 2.5. Where to Find Optimization Potential?

Now that we know how data provenance works, let's look at the possibilities for optimization. In a nutshell: there are three main reasons why provenance queries are slow:

1. arrays,
2. uncorrelated subqueries and
3. logging

The first two concern phase 2 of our approach of data provenance. Query rewriting is used to deal with the arising issues. Since phase 1 queries mostly duplicate the logic of the underlying queries, only adding logging to these queries, there is no generic possibility to get advantages through query rewriting. So, to optimize phase 1, writing logs has to get more efficient.

### 2.5.1. The Problem with Arrays

Query rewriting in translation to the phase 2 query e.g. takes all column-references in `WHERE` expressions and collects them in an array, providing why-provenance. When there are duplicates in the expression, there are duplicates in this array, too. Take an alternative form of Q1 as an example, replacing `WHERE p.density <> 'gaseous'`:

```
SELECT sum(p.mass) as mass_sum
FROM   planets p
WHERE  p.density = 'rocky' OR p.density = 'icy'
```

**Figure 2.14.:** Alternative of query Q1 from figure 2.3a

This query produces similar phase 1 and phase 2 queries as shown in figure 2.11b, but with duplication in why-provenance:

```
LATERAL toY(p.density || p.density) AS wh(y)
```

The attentive reader already spotted a detail of the phase 2 query in figure 2.11b: duplicate and empty expressions were omitted for better readability. This is possible, because dependency sets in phase 2 are exactly that: *sets*[3].

[3] in the mathematical sense

In PostgreSQL arrays represent those sets, but since the internal optimizer of the database management system does not know about this missmatch (arrays vs. sets), it is unaware of the optimization potential:[Mül20, chapter 9.6]

$$\{a\} \cup \{a\} = \{a\} \qquad \bigcup \{a\} = \{a\} \qquad \{a\} \cup \varepsilon = \{a\} \qquad \varepsilon \cup \{a\} = \{a\}$$

In phase 2, we work on potentially large sets and unions of those sets that are automatically generated. In this automatic generation of queries, many duplicate entries and empty expressions occurr like seen in figure 2.15. These entries drain performance in query execution and produce duplicate entries in the output table which then have to be processed by the more outer queries of our nested approach. The solution here is query rewriting (see figure 2.16), to remove these duplicate entries from unions.

It is not enough to remove duplicates in the arrays of the query itself, but duplicates have to be removed in the result arrays, too. For this last step of query evaluation, the `array_concat` function is added to every column in the outermost **`SELECT`** expression (see figure 2.11b).

```
[...]
  SELECT  [...]
  FROM    [...],
  LATERAL toY(p_partkey || ps_partkey
    || p_brand || Array[]::integer[]
    || p_type  || Array[]::integer[]
    || p_size  || Array[]::integer[]
    || p_size  || Array[]::integer[]
    || p_size  || Array[]::integer[]
    || p_size  || Array[]::integer[]
    || p_size  || Array[]::integer[]
    || p_size  || Array[]::integer[]
    || p_size  || Array[]::integer[]
    || p_size  || Array[]::integer[]) AS wh(y)
  WHERE   [...]
[...]
```

**Figure 2.15.:** excerpt from automatically generated phase 2 query for TPC-H's Q16 (see chapter 4.1)

```
[...]
  SELECT [...]
  FROM   [...],
  LATERAL toY(p_partkey
    || ps_partkey
    || p_brand
    || p_type
    || p_size) AS wh(y)
  WHERE  [...]
[...]
```

**Figure 2.16.:** excerpt from optimized phase 2 query for TPC-H's Q16

### 2.5.2. Uncorrelated Subqueries in Why-Provenance

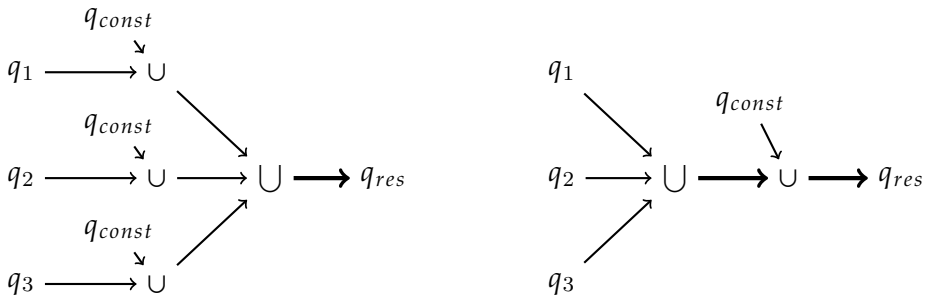
**WHERE** and **HAVING** expressions can include big and expensive *uncorrelated subqueries* (denoted as $q_{const}$ in figure 2.18). Uncorrelated subqueries are queries that can stand on their own and that do not need information from their context. Take the following query as an example, computing the names of all planets having a below-average mass:

```
SELECT p1.name
FROM   planets p1
WHERE  p1.mass < (SELECT avg(p2.mass)
                  FROM   planets p2)
```

**Figure 2.17.:** Example of an uncorrelated subquery, it does not need information from `p1`

Standing in inner nestings of the onion mentioned in chapter 2.4.2, these subqueries are evaluated over and over again. This is especially painful when dealing with why-provenance in phase 2. For phase 2, these expressions get rewritten to a `toY`-UDF for why-provenance.

The AST representing the query can now be scanned for these UDFs, and their contents can be extracted from those UDFs and moved to the outermost layer of the query. This changes the plan made by the database management system to execute the query. The moved subqueries are executed far less often. The result of this call is then joined at the last layer of the onion.




**(a)** Unoptimized query **(b)** Optimized query

**Figure 2.18.:** `toY` optimization of a phase 2 query

As an additional benefit, duplicates in those uncorrelated expressions can be removed when moving them to the outermost `toY` block.

### 2.5.3. Logging

Normally SQL queries are idempotent: Executing them multiple times, one gets the same result every time. This changes with phase 1 queries. To be exact: This changes with the usage of the `writelog(...)`, `writecase(...)` and `writeagg(...)` user defined functions .[4]

These UDFs trigger a side effect, logging the tuid s of the returned rows and generating a new unique tuidout for this result. When the same query is executed twice, the same result will be returned, because the logging functions will not duplicate log entries, but when the logs are deleted and the phase 1 query is executed again, other tuid s will be returned.

The simplest possibility of logging is to generate new logging tables and fill them with **`INSERT INTO`** statements in the `writelog` UDF. This approach is slow, because these tables are stored on hard drive. So each `writelog`-call triggers a slow write operation to the disk.[psq]

The solution is simple: Use memory to store log tables and not the disk. The PostgreSQL extension `pg_hashtable` was used to achieve this. With this extension every logging table is replaced by a hash-table that is only persisted through the current SQL session and stored in memory.

[4] `writelog` will be used as representative for all three logging functions in the future

# 3 IMPLEMENTATION

This chapter focuses on the underlying SQL dialect and its implementation. It shows how *abstract syntax trees* (ASTs) were used to represent and rewrite SQL queries in Haskell. Therefore, two similar ASTs were created: one to represent "normal" SQL and one to represent the set approach of phase 2 queries (see chapter 2.5.1). This makes it possible to automatically and elegantly rewrite phase 2 queries.

In the second part of the chapter, the focus moves to the explicit translation rules used to rewrite an SQL query to its corresponding phase 1 and phase 2 queries. At last, the promised optimizations and their implementation are explained in detail.

## 3.1. Representing SQL in Haskell

For query rewriting, an option was needed to represent SQL queries in a simple, intuitive and manipulable way. Haskell's algebraic data types are ideal for this, because of their simplicity and composability. Figure 3.1 shows where Haskell comes to use in the computation of data provenance.

Let's come back to the example table planets from figure 2.1 and construct another query: In Haskell, for a simple SQL query like Q2 from figure 3.2a, listing the names of all rocky planets in our solar system, the six constructors from figure 3.3 are needed to build a Haskell AST like the one in figure 3.2b to represent the query in Haskell. Recalling chapter 2.4.2, there is no grouping or aggregation involved in this query. Therefore, only the `Join` constructor is needed.

Each constructor in Haskell represents an SQL construct:

- `Join` represents a `SELECT ... FROM ... WHERE ...` block with possible multiple tables in the `FROM`-clause and their join conditions in the `WHERE`-clause.
- `TableRef` represents the name of a table.
- `ColRef` takes a bound table reference in form of `(Var "p")` and a column name, and represents this specific column.
- `Bind` binds a table or column reference to a new name.

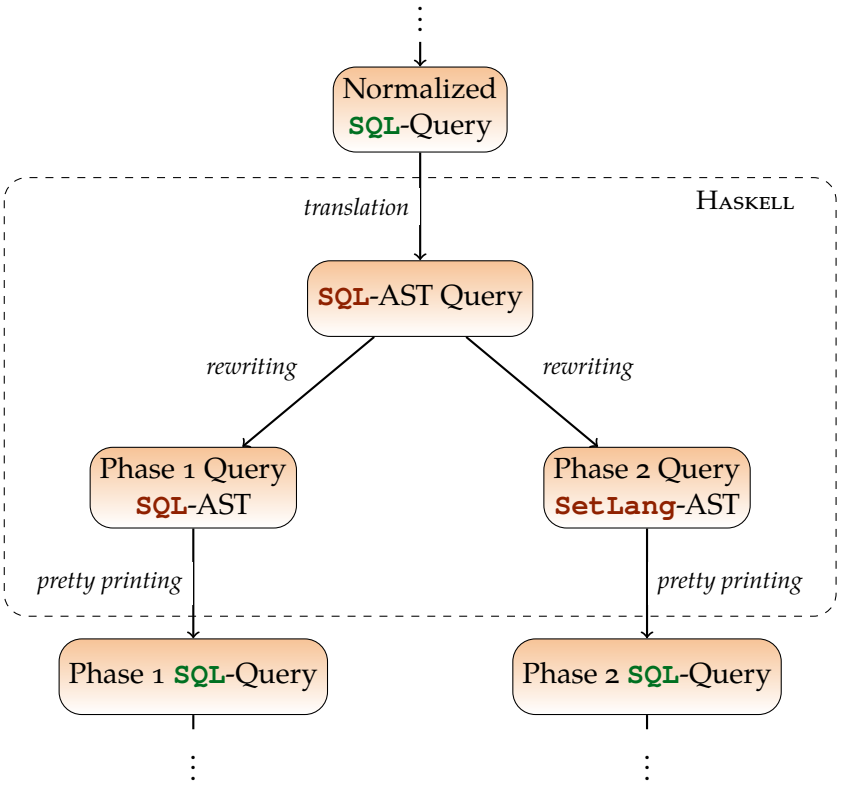



**Figure 3.1.:** Scope of the Haskell implementation

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

**(a)** Q2: List all names of rocky planets

```
Join [Bind (ColRef (Var "p") "name") "name"]
     [Bind (TableRef "planets") "p"]
     (BinOp "=" (ColRef (Var "p") "density")
                (LitS "rocky"))
```

**(b)** Haskell representation of Q2

**Figure 3.2.:** A new SQL query

```
data SQL = Join [SQL]    -- SELECT ...
                [SQL]    -- FROM   ...
                SQL      -- WHERE  ...
         | TableRef Id   -- Id evaluates to a table at runtime
         | ColRef SQL    -- SQL evaluates to a table at runtime
                  Id     -- Id evaluates to a column at runtime
         | Bind SQL      -- Expression to bind
                Id       -- new name of the SQL expression
         | BinOp String  -- Operator: =, <=, like, etc.
                SQL      -- First argument
                SQL      -- Second Argument
         | LitS String   -- String Literal
```

**Figure 3.3.:** Needed constructors for Q2 from figure 3.2a

- **BinOp** takes the string representation of a binary operator like `"+"`, `"-"`, `"*"`, `"/"` or `"like"` and two SQL expressions, and combines the expressions with the operator: $e_1 + e_2$
- **LitS** represents a string literal. There are more literal constructors for integers, dates, booleans etc. Since they only have different behaviours in the translation back to SQL, in the following, they will be refered to as **Lit**.

With a few more constructors, a subset of SQL constructs was implemented, making it possible to build a big subset of SQL queries (see appendix A for a complete overview of the implemented dialect).

To make the Haskell **SQL** dialect clearer and to make manual translation of SQL to the **SQL** AST [5] easier, a query has to be normalized before its manual translation to a Haskell AST.[MDG18] It is easier to focus in each Haskell constructor on the functionality of one SQL construct: joining, grouping, and ordering (see figure 2.8), instead of combining them to a single multi-purpose constructor doing all of this.

This means, it is possible to reference multiple tables only in the innermost **Join** query to represent basic **JOIN** expressions (see chapter 2.4.2 and figure 2.7).[6] With this joined table, it is now possible to group columns (incl. **HAVING**-expressions) and aggregate the other columns with the **Group** constructor. This grouped query can be sorted (**ORDER BY**) in the next step in the **Order** constructor (again see figure 2.7).

[5] no SQL to Haskell parser was implemented because of the vast variety of SQL queries

[6] Constructs like **JOIN**, **INNER JOIN** and **LEFT OUTER JOIN** were not implemented, only the base case **FROM** with multiple tables.

Of course, it is possible to use subqueries. The only requirement is that these queries themselves then must be normalized to fit in the implementation of the **SQL** AST.

### 3.1.1. The **SQL**-Module

Let's take a deeper look into the implemented modules, starting with the base of this work, the **SQL**-module: This module implements the algebraic data type **data SQL** you saw an excerpt from in figure 3.3. This is the base for every query representation in Haskell.

The most important method, the **SQL**-module implements, is the **toString**-method:

```
toString :: SQL -> String
```

This method translates the Haskell representation of an **SQL** query back to valid and executable SQL. Therefore, the module `Text.PrettyPrint.ANSI.Leijen` is used to pretty print the query in a human readable fashion.[ppr]

When you copy the result string to a `psql` console, the query will be executed, given the needed tables, types and functions exist. For query Q2 from figure 3.2a only the table `planets` must exist.

### 3.1.2. The **SetLang**-Module for Phase 2 Queries

To represent phase 2 queries, non-SQL constructs are needed (see chapter 2.5.1). Therefore, a second data type **SL** (short for "set-language") was developed for this work. This new algebraic data type makes query rewriting on set-level possible.

Many constructors look just like their **SQL** pendant, but with **SL** as type. Other constructors like **SQL.Array**, **SQL.Concat**, **SQL.BinOp**, **SQL.And** and **SQL.Or** can be omitted, because they are replaced by the new set union constructor **SL.Union** ($\bigcup$). You find the subset of **data SL** corresponding to figure 3.3 in figure 3.4.[7]

The SQL constructors for literals are not needed in the **SetLang**-module, too, because they are translated to the empty set **SL.Empty** (see the rules in the next chapter 3.2 for details).

To make phase 2 queries executable by the database, the **SetLang** module implements a `toSQL` method, translating constructors with an **SQL** pendant back to

[7] For the complete data type see appendix B

```
data SL = Join [SL] [SL] SL
        | TableRef Id
        | ColRef SL Id
        | Bind SL Id
        | Union [SL] -- new constructor for ⋃
        | Empty      -- the empty set (replaces LitX)
```

**Figure 3.4.:** The subset of **SL** corresponding to figure 3.3

the corresponding constructor and creating an **SQL**-AST. **Union** constructors are replaced by arrays and array concatenation, **Empty** is replaced by an empty array.

This **SQL**-AST now can be normally translated (pretty printed) to executatble SQL with the `toString`-method and executed with PostgreSQL.

## 3.2. Translation and Translation Rules

Translation of queries happen in the Haskell modules **Phase1** and **Phase2**. The **Phase1** module takes a query in **SQL**-AST format and returns its corresponding phase 1 query in the same format. With the `toString` function of the **SQL**-module, the AST gets translated to an executable query.

Phase 2 queries are generated by the **Phase2**-module. This module, again, takes a query in **SQL**-AST format, but returns a query in **SetLang**-AST format. This AST can be translated back to an **SQL**-AST which then can be pretty printed like the phase 1 query.

The modularity in phases 1 and 2 makes the implementation clearer and a better separation of concerns was achieved. There was only one minor complication with this approach: The order of translation of **SELECT**-expressions and **FROM**-expressions mattered, since reading and writing logs must happen in the exact same order in both phases, since log location identifiers are generated with a side effect.

For the translation, the rules from [MDG18] were used as a base. Minor deviations were necessary in a few places. In the following, rules are noted as *inference rules* (see figure 3.5):

Although phases 1 and 2 are implemented separately in different Haskell modules, their rules are denoted in one inference rule, because there would be much duplication otherwise. Hereby $e \mapsto (e^{\mathbb{1}}, e^{\mathbb{2}})$ means that the SQL expression $e$ is translated to $e^{\mathbb{1}}$ in phase 1 and to $e^{\mathbb{2}}$ in phase 2.

$$\frac{\text{premise } 1 \quad \ldots \quad \text{premise } n}{\text{conclusion}} \text{ (Rule Name)}$$

**Figure 3.5.:** Inference rules: When all premises hold, the conclusion also holds

There are two groups of rules: Rules without the need for logging and rules with logging. Let's start with the simpler rules, those without logging:

The most basic expressions in SQL are literals for strings, booleans, integers etc.

$$\frac{}{l \mapsto (l, \emptyset)} \text{ (Lit)}$$

Because a literal is needed to compute the value of the surrounding SQL query, literals are kept for phase 1. In translation to phase 2, they are no longer needed, since only `tuid`s come from the logs and they cannot be combined with these literals to compute a predicate. It is intuitive that the query

```
SELECT 'test' FROM planets
```

has empty provenance sets for each cell, because there were no cells involved in computing `'test'`.

Bound variables for columns and tables on the other hand are needed in both phases 1 and 2:

$$\frac{}{v \mapsto (v, v)} \text{ (Var)}$$

They are needed e.g. to represent the variable `p` from the SQL expression

```
SELECT * FROM planets p.
```

Tables get an index for each phase:

$$\frac{}{\texttt{table} \rightarrowtriangle (\texttt{table}^{\mathbb{1}}, \texttt{table}^{2})} \text{(Table)}$$

In SQL, the table `planets` is translated to `planets_1` in phase 1 and to `planets_2` in phase 2, referencing the generated tables seen in figure 2.5.

Recalling the definition of a column reference in Haskell `ColRef SQL Id`, $e$ is of type `SQL` and $col$ of type `Id`.

$$\frac{e \rightarrowtriangle (e^{\mathbb{1}}, e^{2})}{e.col \rightarrowtriangle (e^{\mathbb{1}}.col, e^{2}.col)} \text{(Col)}$$

So, when $e$ can be translated to $e^{\mathbb{1}}$ and $e^{2}$, the columns can be used on these translated expressions. Keep in mind that $e^{\mathbb{1}}$ references a value in the phase 1 table and $e^{2}$ references a tuid in the phase 2 table.

For binary operators, note how the operator $\oplus$ is kept in phase 1 and replaced by $\cup$ in phase 2, making the `BinOp` constructor obsolete in `SL`:

$$\frac{e_l \rightarrowtriangle (e_l^{\mathbb{1}}, e_l^{2}) \qquad e_r \rightarrowtriangle (e_r^{\mathbb{1}}, e_r^{2})}{e_l \oplus e_r \rightarrowtriangle (e_l^{\mathbb{1}} \oplus e_r^{\mathbb{1}}, e_l^{2} \cup e_r^{2})} \text{(BinOp)}$$

This can be seen in the implementation as well:

```
tr (BinOp op e1 e2) = do
    e1' <- tr e1
    e2' <- tr e2
    return $ BinOp op e1' e2'
```

**(a)** Phase 1 translation

```
tr (S.BinOp _ el er) = do
    el2 <- tr el
    er2 <- tr er
    return $ Union [el2, er2]
```

**(b)** Phase 2 translation

**Figure 3.6.:** Translation of `BinOp` in phases 1 and 2

First the subexpressions `e1` and `e2` are translated. Then, in phase 1, they are combined using the same binary operator that was used in the original query. In phase 2, the operator is discarded and replaced by a generic `Union` constructor.

The next rule toCell is a helper needed to translate subqueries in phases 1 and 2 that return only a single value. Phases 1 and 2 add a tuid -column to the return

$$\frac{e \mapsto (e^{\mathbb{1}}, e^{2}) \qquad [c] := \texttt{columns}(e) \qquad X^{\mathbb{1}} := \begin{array}{ll}\texttt{SELECT} & v.c \\ \texttt{FROM} & e^{\mathbb{1}}\ \texttt{AS}\ v\end{array} \qquad X^{2} := \begin{array}{ll}\texttt{SELECT} & v.c \\ \texttt{FROM} & e^{2}\ \texttt{AS}\ v\end{array}}{\texttt{toCell}(e) \mapsto (X^{\mathbb{1}}, X^{2})}\ (\textsc{toCell})$$

**Figure 3.7.:** Translation of `ToCell` helper

table of a subquery, so the result of the subquery (with two columns) is no longer composable with the rest of the query which expects one column. The toCell-constructor takes this now two column return value, discards the tuid and makes the result single-columned again.

The next two rules (see figures 3.8 and 3.9) are basically the same: In phase 1, the `IN`/`EXISTS` expression is kept, but the right argument is rewritten to a `SELECT` statement which removes the tuid from $e_t^{\mathbb{1}}$. In phase 2, both expressions are replaced by $\bigcup$ which is added in the `SELECT`-clause to collect all tuid s.

Now that we know the more simple rules without logging, let's continue with the more complicated rules with logging.

Starting with the innermost layer of the normalized SQL query (see chapter 2.4.2), we will work our way from the inside out, starting with `Join`, over `Group` to `Order`.

$$\frac{e_s \mapsto (e_s^{\mathbb{1}}, e_s^{2}) \qquad e_t \mapsto (e_t^{\mathbb{1}}, e_t^{2}) \qquad X^{\mathbb{1}} := \begin{array}{ll} e_s^{\mathbb{1}}\ \texttt{IN}\ (\ \texttt{SELECT} & t.col \\ \texttt{FROM} & e_t^{\mathbb{1}}\ \texttt{AS}\ t(\rho, col))\end{array} \qquad X^{2} := \begin{array}{ll} e_s^{2} \cup (\ \texttt{SELECT} & \bigcup t.col \\ \texttt{FROM} & e_t^{2}\ \texttt{AS}\ t(\rho, col))\end{array}}{e_s\ \texttt{IN}\ e_t \mapsto (X^{\mathbb{1}}, X^{2})}\ (\textsc{In})$$

**Figure 3.8.:** Translation of `IN` expression

$$\frac{e \mapsto (e^{\mathbb{1}}, e^{2}) \qquad X^{\mathbb{1}} := \begin{array}{ll}\texttt{EXISTS}\ (\ \texttt{SELECT} & t.col \\ \texttt{FROM} & e^{\mathbb{1}}\ \texttt{AS}\ t(\rho, col))\end{array} \qquad X^{2} := \begin{array}{ll}\texttt{SELECT} & \bigcup t.col \\ \texttt{FROM} & e^{2}\ \texttt{AS}\ t(\rho, col)\end{array}}{\texttt{EXISTS}\ e \mapsto (X^{\mathbb{1}}, X^{2})}\ (\textsc{Exists})$$

**Figure 3.9.:** Translation of `EXISTS` expressions

$$\frac{\begin{array}{l}\ell := fresh() \qquad p \Longmapsto (p^{\mathbb{1}}, p^{2}) \qquad \{f_1, \ldots, f_u\} := v_1, \ldots, v_n \cup fv(p) \\ \quad |e_i \Longmapsto (e_i^{\mathbb{1}}, e_i^{2})|_{i=1,\ldots,m} \qquad |t_i \Longmapsto (t_i^{\mathbb{1}}, t_i^{2})|_{i=1,\ldots,m} \\ X^{\mathbb{1}} := \begin{array}{ll} \texttt{SELECT} & \texttt{writelog}(\ell, f_1.\rho, \ldots, f_u.\rho)\ \texttt{AS}\ \rho, \\ & e_1^{\mathbb{1}}\ \texttt{AS}\ c_1, \ldots, e_m^{\mathbb{1}}\ \texttt{AS}\ c_m \\ \texttt{FROM} & t_1^{\mathbb{1}}\ \texttt{AS}\ v_1, \ldots, t_n^{\mathbb{1}}\ \texttt{AS}\ v_n \\ \texttt{WHERE} & p^{\mathbb{1}} \end{array} \\ X^{2} := \begin{array}{ll} \texttt{SELECT} & v.\rho\ \texttt{AS}\ \rho, \\ & e_1^{2} \cup wh.y\ \texttt{AS}\ c_1, \ldots, e_m^{2} \cup wh.y\ \texttt{AS}\ c_m \\ \texttt{FROM} & t_1^{2}\ \texttt{AS}\ v_1, \ldots, t_n^{2}\ \texttt{AS}\ v_n, \\ & \texttt{LATERAL}\ \texttt{readlog}(\ell, f_1.\rho, \ldots, f_u.\rho)\ \texttt{AS}\ v(\rho), \\ & \texttt{LATERAL}\ \texttt{Y}(p^{2})\ \texttt{AS}\ wh(y) \end{array} \end{array}}{\begin{array}{ll} \texttt{SELECT} & e_1\ \texttt{AS}\ c_1, \ldots, e_m\ \texttt{AS}\ c_m \\ \texttt{FROM} & t_1\ \texttt{AS}\ v_1, \ldots, t_n\ \texttt{AS}\ v_n \quad \Longmapsto (X^{\mathbb{1}}, X^{2}) \\ \texttt{WHERE} & p \end{array}}\ (\textsc{Join})$$

**Figure 3.10.:** Translation of `Join`

`Case` will be the last rule, since it can be used at any place in any nesting level.

Logging is a side effect, introduced by the `writelog` and `readlog` user defined functions.[8] The arguments of those logging functions are a fresh variable $\ell$ as log place identifier and a set of tuid s (denoted as $\rho$ in the rules) to be saved in the log.

The return value of `writelog` is a newly generated tuidout which then is part of the output of the phase 1 query, whereas `readlog` returns a table of tuid s that got logged in phase 1. Because of this, `writelog` happens in the `SELECT`-clause and `readlog` in the `FROM`-clause. You can observe this in the join rule (see figure 3.10).

This is the reason why in the implementation the translation order of `SELECT`, `FROM` and `WHERE` matters: In phase 2, the `FROM` expression has to be evaluated before the `SELECT` expression to ensure the correct logging locations in subqueries.

Note, too, how the predicate $p$ from the `WHERE` expression wanders to `FROM` in phase 2 to be part of the why-provenance set which then is part of every column in the `SELECT`-clause.

Now that we have joined the tables, we can group the result (see figure 3.11). Because of the normalization, there is exactly one table in the `FROM`-clause. The Group rule takes care of this grouping by rewriting all three of the `GROUP BY` clause, the aggregates (using the Agg rule from figure 3.12) and the `HAVING` clause.

[8] these function names are used as placeholders for all logging functions `readlog`/`writelog`, `readagg`/`writeagg`, and `readcase`/`writecase`

$$p \rightmapsto (p^{\mathbb{1}}, p^{2})$$

$$\ell := fresh() \qquad \{f_1, \ldots, f_u\} := fv(e_1) \cup \ldots \cup fv(e_n) \cup fv(p)$$

$$|e_i \rightmapsto (e_i^{\mathbb{1}}, e_i^{2})|_{i=1,\ldots,n} \qquad |a_i \rightmapsto (a_i^{\mathbb{1}}, a_i^{2})|_{i=1,\ldots,m} \qquad t \rightmapsto (t^{\mathbb{1}}, t^{2})$$

$$X^{\mathbb{1}} := \begin{array}{ll} \texttt{SELECT} & \texttt{writeagg}(\ell, \texttt{array\_agg}(v.\rho), f_1.\rho, \ldots, f_u.\rho) \\ & a_1^{\mathbb{1}} \texttt{ AS } c_1, \ldots, a_m^{\mathbb{1}} \texttt{ AS } c_m \\ \texttt{FROM} & t^{\mathbb{1}} \texttt{ AS } v \\ \texttt{GROUP BY} & e_1^{\mathbb{1}}, \ldots, e_n^{\mathbb{1}} \\ \texttt{HAVING} & p^{\mathbb{1}} \end{array}$$

$$X^{2} := \begin{array}{ll} \texttt{SELECT} & v.\rho \\ & v.c_1 \cup v.why \texttt{ AS } c_1, \ldots, v.c_m \cup v.why \texttt{ AS } c_m \\ \texttt{FROM} & (\texttt{SELECT} \quad \texttt{the}(l.\rho) \texttt{ AS } \rho \\ & \qquad\quad a_1^{2} \texttt{ AS } c_1, \ldots, a_m^{2} \texttt{ AS } c_m \\ & \qquad\quad \texttt{Y}\left(\bigcup(e_1^{2}) \cup \ldots \cup \bigcup(e_n^{2})\right) \texttt{ AS } why \\ & \texttt{FROM} \quad t^{2} \texttt{ AS } v \\ & \qquad\quad \texttt{readagg}(\ell, v.\rho, f_1.\rho, \ldots, f_u.\rho) \texttt{ AS } l \\ & \qquad\quad \texttt{Y}(e_1^{2} \cup \ldots \cup e_n^{2}) \texttt{ AS } wh(y) \\ & \texttt{GROUP BY} \quad l.\rho) \texttt{ AS } v \end{array}$$

$$\frac{\begin{array}{ll} \texttt{SELECT} & a_1 \texttt{ AS } c_1, \ldots, a_m \texttt{ AS } c_m \\ \texttt{FROM} & t \texttt{ AS } v \\ \texttt{GROUP BY} & e_1, \ldots, e_n \\ \texttt{HAVING} & p \end{array} \quad \rightmapsto (X^{\mathbb{1}}, X^{2})}{} \text{ (GroupBy)}$$

**Figure 3.11.:** Translation of `Group`

Here **writeagg** logs an array of tuid s in phase 1, but creates a row in the log for each of these tuid s using **UNNEST** in the UDF, so when using **readagg** in phase 2, no aggregation is needed.

You can see, the **HAVING**-clause is no longer needed in phase 2 because of the lack of values in the phase 2 table. Instead, the predicate wanders to the **FROM** clause as why-provenance.

In the **SELECT**-clause of $X^2$, the attentive reader spotted the non-SQL construct **the**. This is another helper in Haskell, denoting that there is no aggregation in SQL, because the column itself is part of the **GROUP BY**-clause. It is needed, because Haskell needs explicit aggregates in $a_1, \ldots, a_m$. It will be removed when translating a phase 2 query back to SQL.

Readers, knowing our approach and the related rule set, spotted a deviation from the rules from [MDG18] in the **Group** rule. This was necessary, because automatically using the rule from this work resulted in a runtime error (`ERROR: aggregate functions are not allowed in functions in FROM`) when working with **HAVING**-expressions.

These expressions contain aggregates, so the **toY**-function cannot use them in **FROM**. To avoid this error, a nested approach was used, to compute why-provenance in the inner **SELECT** once, and use it multiple times in the outer **SELECT**.

Like teasered earlier, the Agg rule takes care of the translation of aggregates:

$$\frac{e \mapsto (e^{\mathbb{1}}, e^{\mathbb{2}})}{\mathbf{AGG}(e) \mapsto (\mathbf{AGG}(e^{\mathbb{1}}), \bigcup e^{\mathbb{2}})}\ (\textsc{Agg})$$

**Figure 3.12.:** Translation of aggregates

In phase 1, the aggregate is kept and executed on $e^{\mathbb{1}}$, while in phase 2, a set union is performed to collect all tuid s that were involved in the aggregate.

The Order rule from figure 3.13 looks more complicated than it is. It replicates the original query in phase 1, ordering the results in a nested query, so the results are in the right order when logging them. Phase 2 can now retrieve the results in the right order, making the **Order** constructor obsolete.

For **CASE**, there are two different syntactic options: For the first option every statement is independently evaluated like seen in figure 3.14. The expressions `a` and `b` are evaluated to booleans. This is the more flexible approach.

$$
\frac{
\begin{array}{l}
q \rightrightarrows (q^{\mathbb{1}}, q^{2}) \\
\ell := fresh() \qquad |e_i \rightrightarrows (e_i^{\mathbb{1}}, e_i^{2})|_{i=1,\ldots,n} \qquad |o_i \rightrightarrows (o_i^{\mathbb{1}}, o_i^{2})|_{i=1,\ldots,m} \\
\\
X^{\mathbb{1}} := \begin{array}{lll}
\texttt{SELECT} & \texttt{writelog}(\ell, t.\rho) & , \\
 & t.c_1^{\mathbb{1}} \texttt{ AS } c_1, \ldots, t.c_n^{\mathbb{1}} \texttt{ AS } c_n & \\
\texttt{FROM} & (\texttt{SELECT} \quad t.\rho \texttt{ AS } \rho, & \\
 & \qquad\qquad e_1^{\mathbb{1}} \texttt{ AS } c_1, \ldots, e_n^{\mathbb{1}} \texttt{ AS } c_n & \\
 & \texttt{FROM} \quad\; q^{\mathbb{1}} \texttt{ AS } t & \\
 & \texttt{ORDER BY } o_1^{\mathbb{1}}, \ldots, o_m^{\mathbb{1}} & \\
 & \texttt{LIMIT} \quad\; l) \texttt{ AS } t &
\end{array} \\
\\
X^{2} := \begin{array}{ll}
\texttt{SELECT} & filter.\rho \texttt{ AS } \rho, \\
 & e_1^{2} \cup Y_{filter} \texttt{ AS } c_1, \ldots, e_n^{2} \cup Y_{filter} \texttt{ AS } c_n \\
\texttt{FROM} & q^{2} \texttt{ AS } t, \\
 & \texttt{LATERAL readlog}(\ell, t.\rho) \texttt{ AS } filter(\rho), \\
 & \texttt{LATERAL Y}(o_1^{2} \cup \ldots \cup o_m^{2}) \texttt{ AS } Y_{filter}
\end{array}
\end{array}
}{
\begin{array}{ll}
\texttt{SELECT} & e_1 \texttt{ AS } c_1, \ldots, e_n \texttt{ AS } c_n \\
\texttt{FROM} & q \texttt{ AS } t \\
\texttt{ORDER BY} & o_1, \ldots, o_m \\
\texttt{LIMIT} & l
\end{array}
\rightrightarrows (X^{\mathbb{1}}, X^{2})
}
\text{(OrderBy)}
$$

**Figure 3.13.:** Translation of `Order`

```
CASE
  WHEN a THEN a1
  WHEN b THEN b1
  ELSE c1
END
```

**Figure 3.14.:** `CASE` expression with independently evaluated branches

The second option is `CASEQ` (read "case eq"): one general statement is evaluated once and the result is checked against a list of possible outcomes (see figure 3.15). Here, the expression `x` is checked against first `a` and then `b`, producing `True` when they match, and `False` when they do not.

Only the first option is implemented, because the second option can be translated to the first. However, the second option is used in the implementation of both phase 1 and 2 translations (see figure 3.16).

$X^{w\mathbb{1}}$ evaluates the `WHEN` expressions $e_i^w$ from the `CASE` and returns the number of the branch that was evaluated to `True`. This number is logged in phase 1 with `writecase`, returning the logged number, so the `CASEQ`-clause can be evaluated and evaluates to the right $e_i^{t\mathbb{2}}$ expression.

In phase 2, `readcase` reads the logged values from the log and can perform the correct branch, too. The `Y`-statement then performs a union over the predicates of all failed branches, because every one of these took part in the evaluation of the evaluated branch.

Now we know how the translation from an SQL query to its provenance queries works. Before we can come to optimization, let's take a quick look at logging.

```
CASE x
  WHEN a THEN a1
  WHEN b THEN b1
  ELSE c1
END
```

**Figure 3.15.:** `CASE` expression with one statement which is checked against multiple possible outcomes

$$\frac{\begin{array}{c}\ell := fresh() \qquad |e_i^w \Mapsto (e_i^{w\mathbb{1}}, e_i^{w2})|_{i=1,\ldots,n} \\ |e_i^t \Mapsto (e_i^{t\mathbb{1}}, e_i^{t2})|_{i=1,\ldots,n} \qquad \{f_1,\ldots,f_u\} := fv(e_1^w) \cup \ldots \cup fv(e_n^w)\end{array}}{\begin{array}{l} X^{w\mathbb{1}} := \begin{array}{l}\texttt{CASE}\ \texttt{WHEN}\ e_1^{w\mathbb{1}}\ \texttt{THEN}\ 1 \\ \quad\vdots \\ \quad\texttt{WHEN}\ e_n^{w\mathbb{1}}\ \texttt{THEN}\ n \\ \quad\texttt{ELSE}\ 0 \\ \texttt{END}\end{array} \\ X^{\mathbb{1}} := \begin{array}{l}\texttt{CASEQ}\ \texttt{writecase}(\ell, f_1,\ldots,f_u, X^{w\mathbb{1}}) \\ \quad\texttt{WHEN}\ 1\ \texttt{THEN}\ e_1^{t\mathbb{1}} \\ \quad\vdots \\ \quad\texttt{WHEN}\ n\ \texttt{THEN}\ e_n^{t\mathbb{1}} \\ \quad\texttt{ELSE}\ e_0^{t\mathbb{1}} \\ \texttt{END}\end{array} \\ X^{2} := \begin{array}{l}\texttt{CASEQ}\ \texttt{readcase}(\ell, f_1,\ldots,f_u) \\ \quad\texttt{WHEN}\ 1\ \texttt{THEN}\ e_1^{t2} \cup \texttt{Y}(e_1^{w2}) \\ \quad\vdots \\ \quad\texttt{WHEN}\ n\ \texttt{THEN}\ e_n^{t2} \cup \texttt{Y}(e_1^{w2} \cup \ldots \cup e_n^{w2}) \\ \quad\texttt{ELSE}\ e_0^{t2} \cup \texttt{Y}(e_1^{w2} \cup \ldots \cup e_n^{w2}) \\ \texttt{END}\end{array}\end{array}}{\begin{array}{l}\texttt{CASE}\ \texttt{WHEN}\ e_1^w\ \texttt{THEN}\ e_1^t \\ \quad\vdots \\ \quad\texttt{WHEN}\ e_n^w\ \texttt{THEN}\ e_n^t \Mapsto (X^{\mathbb{1}}, X^{2}) \\ \quad\texttt{ELSE}\ e_0^t \\ \texttt{END}\end{array}}\ (\textsc{Case})$$

**Figure 3.16.:** Translation of `Case`

```
-- logging table
CREATE TABLE log1 (location tloc NOT NULL,
                   tuidout tuid NOT NULL,
                   tuid1 tuid NOT NULL);
ALTER TABLE log1 ADD PRIMARY KEY (location, tuid1);
ALTER TABLE log1 ALTER COLUMN tuidout
        SET DEFAULT NEXTVAL('tuid_seq');

-- write
CREATE FUNCTION writelog(v_location tloc, v_tuid1 tuid)
RETURNS tuid AS
$$
DECLARE
    res tuid;
BEGIN
    INSERT INTO log1 (location, tuid1)
        VALUES (v_location, v_tuid1)
        RETURNING tuidout INTO res;
        RETURN res;
EXCEPTION
    WHEN UNIQUE_VIOLATION THEN
        -- inlining of readlog() for
        -- improved PostgreSQL performance
        RETURN(SELECT l.tuidout
                 FROM log1 AS l
                WHERE l.location=v_location
                  AND l.tuid1=v_tuid1);
END;
$$ LANGUAGE PLPGSQL VOLATILE;

-- read
CREATE FUNCTION readlog(v_location tloc, v_tuid1 tuid)
RETURNS TABLE(tuid tuid) AS
$$
    SELECT l.tuidout
      FROM log1 AS l
     WHERE l.location=v_location
       AND l.tuid1=v_tuid1
$$ LANGUAGE SQL STABLE;
```

**Figure 3.17.:** Implementation of `log1` as example for logging with tables

## 3.3. Logging

Logging is needed to communicate between phases 1 and 2. It is implemented with PostgreSQL's user defined functions (see figure 3.17 for an example), using a table for each log size (e.g. `Join` needs to log a tuid for every table of the join).

In phase 1, `writelog` creates a new row in the logX table, creating a new tuidout for each row. Duplicates are not logged because of a primary key constraint in the logX table, instead the already present tuidout is returned by the UDF.

Phase 2 reads from those logging tables with the `readlog` function. This function filters the logging table for the primary key v_location and v_tuid1 and returns exactly one row because of this constraint.

There are four scenarios that need logging: joining multiple tables like explained above, aggregating columns with a `GROUP BY` expression (see chapter 2.4.3), ordering tables with `ORDER BY` and using `CASE` expressions (see chapter 3.2).

## 3.4. Optimization Implementation

Optimization is possible in both phases 1 and 2 (see figure 3.18). The approach is different in both phases. In phase 2, optimization was implemented making use of query rewriting, while in phase 1, optimization is possible using log optimization.

This chapter starts with phase 2 and query rewriting to optimize set union and why-provenance. Both optimizations are implemented in separate Haskell modules. Then, both are combined in an `Optimizer` module, so only `Optimizer.optimize` has to be invoked to completely optimize a query.

This `Optimizer` module easily can be extended by additional optimizations, working as a flexible experimentation platform for optimization using query rewriting. With this module, it is possible to switch different optimizations on and off for additional insight in each approach.

The chapter then ends with phase 1 and logging optimization with hash-tables.

### 3.4.1. Set Union Optimization

The `SetUnionOptimizer`-module implements the optimization for set union optimization recursively. Recalling chapter 2.5.1, PostgreSQL does not know about the missmatch between arrays and the sets they represent. The optimization starts with

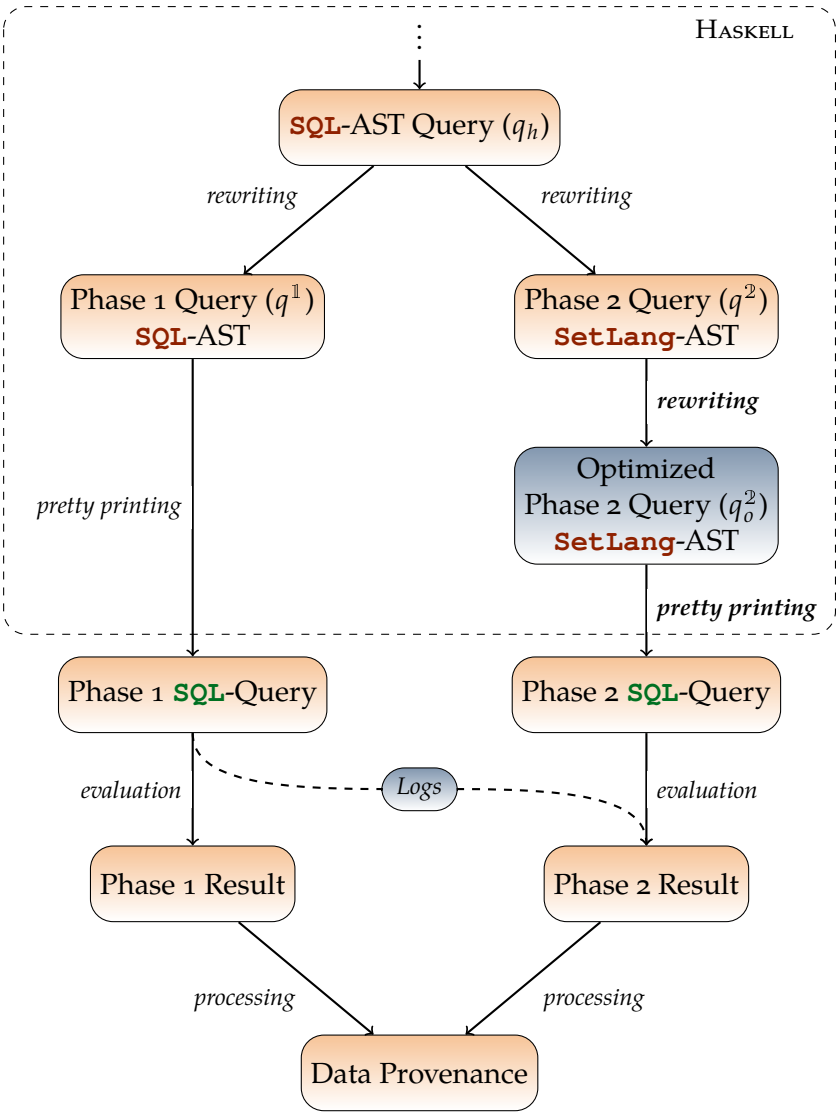



**Figure 3.18.:** Optimization locations

**Join** and **Group**, rebuilding the same constructors and optimizing their arguments:

```
opt (Join s f w) = Join (map opt s) (map opt f) (opt w)
```

Every constructor that has an argument of type **SL** now reconstructs itself with the optimized argument:

```
opt (ColRef ex id) = ColRef (opt ex) id
```

The expressions without **SL** arguments abort the recursion:

```
opt ex = ex
```

The optimization happens only in the **Union** constructors in 4 steps (see figure 3.19). In the first step, nested unions get flattened: e.g. the expression `Union [a, Union a, b]` gets translated to `Union [a, a, b]`. The second step removes duplicates from the resulting array, removing one of the `a`s in the example.

When there are optimizable expressions left in the array, they are optimized in the third step. The last two steps basically do the same and are only separated for better readability: they remove empty expressions and empty **Union** expressions.

### 3.4.2. Why-Provenance Optimization

The optimization of why-provenance (see chapter 2.5.2) is a bit more complex. It is implemented in the Haskell module **ToYOptimizer**. This module optimizes why-provenance in four steps:

```
opt u@(Union _) = let
    xs0 = flattenUnion u -- flatten nested unions
    xs1 = nub xs0 -- remove duplicate expressions
    xs2 = map opt xs1 -- optimize subexpressions
    xs3 = filter (/=Empty) xs2 -- remove empty results
    xs4 = filter (/=(Union [])) xs3 -- remove empty results
  in Union xs4
```

**Figure 3.19.:** Set union optimization

In **step 1**, all why-expressions (`Udf "toY" ...`) in the query $q$ are listed. Therefore, the input query is exploded: a list is recursively generated listing every expression and every subexpression of the query. This list is filtered for the `toY`-UDF.

On the resulting list, **step 2** is executed which filters the remaining why-expressions for uncorrelated expressions. This is achieved by searching free variables in every expression in the list and returning only those expressions which do not have such free variables.

In **step 3**, these uncorrelated expressions are removed from every why-expression in $q$. This step is subdivided into two steps: finding the expressions from the list in query $q$ and replacing them with `Empty`.

**Step 4** adds the found uncorrelated why-expressions to the outermost why-expression. To achieve this, the function has to take into account that different outermost constructors use different notations of why-provenance: `Join` and `Group` use `wh(y)`, `Order` uses `filter(y)`.

This four step procedure additionally removes duplicated uncorrelated expressions from why-provenance, giving another potential performance boost.[9]

### 3.4.3. Replacing Log Tables with Hash-Tables

The idea behind the replacement of database tables with hash-tables (see chapter 2.5.3) is that writing to and reading from tables stored on hard disk is slow. The PostgreSQL extension `pg_hashtable` of the Research Group works with hash-tables stored in RAM.[Bur21] RAM is faster to write to and read from.

So, the log tables are no longer needed and the functions are modified to working with hash-tables instead (see figure 3.20). Hash-tables have an integer as identifier, separating different hash-tables from each other. Every function (in this case `scanHT` and `lookupHT`) on hash-tables has this identifier as their first argument.

The `lookupHT` function is used in its writing state (this is stated by the `true` in the second argument). This means, all further arguments are written in the hash-table when there is no duplicate of it already in the table. Comparing this implementation to the implementation seen in figure 3.17, one can see that the last arguments match.

However, we have learned that `lookupHT` is not well suited for accessing log entries in phase 2. So in `readlog`, the `scanHT` function (with $O(n)$) has to be used to return the log entry, nullifying the second advantages of hash-tables: making

[9] since no implemented query made use of it, this is only an assumption

```
-- read
CREATE FUNCTION readlog(v_location tloc, v_tuid1 tuid)
RETURNS TABLE(tuid tuid) AS
$$
  SELECT tuidout
  FROM scanHT(1) AS l(location tloc, tuid1 tuid, tuidout tuid)
  WHERE l.location=v_location
    AND l.tuid1=v_tuid1;
$$ LANGUAGE SQL STABLE;

-- write
CREATE FUNCTION writelog(v_location tloc, v_tuid1 tuid)
RETURNS tuid AS
$$
  SELECT tuidout
  FROM lookupHT(1, true, v_location,
                v_tuid1, NEXTVAL('tuid_seq')::tuid)
       AS _(location tloc, tuid1 tuid, tuidout tuid);
$$ LANGUAGE SQL STABLE;
```

**Figure 3.20.:** Implementation of `log1` as example for logging with hash-tables

it possible to access log entries in $O(1)$. This is because there was a bug in the implementation of the extension that bloated the memory and made it impossible to use for log retrieval.

But even this workaround did not make it possible to execute all implemented phase 2 queries within reasonable time. Since phase 1 did overall profit from logging with hash-tables, we wanted to keep hash-tables for phase 1, but switch back to normal tables for phase 2.

So, an approach was developed, to compensate for that: Phase 1 is executed with hash-tables and the resulting logs are materialized in database tables, so phase 2 can read from those tables.

# 4 EXPERIMENTS

This chapter shows the results of the experiments and explains our observations. To understand the method of the experiments, first the TPC-H benchmark will be described and the setup of the used system and database will be shown. Then, the results for query rewriting optimizations of phase 2 will be presented, followed by the results for logging optimization in phase 1.

## 4.1. TPC-H Benchmark

In this work, the queries from TPC-H benchmark in version 2.18.0 are used to make the results comparable to results from other works of the Database Research Group, e.g. [Mül20] and [MDG18]. These 22 queries are one part of the *decision support benchmark* TPC-H.[tpc] The queries are designed to work with large amounts of data and show how the database behaves with complex business queries.[10]

Using TPC-H's database generator tool with scale factor 1, an instance of a data set of approximately 1 GB of relational data was created. This data fills eight tables: small tables for e.g. `countries` and `nations` and big tables for e.g. `orders` and `lineitems`. The `lineitems` table has about six million rows.

The benchmark's query generator was used to generate the templates of these queries. They got manually rewritten to Haskell ASTs as described in chapter 3.

In this work, 19 out of 22 queries were implemented. For the remaining queries the dialect should have been expanded to support **`LEFT OUTER JOIN`** for Q13, database views for Q15 and **`DISTINCT`** for Q16.

## 4.2. Database Setup

PostgreSQL 13.5 was chosen as the underlying relational database management system (RDMBS). It is open source and has a very good documentation.[psq] Version 13.5 was the current version of PostgreSQL, when taking the measurements.

[10] The other part of the benchmark are concurrent data modifications which are irrelevant for this work.

The RDBMS ran on an ubuntu 20.04 LTS machine with a Linux 5.4 kernel. The system had 72GB of RAM and two Intel Xeon CPUs X5570 with 2.93GHz each.

PostgreSQL was configured to use 8192MB of shared buffers and 64MB of working memory. For the remaining configuration, the standards were kept. As the only supplement to standard PostgreSQL, the `pg_hashtable` extension was installed.[Bur21, chapter 4.1.2]

To separate the experiments from other works relying on PostgreSQL, a new database `tpch-s1` was generated.

## 4.3. Experiment Setup

For the experiment setup and the actual experiments bash scripts and SQL scripts were used for automation. Each script can be invoked with a hash-table option `-h` to work with hash-tables as log instead of database tables.

The first thing to do when starting a new experiment is to create a new instance of a 1 GB sized dataset for TPC-H with the benchmarks `dbgen`-tool. Any previously generated `tpch-s1` database in PostgreSQL gets dropped and recreated.

In the next step, the logging tables and functions are generated (again, `-h` specifies whether they are generated with database tables or with hash-tables). The helper-functions for why-provenance, duplication removal, and efficient array concatenation are generated as well.

Then, the database `tpch-s1` gets filled with the generated data from the new TPC-H instance. Primary keys and foreign keys for each table are initialized and indizes on common columns generated as recomended.[Mül20, page 157]

After the setup of the TPC-H-tables, the corresponding phase 1 and 2 tables are initialized. The phase 1 tables basically copy the TPC-H tables adding a globally unique row identifier (denoted as `tuid` in SQL and $\rho$ in the rules in chapter 3.2):

```
CREATE TABLE region_1 AS
SELECT nextval('tuid_seq')::int tuid, * FROM region;
```

**Figure 4.1.:** example of phase 1 table creation (`region_1`) from table `region`

Since the data stays the same, the indizes, primary keys and foreign keys from the normal tables are duplicated for the corresponding phase 1 tables.

```
CREATE SEQUENCE annot_seq start 1;

CREATE TABLE region_2 AS
SELECT tuid
     , array[nextval('annot_seq')::int] R_REGIONKEY
     , array[nextval('annot_seq')::int] R_NAME
     , array[nextval('annot_seq')::int] R_COMMENT
FROM region_1;
```

**Figure 4.2.:** example of phase 2 table creation (`region_2`) from table `region_1`, duplicating table `regions` columns

Phase 2 tables replace every value (except for the row identifier `tuid`) in a phase 1 table by an array with one value of the annotation sequence (see figures 2.5b and 4.2). Since these tables are queried only with their primary key, it is neither neccessary nor helpfull to provide additional indizes for phase 2 tables.

In the last step, the queries for the experiment are generated with Haskell, translating the AST representation of the TPC-H queries to their corresponding phase 1 and 2 queries as described in chapters 2.4.3 and 3.2 to ensure the experiment works with the latest changes in the underlying model.

## 4.4. Experiment Structure

The experiments themselves are divided in four steps:

**Step 1 (Preparation)** Setup the database with the setup, that was described in chapter 4.3.

**Step 2 (Phase 1)** Execute each phase 1 query five times "to compensate for network events and other pseudo-random effects"[Mül20, chapter 9.2.1] and measure the execution times.

**Step 3 (Log size)** Execute each phase 1 query once more and count the log lines written by each of those queries.

**Step 4 (Phase 2)** Execute each phase 2 query ten times (five times without optimization and five times with) and measure the execution times.

These four steps are executed twice, one time with normal logging and one time with hash-tables as logs. There was one exception: Step 4 was not executed with hash-tables, because of the adversities described in chapter 3.4.3.

## 4.5. Optimization Results

The results of these experiments are summarized in the figures on the following pages. We start again with phase 2 and the results for query rewriting. Then, the results for logging optimization with hash-tables in phase 1 will be presented and discussed. In the end, the overall results for all optimizations together will be discussed and explanations for them will be provided.

### 4.5.1. Results from Query Rewriting (Phase 2)

On the far left in figure 4.3, you can see the enormous impact that query rewriting had on $Q22^2$ (compare timings of ■ for unoptimized queries with ■ for optimized queries), making the query execute in ten seconds instead of thirteen minutes. $Q18^2$, $Q19^2$, $Q12^2$, too, get a good boost between 15% and 60%.

Then, there are many queries ($Q6^2$ to $Q5^2$) getting only small boost around 2% to 5%. Next, the optimization does not have a real impact on queries between $Q17^2$

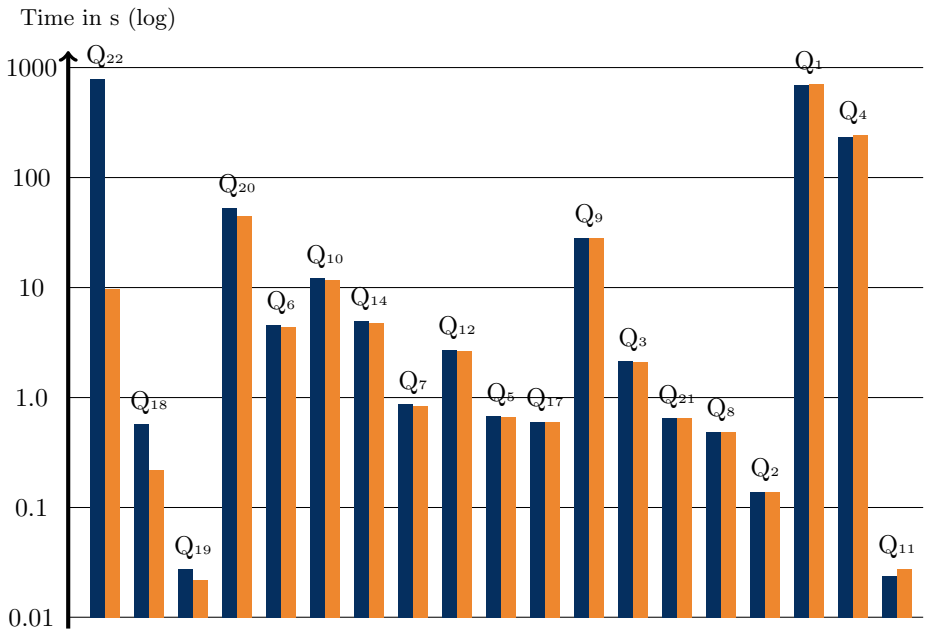



**Figure 4.3.:** Time in milliseconds in phase 2 with (■) and without (■) query rewriting, ordered by the relative boost which they get from the optimization (see figure 4.4)

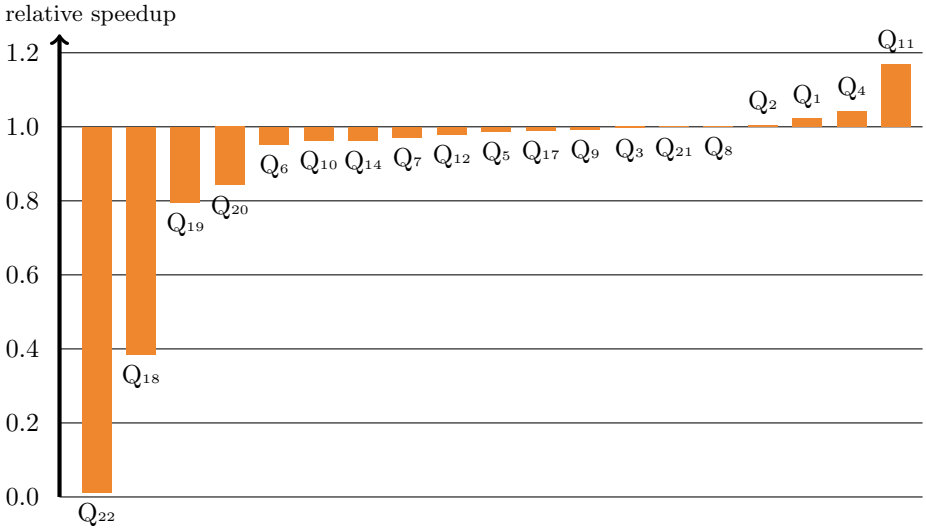



**Figure 4.4.:** Relative impact of query rewriting in phase 2 (order criterion for figure 4.3)

and $Q2^2$. The last three queries on the far right get a small penalty: $Q1^2$ gets a 2% penalty, $Q4^2$ is 4% slower (this translates back to a few seconds) and $Q11^2$ gets 16% slower. However, these 16% are only a few milliseconds which is nothing compared to the minutes win on the far left.

Query $Q22^2$ profits from why-provenance optimization (see chapter 3.4.2), since it has an uncorrelated subquery in the `WHERE` expression. This expression is extracted to the outermost computation of why-provenance. Queries $Q18^2$ and $Q20^2$ also profit from why-provenance optimization.

$Q19^2$, on the other hand, gets its boost from set-union optimization. It has many terms in the `WHERE` expression, producing a huge array with many duplicates and `Empty` placeholders for literals in phase 2. These duplicates and placeholders are removed by the set-union optimizer, accelerating the query exectution.

But what happened to $Q11^2$ on the far right? This query is the only query having an uncorrelated subquery in a `HAVING` expression. So, it could be wise to disable why-provenance optimization in `HAVING` expressions in the future.

All things considered, query rewriting accelerates phase 2 in many cases. In cases where it slows down query execution, the slowdown can be accepted to get a better runtime for the other queries.

### 4.5.2. Results from Logging with Hash-Tables (Phase 1)

The results of phase 1 optimization can be summarized in similar figures like in phase 2. In figure 4.5, you can see the measurements of phase 1 without (■) and with (■) hash-tables.

When measuring phase 1 with hash-tables, there are three steps measured together: Preparation of the hash-tables, execution of the phase 1 query and materialization of the logs to make the logs of phase 1 usable in phase 2 (see chapter 3.4.3). Without hash-tables, only the execution of the query is measured.

Again, on the far left, we can see a huge performance boost between 37% and 83% for queries $Q21^{\mathbb{1}}$, $Q17^{\mathbb{1}}$ and $Q4^{\mathbb{1}}$. Then, there are the queries from $Q22^{\mathbb{1}}$ to $Q20^{\mathbb{1}}$ which get a decent boost of 10% to 20%. Since the queries do not change when switching to hash-tables, and since the execution plans made by PostgreSQL do not change much either, this boost can be traced back to hash-tables staying in memory instead of being written to disk.

In the middle between $Q10^{\mathbb{1}}$ and $Q18^{\mathbb{1}}$, hash-tables have no real impact on the execution times. The last queries get a huge penalty between 6% up to 161%. How-

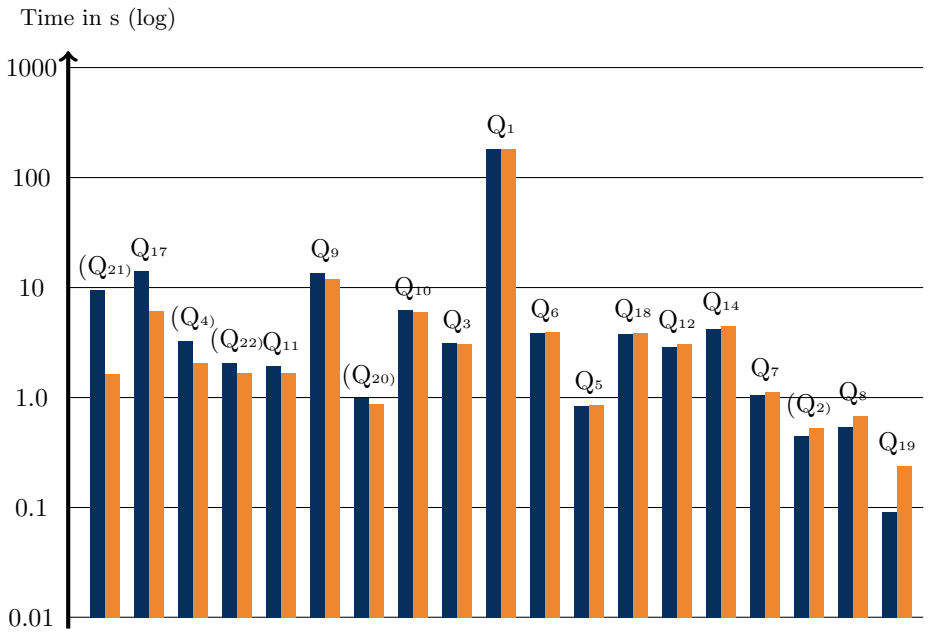



**Figure 4.5.:** Time in milliseconds in phase 1 with (■) and without (■) hash-tables, ordered by the relative boost which they get from the optimization (see figure 4.6)

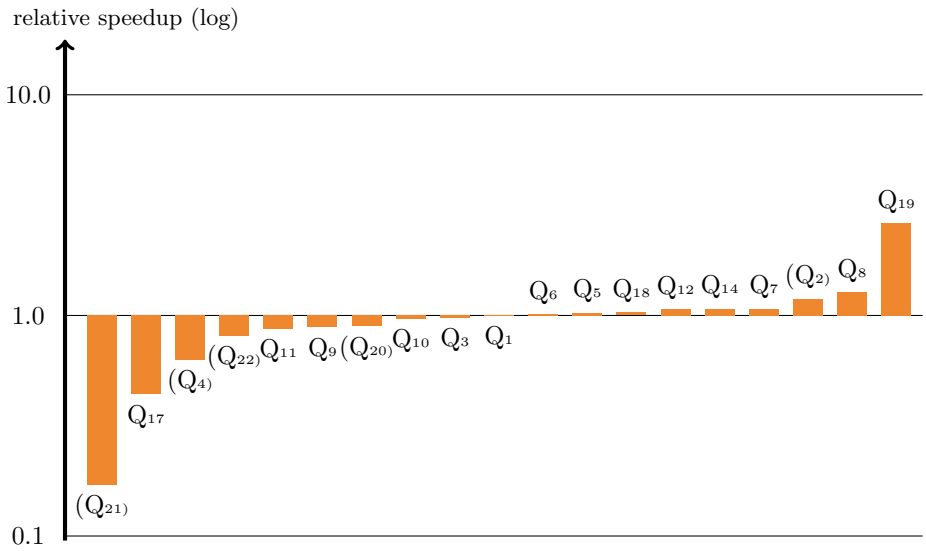



**Figure 4.6.:** Relative impact of switch to hash-tables in phase 1 (order criterion for figure 4.5)

ever, these penalties are only a few hundred milliseconds. Example measurements showed that the materialization of the hash-tables takes approximately 100ms. This did not really change with different log lengths that were materialized.

So, the penalty on the far right can be ommited in the future when the `pg_hashtable` plugin gets modified to properly work in phase 2 as well. The queries in the middle then will get a small performance boost, too.

Queries using **`IN`** or **`EXISTS`** expressions can have different log sizes after the switch to hash-tables, since the PostgreSQL query planner changes the plans for these queries. This results in different orders, the database management system iterates over the subquery results, so it finds different elements at different times. Because of this, the measurements of queries Q2, Q4, Q20, Q21, and Q22 cannot be compared to the results with database logging.[11]

### 4.5.3. Cumulated Results from All Optimizations

Looking at all queries, and analyzing which phase had how much impact on the overall performance, we come to figure 4.7.

Adding the results from both phases in figure 4.8, we can see the overall impact of our approaches of provenance optimization for the queries of the TPC-H benchmark.

[11] This is denoted by brackets in figure 4.5, figure 4.6, and figure 4.8

| Impact | Phase 1 | Phase 2 | Phases 1 + 2 |
|---|---|---|---|
| ↑ | Q4, Q17, Q21 | Q18, Q19, Q20, Q22 | Q11, Q17, Q20, Q21, Q22 |
| ↗ | Q9, Q11, Q20, Q22 | Q6, Q7, Q10, Q12, Q14 | Q9, Q10, Q18, Q19 |
| → | Q1, Q3, Q5, Q6, Q7, Q10, Q12, Q14, Q18 | Q2, Q3, Q5, Q8, Q9, Q17, Q21 | Q1, Q3, Q4, Q5, Q6, Q7, Q12, Q14 |
| ↘ | Q2, Q8 | Q1, Q4 | Q2, Q8 |
| ↓ | Q19 | Q11 | Q19 |

**Figure 4.7.:** Performance impact from high boost (↑) to high penalty (↓)

On the far left, query Q22 with an overall speedup of 99% hugely profits from query rewriting with a little support from hash-tables. Query Q21 on the other hand profits from phase 1 optimization. Likewise, Q17 gets its perfomance boost only from phase 1, while Q20 gets its boost from both phases. For Q11, the boost from phase 1 compensates for the penalty in phase 2, giving the query an overall boost of 13%.

On the right, query Q19 suffers from its penalty from phase 1, although it gets a decent boost in phase 2. Q2 and Q8 neither profit from nor loose in phase 2, and so keep their penalty from phase 1 (again, see figure 4.7).

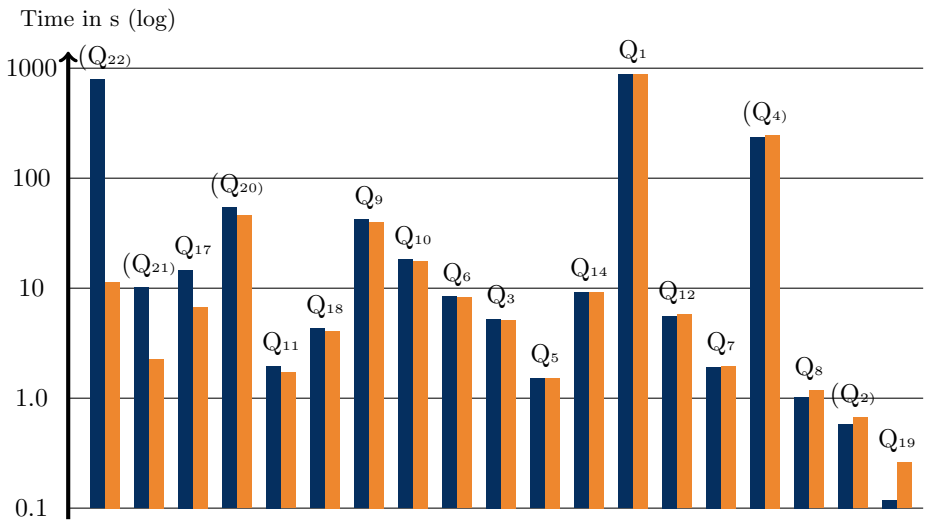



**Figure 4.8.:** Added results from logging with hash-tables in phase 1 and query rewriting in phase 2. Unoptimized (■), optimized (■)

# 5 CONCLUSION & OUTLOOK

## 5.1. Summary

In this work, we examined the impact of optimization in data provenance for where- and why-provenance. We took the two-phase approach from the Research Group from the University of Tübingen and applied different techniques in both phases.

In phase 1, logging was optimized by switching from logging with database tables to the `pg_hashtable` plugin and to logging with hash-tables. Query rewriting was the tool of choice in phase 2. Two optimizations were implemented: the first removes duplication in arrays and the second reallocates expensive why-provenance from often executed inner positions to less often executed outer positions.

These approaches to data provenance optimization had a positive impact on most of the measured queries, despite the difficulties with the `pg_hashtable` plugin which made it impossible to use hash-tables in phase 2. The next step that can be pursued, is fixing this plugin to properly work with phase 2 queries. This could bring an enourmous performance boost to phase 2 queries, since the switch to hash-tables will improve the time for log access from $O(n)$ to $O(1)$.

In summary, the goals of this work were achieved:

- We build a prototype that implemented automated approaches to query rewriting for optimization purposes.
- These optimizations are collected in a Haskell module providing a flexible and expandable platform for them.
- Logging optimization with hash-tables was implemented as well and measurements showed the positive impact in phase 1.

## 5.2. Future Work

There were a few topics that emerged in the context of this work but were not considered. Some of them simply were not in the scope of optimization, some others would have been too much effort to pursue in this work.

Automatic query parsing from SQL to a Haskell `SQL` AST was one of the topics not in the scope of this work. There would habe been too many cases to take into account, because of the flexibility of SQL queries, making it impossible to create a minimal working implementation in reasonable time. For a use in application development and debugging, this step must be considered.

Another step in the pipeline to make data provenance practically useful, is automatic query normalization, since our current approach only works for fully normalized queries. This, too, was not in the scope of this work.

A complete pipeline for use of data provenance in application development and debugging could look like figure 5.1. Let's take a closer look at this figure: A developer has a buggy query $q_b$. This query $q_b$ gets automatically normalized to $q_n$. This query $q_n$ gets parsed to a Haskell `SQL` AST $q_h$.

Now the translation to the phase 1 and 2 queries takes place, resulting in queries $q^1$ (an `SQL` AST) and $q^2$ (a `SetLang` AST). Phase 2 gets rewritten one more time to $q_o^2$, using the optimization strategies explained in chapter 3.4. This `SetLang` AST of $q_o^2$ can now be translated back to an `SQL` AST, and both queries $q^1$ and $q_o^2$ can be pretty printed to processable SQL.

The phase 1 query $Q^1$ gets executed, writing logs the optimized phase 2 query $Q_o^2$ can read. The results of this data provenance analysis can now be displayed on screen.

In the translation rules in chapter 3.2, there are a few corner cases that were not implemented that could result in further performance boosts:

When there is only one table $t$ in `FROM t AS v` with a predicate of `WHERE True`, then there is no need for logging. Instead simply return the `tuid` of every row in the table $(v.\rho)$. If the `WHERE` clause is not trivial, there has to be logging in any case.

A similar corner case was not considered in `Group` (see figure 3.11): If there are no grouping criteria and a trivial `HAVING` predicate, then there is no need for logging. Since logging is a great performance bottleneck, saving a logging location can boost performance significantly.

The paper [MDG18] explored the possibility to boost phase 2 queries by using another PostgreSQL plugin, implementing *roaring bitmaps* that are a better representation for sets than arrays. This approach resulted in a performance boost, too. Combining these results with our results from query rewriting could further speed up phase 2.

Another thought that was not further pursued, is the best location for why-

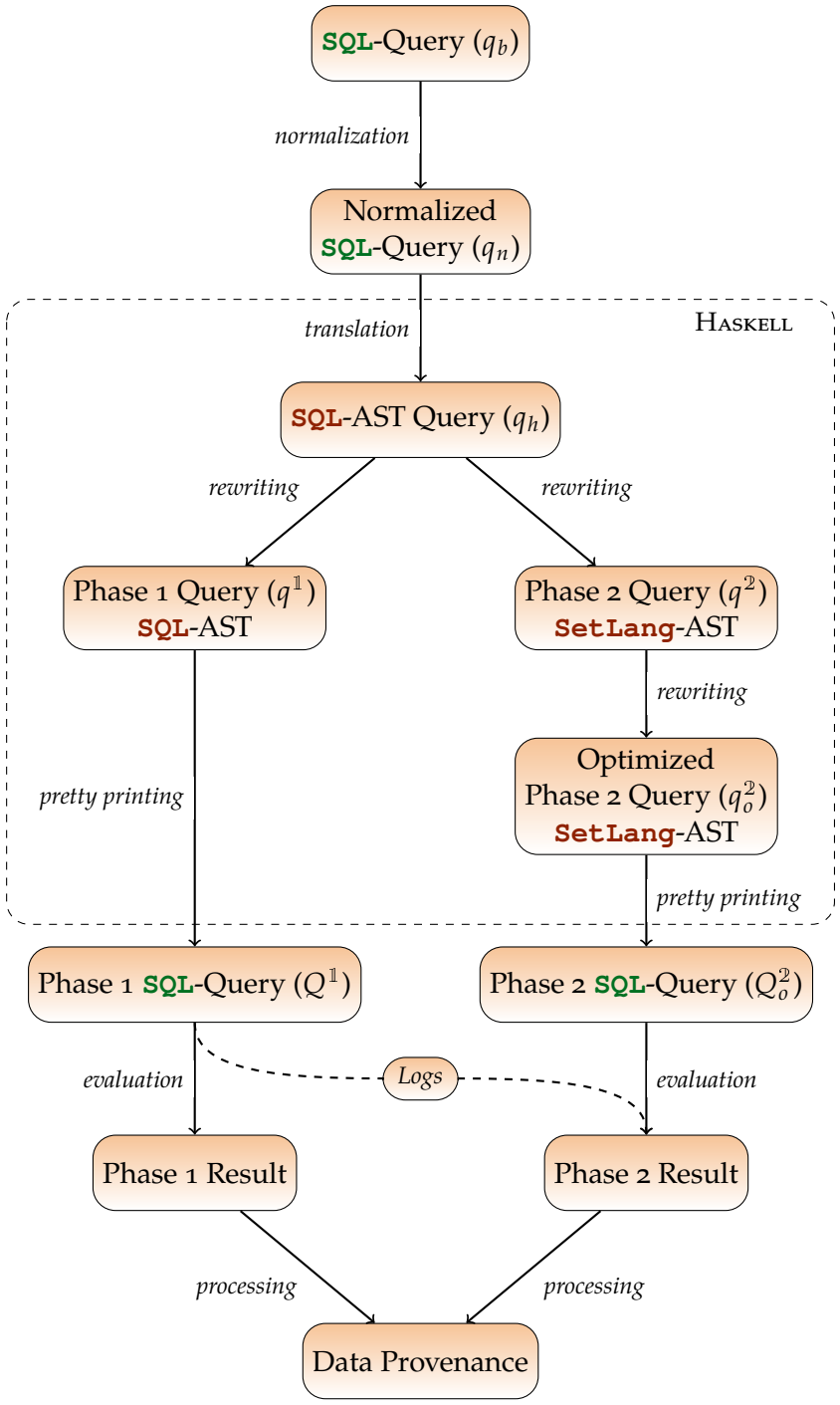



**Figure 5.1.:** Complete pipeline for usage of data provenance in application development

provance: In the approach of the Research Group, why-provenance is computed in the **FROM** expression and displayed in **SELECT**. Especially with aggregation, this is a problem, since aggregation is not compatible with **FROM** expressions (see the GROUP rule in chapter 3.2). So in the **Group** rule (see figure 3.11), there has to be another nesting in phase 2. This could be omitted, if why-provenance was computed in **SELECT**.

In the present approach, this would lead to a great overhead in computing the result, because why-provenance is added to every column in the result. The duplicated expression would have to be evaluated for every single column, although the why-provenance set is the same for every column, since it only depends on predicates working on rows of data.

Extracting why-provenance to a separate column why could bring more clarity to the result, separating where- and why-provenance. The current distinction with positive and negative tuids (see figure 2.13) could then be dropped.

A last thought that was not further pursued, is the design of duplicate elimination in the result set. The current approach is to eliminate duplicates only in the outermost query and work in the nested subqueries with these duplicates.

Another idea is to accept the overhead of duplicate elimination in every step to reduce the size of the interim results the database has to further process. This could bring another performance boost.

# Bibliography


[Bur21] Tobias Burghardt. From Recursion To Iteration: Compiling SQL UDFs with Continuations. Master's thesis, Universität Tübingen, 2021.

[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. ACM, 1974.

[GH18] Bernhard Gärtner and Martin R.W. Hiebl. *The Routledge Companion to Accounting Information Systems*. 2018.

[MDG18] Tobias Müller, Benjamin Dietrich, and Torsten Grust. You Say 'What', I Hear 'Where' and 'Why'? (Mis-)Interpreting SQL to Derive Fine-Grained Provenance. In *Proceedings of the 44th Int'l Conference on Very Large Databases. Rio de Janeiro, Brazil*, volume 11, pages 1536–1549, 2018.

[ME22] Tobias Müller and Pascal Engel. How, Where, and Why Data Provenance Improves Query Debugging. In *Proceedings of the 38th IEEE Int'l Conference on Data Engineering, Kuala Lumpur, Malaysia. To be published*, 2022.

[Mül20] Tobias Müller. *Detached Provenance Analysis*. PhD thesis, University of Tübingen, Germany, 2020.

[OMG18] Daniel O'Grady, Tobias Müller, and Torsten Grust. How "How" Explains What "What" Computes - How-Provenance for SQL and Query Compilers. In Melanie Herschel, editor, *10th USENIX Workshop on Theory and Practise of Provenance (TaPP 2018), London, UK*. USENIX Association, 2018.

[ppr] Text.PrettyPrint.ANSI.Leijen. https://hackage.haskell.org/package/ansi-wl-pprint-0.6.9/docs/Text-PrettyPrint-ANSI-Leijen.html.

[psq] PostgreSQL 13.5 Documentation. https://www.postgresql.org/docs/13/index.html.

[PW18] Fotis Psallidas and Eugene Wu. Smoke: Fine-grained lineage at interactive speed. *Proc. VLDB Endow.*, 11(6):719–732, 2018.

[sql] MySQL What is DDL, DML and DCL? https://www.w3schools.in/mysql/ddl-dml-dcl/.

[tpc] TPC-H Version 2 and Version 3. http://www.tpc.org/tpch/.

# List of Figures


1.1. Approach to data provenance of the Research Group . . . . . 1
1.2. Optimization locations in the approach from figure 1.1 . . . . . 2

2.1. Table `planets` . . . . . 5
2.2. General SQL query . . . . . 6
2.3. "What is the total mass of all non-gaseous planets?" . . . . . 7
2.4. Intuitive where- and why-provenance for query Q1 (figure 2.3a) . . . 7
2.5. Automatically generated phase 1 and phase 2 tables . . . . . 8
2.6. Add normalization to the approach . . . . . 10
2.7. A generic normalized query . . . . . 10
2.8. Possible query forms after normalization . . . . . 11
2.9. Normalized version of Q1 . . . . . 11
2.10. Query execution in phases 1 and 2 . . . . . 12
2.11. Query Q1 and its corresponding phase 1 and phase 2 queries . . . . . 13
2.12. Logs generated by Q1[1] . . . . . 14
2.13. Output of query Q1[2] using the logs from figure 2.12 and the tuids from figure 2.5b . . . . . 14
2.14. Alternative of query Q1 from figure 2.3a . . . . . 15
2.15. excerpt from automatically generated phase 2 query for TPC-H's Q16 (see chapter 4.1) . . . . . 16
2.16. excerpt from optimized phase 2 query for TPC-H's Q16 . . . . . 17
2.17. Example of an uncorrelated subquery, it does not need information from `p1` . . . . . 17
2.18. `toY` optimization of a phase 2 query . . . . . 18

3.1. Scope of the Haskell implementation . . . . . 20
3.2. A new SQL query . . . . . 20
3.3. Needed constructors for Q2 from figure 3.2a . . . . . 21
3.4. The subset of **`SL`** corresponding to figure 3.3 . . . . . 23
3.5. Inference rules: When all premises hold, the conclusion also holds . 24
3.6. Translation of **`BinOp`** in phases 1 and 2 . . . . . 25
3.7. Translation of **`ToCell`** helper . . . . . 26
3.8. Translation of **`IN`** expression . . . . . 26
3.9. Translation of **`EXISTS`** expressions . . . . . 26
3.10. Translation of **`Join`** . . . . . 27
3.11. Translation of **`Group`** . . . . . 28
3.12. Translation of aggregates . . . . . 29
3.13. Translation of **`Order`** . . . . . 30
3.14. **`CASE`** expression with independently evaluated branches . . . . . 31

3.15. **CASE** expression with one statement which is checked against multiple possible outcomes . . . 31
3.16. Translation of **Case** . . . 32
3.17. Implementation of `log1` as example for logging with tables . . . 33
3.18. Optimization locations . . . 35
3.19. Set union optimization . . . 36
3.20. Implementation of `log1` as example for logging with hash-tables . . . 38

4.1. example of phase 1 table creation (`region_1`) from table `region` . . . 40
4.2. example of phase 2 table creation (`region_2`) from table `region_1`, duplicating table `regions` columns . . . 41
4.3. Time in milliseconds in phase 2 with (■) and without (■) query rewriting, ordered by the relative boost which they get from the optimization (see figure 4.4) . . . 42
4.4. Relative impact of query rewriting in phase 2 (order criterion for figure 4.3) . . . 43
4.5. Time in milliseconds in phase 1 with (■) and without (■) hash-tables, ordered by the relative boost which they get from the optimization (see figure 4.6) . . . 44
4.6. Relative impact of switch to hash-tables in phase 1 (order criterion for figure 4.5) . . . 45
4.7. Performance impact from high boost (↑) to high penalty (↓) . . . 46
4.8. Added results from logging with hash-tables in phase 1 and query rewriting in phase 2. Unoptimized (■), optimized (■) . . . 46

5.1. Complete pipeline for usage of data provenance in application development . . . 49

# APPENDIX

## A. SQL Dialect

```
data SQL = Join [SQL]      -- SELECT-list
                [SQL]      -- FROM-list
                SQL        -- WHERE-predicate
  | Group [SQL]            -- Select (every Array entry has to use AGG)
          [SQL]            -- Join query as FROM
          [SQL]            -- GROUP BY
          SQL              -- HAVING
  | Order [SQL]            -- Select
          [SQL]            -- From
          [(SQL, Id)]      -- Order
          (Maybe Int)      -- Limit
  | Agg Id SQL
  | BinOp String SQL SQL   -- scalar binop
  | Var Id                 -- reference to tuple variable
  | ColRef SQL Id          -- column reference:
                           --   _arg1_ evaluates to a row at runtime
                           --   _arg2_ is a column name
  | TableRef Id            -- table reference:
                           --   id evaluates to a table at runtime
  | Array [SQL]            -- array with 0 to n elements
  | Concat SQL SQL         -- array concatenation
                           --   operands must both (!) evaluate to arrays
  | LitS String            -- String Literal
  | LitI Int               -- Integer Literal
  | LitF Float             -- Integer Literal
  | LitB Bool              -- Bool Literal
  | Date String            -- converts String to Date
  | Interval Int String    -- e.g. 5 'months'
  | Extract String SQL     -- extracts part of date
  | Bind SQL Id            -- ... AS ...
  | And [SQL]              -- logical conjunction
  | Or [SQL]               -- locical disjunction
  | Udf Id                 -- User defined function, id: name
        [SQL]              -- list of arguments
  | ToCell SQL Id          -- helper for subqueries
  | Case (Maybe SQL)       -- CASE ...
         [(SQL, SQL)]      -- WHEN ... THEN ...
         SQL               -- ELSE
```

```
  | In SQL SQL            -- is element in list?
  | Exists SQL           -- exists an element in list?
  | NotExists SQL
  | Lateral SQL          -- LATERAL keyword
  | Substring12 SQL      -- first two characters in a string
deriving (Show)
```

# B. SetLang Dialect

```
data SL = Join [SL] [SL] SL  -- SFW expr. with S only
        | Group [SL] [SL] [SL] SL
        | Agg SL
        | Var Id        -- tuple variable
        | ColRef SL Id -- column reference
        | TableRef Id -- table reference
        -- set handling
        | Union [SL]    -- union of _n_ flat set expressions
        | Empty         -- the empty set
        | LitB Bool
        | Extract String SL
        | Bind SL Id
        | Udf Id [SL]
        | Case SL [(SL,SL)] SL
        | Lateral SL
        deriving (Show, Eq)
```

# Acknowledgements

Let's start with my great supervisor, Dr. Tobias Müller: I'm extremely grateful for his valuable support, advise, and the speed in which he provided both. He always gave me exactly the food for thought I needed whenever I encountered a problem.

I would like to extend my gratitude to Denis Hirn who provided me with the `pg_hashtable` plugin and the LaTeX-template for this thesis. I am also grateful to Prof. Dr. Torsten Grust and the Research Group for their work. It was a pleasure working with their approach, as it is very well explained and documented in their various papers.

I also very much appreciate the help of my sister Leonie Täsch who has proofread this entire work without understanding a word of the technical jargon. Lastly, I want to thank my partner Viktoria Edenharter for her patience when I explained the same details of this work a thousand times from different angles.

# Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe. Alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, sind durch Angaben von Quellen als Entlehnung kenntlich gemacht worden. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt. Auch wurde die Arbeit weder vollständig noch in Teilen bereits veröffentlicht. Das in Dateiform eingereichte Exemplar stimmt mit den eingereichten gebundenen Exemplaren vollständig überein.

| | |
|---|---|
| Ort, Datum | Unterschrift<br>*Daniel Täsch* |