Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelorarbeit Informatik

How-Provenance Through Query Rewriting

Pascal Engel

August 14, 2020

Gutachter
Prof. Dr. Torsten Grust
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Betreuer
Dr. Tobias Müller
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Pascal Engel

How-Provenance Through Query Rewriting
Bachelorarbeit Informatik
Eberhard Karls Universität Tübingen
Matrikelnummer: 4118448
Bearbeitungszeit: July 2020 – November 2020

Abstract

For SQL queries in databases, how–provenance can add valuable insight into which expressions of the query are responsible for which cell values from the result. In this work, a two–phase approach developed by the Database Research Group of the University of Tübingen is used to translate a given query into two distinct queries that compute its how–provenance. This work presents a Haskell program to automate this rewriting process for a multitude of SQL constructs.

Contents

List of Figures

1 | Introduction

With the ever growing amount of data provided today through the internet, it is increasingly important in databases to ensure their trustworthiness and integrity. Here, provenance may help to add value to data by showing its source. [CCT09] Specifically, the notion of *how*–provenance may help debugging a query and finding flaws that lead to unexpected behaviour. Moreover, it is useful to simply understand what query expressions do and how they contribute to a specific result value.

Listing 1.1 shows a *SQL* query that selects the destination of all excessively cheap and expensive inner-European flights from a database (Table 1.1). The query is composed of two subqueries that are being unified. Usually, when running the query we will only be provided with the result table. What *how*–provenance offers here is insight into which part of the query is responsible for which result value. For the sake of simplicity in this example, we are only concerned with subqueries. The first subquery colored in orange selects those destinations of flights with a low price. Analogously, the result values that went through the second subquery with a high price are colored green.

```
(SELECT destination
 FROM flights
 WHERE price < 100)
UNION ALL
(SELECT destination
 FROM flights
 WHERE price > 200);
```

Listing 1.1: Example query

In Table 1.1 the corresponding values, that are being calculated by it, are colored accordingly. We can see here, exactly which subquery is responsible for which resulting cell value, which is more than *SQL* can provide by itself.

flights

origin	destination	distance	price
Porto	Amsterdam	1612	212
Amsterdam	Munich	668	164
London	Amsterdam	357	92
London	Rome	1435	283
Munich	Rome	698	194

output

destination
Amsterdam
Amsterdam
Rome

Table 1.1: Input and output table

The example above is only concerned with subquery expressions, but it shows how how–provenance adds insight into the computations of a query, which a Database Management System cannot provide. Using this notion for a variety of additional *SQL* expressions, we can compute more fine-grained provenance, as well.

The goal of this work is to implement a SQL-to-SQL compiler that automates the computation of How-Provenance for any given *SQL* query. For this, the Database Research Group from the University of Tübingen has developed a two-phase approach [OMG18] which lays the theoretical groundwork for this thesis.
In the following chapters, I will begin by explaining Provenance in the context of Databases and the query language *SQL*. Then, the theoretical groundwork, including the main prerequisites and the formalization of the implemented translation rules are presented. Finally, in the main part, the implementation in Haskell is presented and discussed, along with possible further work based on the program.

2 | Provenance in Databases

In this chapter, I will outline the Provenance aimed to compute in this work in the context of databases and the associated query language *SQL*.

2.1 SQL

The *Structural Query Language* (*SQL*) has been the dominant language for managing data in databases. It is built on top of the theoretical works on relational algebra. Its syntax is supposed to be intuitive in plain English, so a *SQL* query can be roughly interpreted as an English sentence, making it easily understandable even for people who are not familiar with databases. [CB74]

Besides queries, *SQL* offers various functionalities to create and manipulate tables in a database, as well as guaranteeing their integrity. With the recent addition of recursion it is now even seen as a very capable Turing-complete programming language.

2.1.1 Queries

In *SQL* queries are the essential tool to answer questions about the data present in a given database. The standard form of a query is a *Select From Where* (*SFW*) expression. In this work a limited *SQL* dialect is used, so not all common *SQL* expressions are covered. *SFW* expressions are limited to the following form:

SELECT e_1 , ...
FROM et_1 , ...
WHERE w
GROUP BY g_1 , ...
ORDER BY o_1
DISTINCT ON d_1 , ...
LIMIT ℓ_l
OFFSET ℓ_o

In such a query all clauses are optional except the **SELECT** clause which is responsible for returning a result value and thus cannot be left out.

2.1.2 User Defined Functions

Generally, *User Defined Functions* (*UDF*s) can be used to compute a result value or manipulate a database. In the scope of this work, they are used without any side–effects. The syntax of these *UDF*s is briefly explained in Listing 2.1, where each τ_i stands for a type and q represents a *SQL* query expression. *UDF*s are more powerful than this template suggests, however in the scope of this work *UDF* definitions of this form are sufficient. Note that here the function arguments are unnamed, so in the function body they will appear in the order they are bound as `$1`,`$2`,....

```
CREATE [OR REPLACE] FUNCTION f(τ_1, ..., τ_n) RETURNS τ AS
$$
   q
$$ LANGUAGE SQL;
```

Listing 2.1: Syntax for UDF definitions

2.1.3 PostgreSQL

PostgreSQL (or simply Postgres) is an open–source relational database management system (DBMS) first introduced in 1996. [pos] Although the query–rewriting is designed independent of the DBMS, Postgres offers some useful functionalities. It generally supports all common *SQL* constructs together with some additions thanks to its interface for extensions. For these reasons Postgres is the chosen DBMS at the Database Research Group. Here, it is used to run the generated queries from the compiler presented in this work. Additionally, its internal parser is used to parse a query into an Abstract Syntax Tree.

2.2 Data Provenance

In general, 'provenance information describes the origin and history of data in its life cycle.' [CCT09, p.1] Provenance can be applied to all kinds of data, wherever it is useful to know how it is created. 'In databases it aims to answer record-level questions, e.g., which tuples (rows) in the input tables contributed to a particular output tuple and how.' [Lud03, p.10] These questions further divide provenance analysis into subcategories concerned with *why* a specific result is being computed, *where* the resulting data originally comes from and *how* it has been computed by the query.

2.2.1 How–Provenance

In this work, I focus on the *how*-aspect. The notion of *how*-provenance has first been introduced by *Green et al.* to add insights to data provenance where *why*-provenance could not answer questions sufficiently. [GKT07] *How*-provenance helps tracing output items back through the query and can explain complex queries that build on advanced *SQL* constructs. [OMG18] So analyzing the provenance of a result value will point us at exactly those query constructs that are involved in its computation. It is therefore particularly helpful when debugging queries and generally handy while learning the *SQL* language to gain further insights that a simple result table cannot offer.

2.2.2 Phases of Provenance

This work relies on a two-phase approach to translate an input query into two distinct queries, that compute its how-provenance. Using this query-rewriting, a database management system can run the generated queries and return the desired provenance data. This is useful because no additional software is required in this

step. A DBMS is already capable of running queries efficiently, so using it for this purpose saves a lot of work.
As shown in Fig. 2.1, in this process each expression of an input query is annotated with a label and translated independently in two phases: In Phase𝟙 the query is being evaluated and as a side–effect corresponding data is logged in a dedicated table. In Phase𝟚 literal data is ignored, instead the query-annotations are *pushed* into the query result while the logging data from Phase𝟙 is being read to ensure correctness. The resulting relations of both phases are tables that are isomorphic to each other (i.e. they consist of the same columns and the same number of rows). This means that for every cell in the result of Phase𝟙, that contains the data of the original query, there is a corresponding cell in the result of Phase𝟚, that contains the collected provenance annotations. Thus, by comparing the annotated query with the resulting relations, we can deduce exactly which part of the query is involved in which resulting cell value.

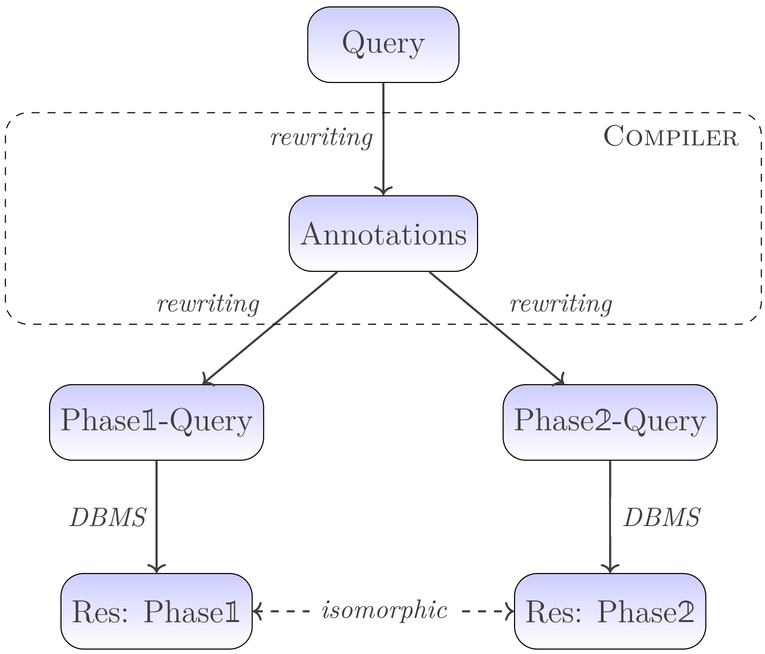

Figure 2.1: Computation of How-Provance using 2 phases

The compiler developed for this thesis aims to automate the rewriting process for both phases. As demonstrated in Fig. 2.1 the compiler covers all rewriting processes: For an input query, an annotated query, a Phase𝟙 query and a Phase𝟚 query are generated. The queries of both phases can then be executed by a DBMS and the resulting tables are isomorphic to each other, so for each cell a provenance set is calculated.

2.3 Example

To explain this with a more in-depth example, let's take another look at the query in Listing 1.1. For this rewriting process, the query is first annotated, so each expression has an unique label between 1 and 13 (Listing 2.2).

```
^2(SELECT ^3destination
   FROM ^4flights
   WHERE ^5price ^7< ^6100)
^1UNION ALL
^8(SELECT ^9destination
   FROM ^10flights
   WHERE ^11price ^13> ^12200);
```

Listing 2.2: Annotated query

This query is then transformed based on rules for each expression. The Phase1 translation produces a table containing tuple identifiers and original cell values. For Phase2 the result contains the same tuple identifiers, but the cell values are replaced with all provenance annotations that were involved in the computation for that cell. These annotations are collected in an array for each cell.

result1

tuid	destination
3	Amsterdam
1	Amsterdam
5	Rome

result2

tuid	destination
3	$\{1, 2, 3, 4, 5, 6, 7\}$
1	$\{1, 8, 9, 10, 11, 12, 13\}$
5	$\{1, 8, 9, 10, 11, 12, 13\}$

Table 2.1: Results of generated queries

We can interpret these two result tables together with the annotated query: For each cell, by inspecting the collected provenance annotations and comparing them to the labels in the annotated query, we can deduce exactly which expressions are involved in its computation. For example, for the cell Rome in Phase1, we can compare its tuid value with the ones in Phase2 and thus associate the provenance set $\{1, 8, 9, 10, 11, 12, 13\}$ with it. By comparing the elements in the provenance set with the labels in the annotated query, we can deduce that the **UNION ALL** expression and all expressions contained in the second subquery are involved in its computation.

Additionally, for both Amsterdam cells in Phase1 we gain the same insight shown in the introduction: The cell with the tuple identifier 3 was computed by the first subquery and the one with 1 by the second subquery.

3 | Formalization

In this chapter, the theoretical background for the compiler presented in this work is outlined.

3.1 Notation

This section provides an overview of the notations used in the formalization.

Meta Variables These meta variables represent different kinds of *SQL* expressions.

Symbol	Description
e	Meta variable for an arbitrary expression
ℓ	Meta variable for a literal expression
ec	Meta variable for a cell expression
et	Meta variable for a table expression

Provenance Data The data describing provenance is notated as sets. Annotations form the elements of these sets.

Symbol	Description
α	Meta variable for an annotation (represented as integer)
e^{α}	Annotated expression
$\mathbb{p}$	Meta variable for a provenance set
$\{\ldots\}$	Set of provenance
$\cup$	Binary set union
$\Psi(e, \mathbb{p})$	Add data provenance $\mathbb{p}$ to each provenance set found in e's result value

Logging Various logging functions are used to keep track of data (e.g. while filtering).

Symbol	Description
f	Meta variable for a free tuple variable
$\{f_1, \ldots\} \coloneqq \mathsf{fv}(e)$	Function to find free tuple variables in an expression e
$\textcircled{\ell}$	Text location identifier (to keep track of logging call)
$\mathsf{cast}(\alpha)$	Convert an annotation to a location identifier
`writeX`	*UDF* to log data in Phase$\mathbb{1}$ in the context of `X`
`readX`	*UDF* to read logs in Phase$\mathbb{2}$ in the context of `X`

3.2 Prerequisites

In order for the generated *SQL* code to run properly, a few conditions have to hold, before the query rewriting can begin. These are shown in this section.

3.2.1 Provenance Relations

For each table referenced in the query, there have to be two relations concerned with its provenance. These relations replace the original relations referenced in the input query. In this work, they are defined by *SQL* Views. As shown in Fig. 3.1, for the `flights` table in the introduction, the Phase$\mathbb{1}$ relation contains the same data with an additional `tuid` column. The Phase$\mathbb{2}$ relation is made of the same columns and contains the same values as tuid, however each original data cell is replaced with an empty provenance set. These provenance sets are finally used to accumulate all relevant annotations for the result.

flights

origin	destination	distance	price
Porto	Amsterdam	1612	212
Amsterdam	Munich	668	164
London	Amsterdam	357	92
London	Rome	1435	283
Munich	Rome	698	194

flights𝟙

tuid	origin	destination	distance	price
1	Porto	Amsterdam	1612	212
2	Amsterdam	Munich	668	164
3	London	Amsterdam	357	92
4	London	Rome	1435	283
5	Munich	Rome	698	194

flights𝟚

tuid	origin	destination	distance	price
1	{}	{}	{}	{}
2	{}	{}	{}	{}
3	{}	{}	{}	{}
4	{}	{}	{}	{}
5	{}	{}	{}	{}

Figure 3.1: Phase𝟙/𝟚 relations for table flights

3.2.2 Normalized Queries

SFW-blocks have to be normalized so each SFW-block falls under just one of four categories. SFW-blocks can be a plain select, n-way joins, aggregates and order-by. Queries, that don't hold this, need to be transformed to separate the responsibilities to subqueries. This clause isolation imposes clear advantages here: Instead of defining rules for all possible ways of defining a SFW-block, we only use rules for specific cases of blocks and limit the total number of translation rules and thus achieve a clear separation of responsibilities. Furthermore, this adds expressiveness to the query constructs which may help while debugging.

As described in *Müller et al.*, a given query can be normalized to a query containing multiple nested subqueries with distinct cases. [MDG18, p.1541]

```
                                         SELECT e1 , ...
                                           FROM (
                                                 SELECT e1 , ...
                                                   FROM (
   SELECT e1 , ...                                       SELECT e1 , ...
     FROM et1 , ...                                        FROM et1 , ...
    WHERE w                                               WHERE w
 GROUP BY g1 , ...     ⟹norm                              )
 ORDER BY o1                                    GROUP BY g1 , ...
DISTINCT ON d1 , ...                             )
    LIMIT ℓl                             ORDER BY o1 , ...
   OFFSET ℓo                          DISTINCT ON d1 , ...
                                            LIMIT ℓl
                                           OFFSET ℓo
```

3.2.3 Annotations

Before rewriting can begin, each expression e needs to be annotated with a label α to e^{α} that represents the expression's provenance data.

3.2.4 Logging

For case–expressions, joining, filtering, aggregating and unifying queries, tuples need to be logged. Therefore, dedicated logging-UDFs have to be loaded into the database before running the generated queries. These UDFs write in Phase$\mathbb{1}$ relevant tuple identifiers into a logging table and read them in Phase$\mathbb{2}$, so that branches and filters still apply even when the relevant data is not present. [MDG18, p.1543]

3.3 Rules

This work includes the rewriting of most *SFW* expressions, basic operations like function applications, column references, **ROW** and **VALUES** constructors, **CASE** expressions, **WITH** bindings and **UNION ALL**. The following presents a selection of the provided translation rules in order to exemplary explain the rewriting-process for both phases. A complete list of the translation rules is added in the Appendix. The section above the bar defines the environment under which the rule is applied. This also includes prerequisites for the rule, recursively already translated subexpressions and other bindings for various terms. Under the bar, on the left hand side is a generalized annotated expression as input and the rewritten expression on the right hand side as output expression. A translation rule is denoted as

$$SQL^{\alpha} \Rrightarrow^{\tau} (SQL^{\mathbb{1}}, SQL^{\mathbb{2}})$$

where SQL^{α} is an annotated SQL expression, τ is the type of an SQL expression and $(SQL^{\mathbb{1}}, SQL^{\mathbb{2}})$ is the tuple consisting of the translated Phase$\mathbb{1}$ and Phase$\mathbb{2}$ SQL expressions.
The overall rewriting process is the composition of these rules.

3.3.1 Operators

This rule is concerned with binary operators on a cell level. These can be arithmetic (+,*,...) or logic (&,...). In Phase$\mathbb{1}$ the generalized operator $\circledast$ is simply applied to its arguments $ec_1^{\mathbb{1}}, ec_2^{\mathbb{1}}$. In Phase$\mathbb{2}$, because the subexpressions $ec_1^{\mathbb{2}}, ec_2^{\mathbb{2}}$ are expected to be provenance sets, they are unified to a singular set, where the label of the operator expression is added to. The provenance of the operator is computed by replacing $\circledast$ with $\cup\{\alpha\}\cup$, so instead of the operator its label is added and the resulting how–provenance consists of the provenance of the subexpressions and the label of the operator expression.

BINOP SCALAR

$$\frac{ec_1^{\alpha_1} \Rrightarrow^{cell} (ec_1^{\mathbb{1}}, ec_1^{\mathbb{2}}) \qquad ec_2^{\alpha_2} \Rrightarrow^{cell} (ec_2^{\mathbb{1}}, ec_2^{\mathbb{2}})}{(ec_1^{\alpha_1} \circledast ec_2^{\alpha_2})^{\alpha} \Rrightarrow^{cell} (ec_1^{\mathbb{1}} \circledast ec_2^{\mathbb{1}}, ec_1^{\mathbb{2}} \cup \{\alpha\} \cup ec_2^{\mathbb{2}})}$$

3.3.2 Case

This rule is concerned with the *SQL* **CASE**-construct. The idea for this rule is to label each branch and memorize the one which is taken in Phase𝟙 and read that information in Phase𝟚 to ensure the same branch is being taken, while collecting all provenance for expressions that need to be evaluated for the branch-decision.

In detail, in Phase𝟙 all **WHEN**-expressions are evaluated, until one evaluates to true. Then, the corresponding integer value will be passed to the function `writeCase` in order to log the branch to be taken. After that, the corresponding **THEN**-expression will be evaluated.

In Phase𝟚 the logged integer to determine the branch is loaded with the `readCase` function. Accordingly, the corresponding provenance of the **THEN**-expression is returned, alongside the provenance of *all* **WHEN**-expressions looked at until this point. The provenance of all previous **WHEN**-expressions has to be considered, because all of them have to be evaluated and rejected until the correct branch is being taken. This is why, if the **ELSE**-branch is taken, all **WHEN**-expressions are in the result. Finally, the label α of the **CASE**-expression is pushed into the result.[1]

[1] Note that free Variables are not considered here, however the rule presented in the Appendix covers free variables.

$$
\dfrac{
\begin{array}{l}
\text{CASE} \\
\quad \Big| \; ec_{w_i}^{\alpha_{w_i}} \Rrightarrow^{cell} (ec_{w_i}^{\mathbb{1}}, ec_{w_i}^{\mathbb{2}}) \Big|_{i=1\ldots} \quad \Big| \; ec_{t_i}^{\alpha_{t_i}} \Rrightarrow^{cell} (ec_{t_i}^{\mathbb{1}}, ec_{t_i}^{\mathbb{2}}) \Big|_{i=0\ldots} \\
\qquad\qquad\qquad \textcircled{l} := \mathsf{tolocation}(\alpha) \\
X^{\mathbb{1}} := \begin{array}{l}
\textbf{CASE}\ \texttt{writeCase}\ (\textcircled{l}, \\
\quad \textbf{CASE} \\
\qquad \textbf{WHEN}\ ec_{w_1}^{\mathbb{1}}\ \textbf{THEN}\ 1 \\
\qquad \vdots \\
\qquad \textbf{ELSE}\ 0 \\
\quad \textbf{END} \\
) \\
\textbf{WHEN}\ 1\ \textbf{THEN}\ ec_{t_1}^{\mathbb{1}} \\
\vdots \\
\textbf{ELSE}\ ec_{t_0}^{\mathbb{1}} \\
\textbf{END}
\end{array} \\
X^{\mathbb{2}} := \begin{array}{l}
\textbf{CASE}\ \texttt{readCase}\ (\textcircled{l}) \\
\quad \textbf{WHEN}\ 1\ \textbf{THEN}\ \Psi(ec_{t_1}^{\mathbb{2}}, ec_{w_1}^{\mathbb{2}}) \\
\quad \vdots \\
\quad \textbf{WHEN}\ i\ \textbf{THEN}\ \Psi(ec_{t_i}^{\mathbb{2}}, ec_{w_1}^{\mathbb{2}} \cup \ldots \cup ec_{w_i}^{\mathbb{2}}) \\
\quad \vdots \\
\quad \textbf{ELSE}\ \Psi(ec_{t_0}^{\mathbb{2}}, ec_{w_1}^{\mathbb{2}} \cup \ldots) \\
\textbf{END}
\end{array}
\end{array}
}{
e_{in} := \left(\begin{array}{l}
\textbf{CASE} \\
\quad \textbf{WHEN}\ ec_{w_1}^{\alpha_{w1}}\ \textbf{THEN}\ ec_{t_1}^{\alpha_{t1}} \\
\quad \vdots \\
\quad \textbf{ELSE}\ ec_{t_0}^{\alpha_{t0}} \\
\textbf{END}
\end{array} \right)^{\alpha} \Rrightarrow^{cell} (X^{\mathbb{1}}, \Psi(X^{\mathbb{2}}, \{\alpha\}))
}
$$

3.3.3 SFW: n–way join

The n–way Join is a centerpiece in relational algebra for combining multiple relations and filtering results. Here, its generalized pattern is given with et_{inp}. The **FROM** section may yield an arbitrary number of relations, the **WHERE**-expression is optional (will be interpreted as *true* if missing). No additional clauses are permitted in the *SFW* block.

In Phase𝟙 the `writeJoin` *UDF* logs those tuple identifiers, that have passed the filtering and joining process. Exactly those are also returned in the **SELECT**-clause. As Phase𝟚 cannot handle literal data the information of the Phase𝟙-logging has to be read as a table *log* that is joined with the original relations to only include rows with a tuple identifier, which has been previously logged. This way, the joining and filtering process from Phase𝟙 is reconstructed. Additionally, because here we cannot make use of the **WHERE**-expression, its provenance is pushed into each column reference in the **SELECT**-section, so it will be present in the final provenance set. At last, the label α is *pushed* to the final result of the expression.

SFW-Join

$$\frac{\begin{array}{c} ⓒ := \mathsf{cast}(\alpha) \quad \{f_1, \ldots\} := \mathsf{fv}(et_{inp}) \\ \left| \; ec_i^{\alpha_i} \Rrightarrow^{cell} (ec_i^{\mathbb{1}}, ec_i^{\mathbb{2}}) \; \right|_{i=0,..n} \\ \left| \; et_i^{\alpha_i} \Rrightarrow^{table} (et_i^{\mathbb{1}}, et_i^{\mathbb{2}}) \; \right|_{i=(n+1),..(n+m)} \\ X^{\mathbb{1}} := \begin{array}{rl} \texttt{SELECT} & \texttt{writeJoin}\,(ⓒ, var_1.\rho, \ldots, f_1.\rho, \ldots)\ \texttt{AS}\ \rho \\ & ec_1^{\mathbb{1}}\ \texttt{AS}\ col_1, \ldots \\ \texttt{FROM} & et_1^{\mathbb{1}}\ \texttt{AS}\ var_1, \ldots \\ \texttt{WHERE} & ec_0^{\mathbb{1}} \end{array} \\ X^{\mathbb{2}} := \begin{array}{rl} \texttt{SELECT} & log.\rho\ \texttt{AS}\ \rho \\ & \Psi(ec_1^{\mathbb{2}}, ec_0^{\mathbb{2}})\ \texttt{AS}\ col_1, \ldots \\ \texttt{FROM} & et_1^{\mathbb{2}}\ \texttt{AS}\ var_1, \ldots, \\ & \texttt{readJoin}\,(ⓒ, var_1.\rho, \ldots, f_1.\rho, \ldots)\ \texttt{AS}\ log \end{array} \end{array}}{et_{inp} := \left(\begin{array}{rl} \texttt{SELECT} & ec_1^{\alpha_1}\ \texttt{AS}\ col_1, \ldots \\ \texttt{FROM} & et_{n+1}^{\alpha_{n+1}}\ \texttt{AS}\ var_{n+1}, \ldots \\ \texttt{WHERE} & ec_0^{\alpha_0} \end{array} \right)^{\alpha} \Rrightarrow^{table} (X^{\mathbb{1}}, \Psi(X^{\mathbb{2}}, \{\alpha\}))}$$

4 | Implementation

In this chapter, the main part of this work is discussed: the implementation of the compiler based on the mostly given formalization in Chapter 3. I will begin by presenting the tools and structures used for the program and then explain the project structure and each module in detail.

4.1 Haskell

The chosen programming language for the implementation is Haskell. [Jon03] Haskell is a declarative, purely functional language first introduced in 1990. In the scope of this work its functional aspect is very useful for defining the translation rules presented in Section 3.3 because in a Haskell program mathematical syntax can easily be interpreted as a program. Therefore, the relation between the formal rules and the implementation in Haskell is particularly intuitive.

Furthermore, the static type system allows us to check the correctness of the program on type level while developing, which makes testing a lot easier. The type system additionally helps separate the scope of distinct parts of the program. E.g. in this project, only the Main module is capable of handling Input/Output, so the query translation which is done in different modules cannot interfere or be influenced by side effects on the machine. Here, the modularity of Haskell also helps separate responsibilities of independent computations (like Phase$\mathbb{1}$ and Phase$\mathbb{2}$).

The inherent *Laziness* and the *Algebraic Data Types* work together neatly while recursively traversing the syntax tree of a *SQL* expression. The *lazy* evaluation of subexpressions can save a decent amount of runtime here.

Finally, the usage of language extensions provides a more expressive and declarative way of coding according to specific needs. In this work, the language extensions *ViewPatterns*, *RecordWildCards* and *GADTs* have proven themselves useful.

4.1.1 Abstract Syntax Tree

Previous work on the topic has already been implemented in Haskell: Most notably the *LogParser* designed by Denis Hirn [Hir17]. The *LogParser* library uses an expansion for Postgres to extract logs from the internal Postgres *SQL*–parser and uses them to create an *Abstract Syntax Tree* (AST) representation in Haskell. The type of the AST is a *Generalized Algebraic Datatype* (GADT), which means it encapsulates multiple data types (in this case different kinds of *SQL* expressions) in one generalized type for all *SQL* expressions. So the type system of Haskell is still useful to distinguish different types of *SQL* expressions.
In the implementation presented in this work, all transformations work on top of this AST. Because typically records in the AST have a lot more fields than are relevant for our purposes, using *RecordWildCards* is a simple way to ignore those fields and pass the corresponding values through by inexplicitly binding them to variables that are not being manipulated.
Furthermore, the library exports the `transformM` function which implements an easy full traversal of an AST while applying a monadic function to all nodes. This is very useful to ensure the totality of a function defined with `transformM`.
For the output of the program, the *PrettyPrinter* library prints syntax-highlighted SQL code from the AST to the console. So it can be checked and inserted into a DBMS to compute provenance.

4.2 Project Structure

4.2.1 Modules

Fig. 4.1 describes how the structure and internal dependencies of the project presented in this work.

- The *Main* module provides a minimal user interface to load a query, parse it using the *LogParser* and print the annotated query and its Phase1- and Phase2-translation using the *PrettyPrinter*.

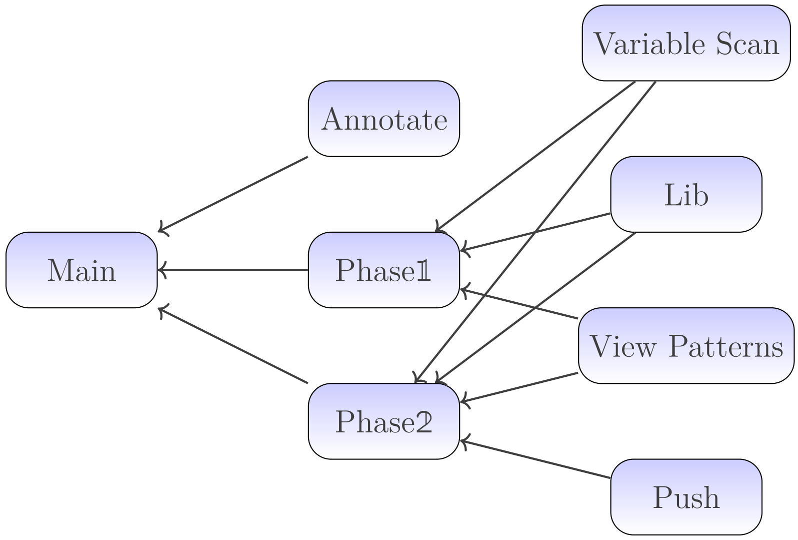

Figure 4.1: Modular dependencies

- In the *Annotate* module, the AST is traversed and to each node a globally unique label is added where uniqueness is ensured using a **`State`** monad as counter.

- In the dedicated modules Phase1 and Phase2, the rules described in Section 3.3 are implemented using plain Haskell pattern–matching as well as the Language Extension *ViewPatterns*.

- The *Lib* module includes all kinds of auxilary functions, particularly the implemantation of provenance sets, tuid–colum references and the handling of logging functions.

- The *View Patterns* strictly defines the various forms of SFW-blocks to differentiate in the translation process.

- *Push* includes all functionality associated with the Ψ–operator and thus enables pushing labels to provenance sets in Phase2.

- *VariableScan* is used for getting all free variables in a given subexpression to deal with correlated subqueries.

4.2.2 Annotate

In a preprocessing step, labels are distributed to each expression in the AST. **`transformM`** traverses the AST and each node relevant for *how*–provenance is wrapped in an additional **`GLabel`**-node. Using a state, it is ensured that every label is unique within the AST.
The labels are only used internally during the translation process. Printed out, the annotated query is not executable because the Label nodes are not part of SQL. However, it needs to be shown to the user in some way in order to interpret the resulting provenance sets.

4.2.3 Provenance Relations

As stated in Section 3.2.1, for each table in the query, a relation for both phases needs to be defined. This has already been achieved in related work using materialized views [Par18] and works the same way here: For Phase𝟙 the view consists of the original table with an additional tuid column that is filled by a sequence. The Phase𝟚 view references the view from Phase𝟙, so the tuid columns have the same values, and replaces all other cells with empty provenance sets. An example for these view definitions is included in the Appendix.

4.2.4 Correlated Subqueries

As shown in Section 3.1 the function $\mathsf{fv}(e)$ is used to get all free tuple variables in an expression e. This is done to be able to deal with correlated subqueries. A correlated subquery is a subquery that references variables that were outside of its scope, if it was executed separately. In order to deal with these kinds of subqueries, everytime a query-expression is found, it has to be checked, whether there are any free variables present. If so, then all free variables need to be considered while logging, so they need to be passed as arguments to the logging-functions, too.
Finding free variables is done by the function ***`scan`*** in the module *VariableScan*, where all definitions of variables and all references of variables are collected and compared in a state monad, so locally undefined tuple variables can be found. This step is done for every SFW-expression (compare Snippets 4.7 and 4.8), which ensures the correctness for correlated subqueries.

```
module VariableScan (scan) where

trav :: Rule (SQL a) (SQL a)

-- collect var-definition in a table reference
trav RTE {..} =
  modify (<> StateC [aliasname] [])
  *> transformM trav RTE {..}
  <* ensure

-- collect var-reference in a column reference
trav EColRef {..} =
  modify (<> StateC [] [rowVar])
  *> transformM trav EColRef {..}
  <* ensure
```

Snippet 4.1: Variable Scan

4.2.5 Provenance Sets

Provenance sets are implemented in *SQL* as arrays of integers. This is only an approximation, since arrays do not have the same properties as sets, e.g. they may include duplicate elements. In further work, the implementation type of provenance sets may be subject of change.

On these implemented provenance sets, set operations need to be executed on just like on basic Haskell datatypes, therefore the implementation may look like a `Monoid` instance declaration. However, since the `EArrayExpr` data constructor is only one of many of type `Expr`, a generalized *Monoid* instance can not be defined. Thus, the final implementation as seen in Snippet 4.2 may resemble an instance declaration with the typical functionalities associated with *Monoids*.

```
-- ARITHMETICS OF PROVENANCE SETS

-- toLiteral-function creates integer literal
toLiteral :: Integer -> Expr
toLiteral tuid = Lit (ConstI tuid) (TAtom "int4") Nothing

-- type of provenance sets
psetType :: Type
```

```
psetType = TArray (TAtom "int4")

toSet :: [Integer] -> Expr
toSet pids = EArrayExpr
                { typ       = psetType
                , multidims = False
                , elements  = map toLiteral pids
                , location  = Nothing
                }

-- empty provenance set (identity element)
pempty :: Expr
pempty  = EArrayExpr psetType False [] Nothing

-- unification of provenance sets
(\/) :: Expr -> Expr -> Expr
-- static  array append using (++)
(\/) (EArrayExpr _ _ p1 _) (EArrayExpr _ _ p2 _) =
                  EArrayExpr
                    { typ       = psetType
                    , multidims = False
                    , elements  = p1 ++ p2
                    , location  = Nothing
                    }
-- dynamic array append (using pg operator)
(\/) e1 e2 = EOpexpr { oprleft  = Just e1
                     , oprright = e2
                     , oprname  = "pg_operator.||"
                     , typ      = TFunc [] (TPseudo "anyarray")
                     , location = Nothing
                     }

-- concat for provenance sets
pconcat :: [Expr] -> Expr
pconcat = foldr (\/) pempty
```

Snippet 4.2: Provenance Sets

4.2.6 Push

For each node in the *AST*, we need to be able to add the corresponding provenance annotation to the result. This should work for all expressions, regardless of their type. E.g. in $\Psi($`ROW(42, {7}),{10}`$)$ = `ROW(42, {7,10})` the first column is the tuple identifier as expected. Because it does not contain provenance annotations, nothing can be *pushed* into it. However, all other columns contain provenance sets, so the new set `{10}` is unified with the second column.
In general, *pushing* a label into a data structure means adding it to all parts of the structure where a provenance set can be found. Formally we can define the push-operator Ψ recursively as follows

$$\Psi(v, \mathbb{p}) := \begin{cases} v \mid\mid \mathbb{p} & \text{for } v \text{ atomar} \\ \texttt{ROW}(\rho, \Psi(col_1, \mathbb{p}), \Psi(col_2, \mathbb{p}), \dots) & \text{for } v = \texttt{ROW}(\rho, col_1, col_2, \dots) \\ [\Psi(e_1, \mathbb{p}), \Psi(e_2, \mathbb{p}), \dots] & \text{for } v = [e_1, e_2, \dots] \end{cases}$$

If we assume parametric polymorphism, the type of Ψ looks as follows

$$\Psi : \mathbb{T}\ \mathbb{P} \to \mathbb{P} \to \mathbb{T}\ \mathbb{P}$$

where $\mathbb{P}$ is the (static) type of a provenance set and $\mathbb{T}$ a parameter for an arbitary *SQL* data structure (e.g. row).
However, since the argument is generally type-polymorphic, *pushing* into it can not be done with just one operator. E.g. the result of a SFW-Block will be a table (with a specific type), while the result of a function application or a column reference might be a literal.
Therefore, the implementation of the Ψ–operator is particularly challenging. The custom *UDF* definitions and calls for this implementation are presented in the following sections.

4.2.6.1 UDF Definitions

Because *SQL* does not support parametric polymorphism though, there can not be one singular defintion for Ψ. Rather, for each specific input type $\mathbb{T}\ \mathbb{P}$, we need to generate a corresponding *UDF*, that also returns $\mathbb{T}\ \mathbb{P}$. These generated *UDF*s will be written in a log using the **WriterT** monad transformer. Their definitions are dependent on the type. In the scope of this work only atomar values and (unnested) rows are supported. For rows, there needs to be a distinction between custom row types (such as the result of a query or of a **ROW**-constructor) and table values. E.g. for table flights the type of v is `flights` and the corresponding *UDF* definition reflects that (Listing 4.1). The function body reflects the formal definition above: The `tuid` column as primary key is immutable and all other column values are appended using the ||–operator with the given provenance set.

```
CREATE OR REPLACE FUNCTION push(flights2, int4[]) RETURNS flights2
AS
$$
    SELECT ($1 :: flights2).tuid,
           (($1 :: flights2).origin) || $2,
           (($1 :: flights2).destination) || $2,
           (($1 :: flights2).distance) || $2,
           (($1 :: flights2).price) || $2
$$ LANGUAGE SQL;
```

Listing 4.1: UDF definition for table flights

When a type is not already defined in a relation, we need to introduce custom type definitions. This is because, in *UDF*s anonimous composed types are not supported. So, for each composed type, found at a expression with label α, there is a new type definition whereas α is added to its name to ensure uniqueness. Considering the example above, if we reduce the type to the `tuid` and `origin` columns, we need to define a new type for the generated *UDF* (Listing 4.2).

```
DROP TYPE IF EXISTS row7 CASCADE;
CREATE TYPE row7 AS (tuid int4, origin int4[]);
CREATE OR REPLACE FUNCTION push7(row7, int4[]) RETURNS row7
AS
$$
    SELECT ($1 :: row7).tuid,
           (($1 :: row7).origin) || $2
$$ LANGUAGE SQL;
```

Listing 4.2: UDF and type definition for custom row type

4.2.6.2 UDF Calls

As stated above, giving a function definition for Ψ is not going to be sufficient. Moreover, the function call may have to be wrapped in an additional structure to ensure the output type equals the input type and is valid *SQL*-code. There are two cases, in which function calls need to be wrapped: When pushing into a SFW-block and into a table expression within a **FROM**-clause. A simple cell expression *ec* can just be replaced with the function call. In this generalized example, the arguments for the push operator are flipped for better readability. This format is not valid *SQL* syntax.

```
         SELECT var.ρ AS ρ
Ψ3(ℓ3,          Ψ1(ℓ1, ec1²) AS col1, ... )
           FROM Ψ2(ℓ2, et²) AS var
```

Wrapping the relevant Ψ function calls each in a subquery, produces this expanded query, which finally is syntactically correct:

```
SELECT call3.*
  FROM (
        SELECT var.ρ AS ρ
               Ψ1(ℓ1, ec1²) AS col1, ...
          FROM (
                SELECT call2.*
                  FROM et² AS arg2
                       Ψ2(ℓ2, arg2) AS call2
               ) AS var
       ) AS arg3
       Ψ3(ℓ3, arg3) AS call3
```

In this example, nothing happens at the call of Ψ_1 because it is a cell expression and the function call returns a cell expression, too. The function call Ψ_2 replacing a table expression is wrapped in a subquery where the function result is dereferenced in the **SELECT**-clause. Finally, Ψ_3 called on a query is wrapped in another query analogous to a table expression.

4.3 Translation

The translation rules for Phase𝟙 and Phase𝟚 explained in Section 3.3 are each implemented as a function

```
tr :: Rule (SQL a) (SQL a)
```

in the modules Phase1 and Phase2 representing $\Rrightarrow$. This means that **`tr`** computes for any given *SQL* expression a *SQL* expression wrapped in a monadic context. In this case the monad is the **`OperSem`** monad used by the Database Research Group. **`OperSem`** is a combination of the **`ExceptT`**, **`WriterT`**, **`StateT`** and **`ReaderT`** monad transformers. Using the mtl-package the functionality of this monad stack can be accessed easily without any lifting. In Phase𝟙 the monad stack is only used to throw exceptions, however it is still useful during development to log information about the AST traversal and might have additional purpose in future work. In Phase𝟚 the **`WriterT`** functionality is used to collect *UDF* and type definitions and for handling exceptions.
Pattern matching the expressions in the AST allows us to project each rule onto a distinct case in the definition of **`tr`**. The function **`transformM`** exported by the LogParser allows us to 'skip' all nodes which are not relevant for How-Provenance by applying **`tr`** to all subtrees of those nodes. This ensures a full traversal of the AST and makes **`tr`** a total function.
When a translation rule is defined for an expression, the left side of the corresponding rule defines the pattern for the argument of **`tr`**. In this case, using **`WriterT`** subexpressions are recursively evaluated. For the various types of SFW expressions, plain pattern matching is not powerful enough. Hence, we introduce View Patterns. In this work, these are functions of type **`Expr -> Maybe Expr`** that can be applied to pattern expressions in the function definition, where the result is the pattern to be matched with wrapped in a **`Just`** if the pattern holds the desired properties.

The following uses the rules presented in Section 3.3 to exemplify in further detail how the translation process works. In all of them the language extension **`Expr -> Maybe Expr`** is used to bind each record selector inexplicitly to a variable with the construct `{..}`.

4.3.1 Operators

The implementation of the operator rule is fairly simple: In Phase1 subexpressions are recursively evaluated and the operator expression containing them is being returned.

```haskell
module Phase1 where

tr :: Rule (SQL a) (SQL a)

-- ...


tr EOpexpr {..} = do
    oprleft <- mapM tr oprleft -- map over Maybe Expr
    oprright <- tr oprright
    return $ EOpexpr {..}

-- ...
```

Snippet 4.3: Phase1 Rule: Binary Operator

Phase2 works similarly, only instead of returning the expression, the provenance set containing both sets of the subexpressions is returned together with the label of the operator expression.

```haskell
module Phase2 where

tr :: Rule (SQL a) (SQL a)

-- ...

tr (Lab l EOpexpr {..}) = do
    oprleft <- mapM tr oprleft -- map over Maybe Expr
    oprright <- tr oprright
    --operands are expected to yield an atomic value
    return $ fromMaybe pempty oprleft \/ oprright \/ pset l

-- ...
```

Snippet 4.4: Phase2 Rule: Binary Operator

4.3.2 Case

In this rule, again, subexpressions are evaluated. Additionally, free variables are collected to be included in the logging process.
In Phase1 a new inner **ECaseExpr** is created to form the argument for the logging *UDF*. Therefore, all *when* and *then* expressions are extracted. Finally, new *then* expressions are created to use the integer labels, as shown in Section 3.3.2.

```
module Phase1 where

tr :: Rule (SQL a) (SQL a)

-- ...

tr (Lab l ECaseExpr {..}) = do
    let freeVars = scan ECaseExpr {..}
    args <- mapM tr args
    defresult <- tr defresult
    let whens = map compareExpr args
    let thens = map result args
    let innerCase = ECaseExpr
                    { arg = Nothing
                    , args = zipWith createWhenExpr whens [1..]
                    , defresult = Lit (ConstI 0) (TAtom "int4") Nothing
                    , typ = TRow []
                    , location = Nothing
                    }
    unless (isNothing arg) (throwError "CASE: pattern does not match")
    let arg = Just $ writeCase $ toLiteral l:innerCase:map tuidRef freeVars
    let args = zipWith createThenExpr thens [1..]
    return $ ECaseExpr {..}

-- ...
```

Snippet 4.5: Phase1 Rule: Case

For Phase2 *UDF* and type definitions are created and logged based on the return type of the case expression. The provenance of *when* expressions is culminated and a function to read the log is added to the top of the case expression. Finally, the corresponding label is pushed into the result of the whole expression.

```
module Phase2 where

tr :: Rule (SQL a) (SQL a)

-- ...

tr (Lab l ECaseExpr {..}) = do
    let freeVars = scan ECaseExpr {..}
    let (udf,call) = push typ l
    tell $ LogC [udf] $ typeDef typ l
    args <- mapM tr args
    defresult' <- tr defresult
    let whens = map pconcat $ culminate $ map compareExpr args
    let thens = map result args
    unless (isNothing arg) (throwError "CASE: pattern does not match.")
    let arg = Just $ readCase $ toLiteral l:map tuidRef freeVars
    let args = zipWith3 pushWhenExprs whens thens [1..]
    let defresult = call [defresult', whyProv (last whens)]
    return $ call [ECaseExpr {..}, pset l]

-- ...
```

Snippet 4.6: Phase2 Rule: Case

4.3.3 n-way-Join

Here, the implementation additionally introduces the usage of the language extension **ViewPatterns** to specify the kind of query block. The pattern only matches with the specifications that need to be ensured by normalization, if this is not the case, no expression will match and the compiler throws an exception.
In both phases again, the expression is scanned for free variables and subexpressions are recursively evaluated.
Phase1 adds the tuid column to the *select* clause by passing the label and all relevant tuple identifiers to the logging function. This logging function is then added to the top of the *select* statement, while the rest stays the same.

```
{-# LANGUAGE ViewPatterns #-}
-- ...

```

```haskell
module Phase1 where

import qualified ViewPatterns as VP
-- ...

tr :: Rule (SQL a) (SQL a)

-- ...

-- n-way Join
tr (Lab l (VP.nWayJoin -> Just QBlockGeneric {..})) = do
    let freeVars = scan QBlockGeneric {..}
    from <- mapM tr from
    whereEx <- mapM tr whereEx
    select' <- mapM tr select
    -- list of all toLiteral columns of all tables involved
    let colRef = map (tuidRef . rRowname) from
    let cast = toLiteral l -- convert Haskell label to Literal in postgres
    -- add writeJoin to select
    let select = writeJoin (cast:colRef++map tuidRef freeVars):select'
    return $ QBlockGeneric {..}

-- ...
```

Snippet 4.7: Phase1 Rule: n-way join

In Phase2 the logging function is added to the bottom of the from clause to form a join. The locally bound function ***p*** is used to add provenance from the *where* clause to all expressions in the *select* clause, while the original *where* clause is nullified. The pushing-process including logging is here abbreviated for query blocks to a singular function ***psi***.

```haskell
{-# LANGUAGE ViewPatterns #-}
-- ...

module Phase2 where

import qualified ViewPatterns as VP
-- ...

tr :: Rule (SQL a) (SQL a)
```

```haskell

-- ...

-- n-way Join
tr (Lab l (VP.nWayJoin -> Just QBlockGeneric {..})) = do
    let freeVars = scan QBlockGeneric {..}
    from' <- mapM tr from
    let colRef = map (tuidRef . rRowname) from' -- get all rownames
    let cast = toLiteral l -- convert Haskell label to Literal in postgres
    -- last element of from references readJoin
    let from = from' ++ [readJoin (cast:colRef++map tuidRef freeVars)]
    select' <- mapM tr select
    whereEx' <- mapM tr whereEx
    -- add atomar Push for each colRef
    let p ETargetEx {..} = ETargetEx
                            {expr= expr \/ fromMaybe pempty whereEx',..}
    let select = tuidDeRef:map p select'
    let whereEx = Nothing
    psi (QBlockGeneric {..},l) -- push l and log udf/type def.

-- ...
```

Snippet 4.8: Phase2 Rule: n-way join

5 | Outlook & Conclusion

We have succesfully implemented the core process required for automating how–provenance. The general goal of this work has been achieved: For the central queriy constructs of *SQL* a reliable translation can be generated that is executable by a DBMS and returns the provenance data as expected. Additional rules can easily be inserted by simply adding the relevant pattern to the translation function. However, there are various other tasks to be done in possible further work:

Push is not yet implemented as a total function because at this moment nested rows can not be dealt with. These require a broader implementation of new defined types for each sub–row and a more case–sensitive recursive definition of the Ψ operator.

Furthermore, the automatically defined *UDF*s can be subject to a vast optimization by removing duplicate type and function definitions and replacing atomar function definitions with the *SQL*–internal union operator. This can reduce the runtime of Phase2 queries significantly, because *push* operations impose the biggest boilerplate in the generated code.

One prerequisite for the compiler to work reliably is the normalization of input queries. This is particularly important for the distinction of SFW-expressions. Normalizing by hand is not only inconvenient, it also introduces additional *how*-provenance which is not desired. In further work, by automating the normalization process after annotating the input query, this problem can be evaded.

Furthermore, for a more day-to-day orientated usage, the addition of a graphical user interface is an important step. It may help greatly for debugging because *how*–provenance could be visualized e.g. by hovering over the query result. A lot of editors support similar plugins for different programing languages already, so an extension of this sort can be easily imagined.

Finally, as suggested by the Y–operator in the appendix, the translation can be expanded to include *why*–provenance. This way the provenance of input cells, which at this point is empty, would be taken into account, too.

Bibliography

[CB74] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, page 249–264, New York, NY, USA, 1974. Association for Computing Machinery. doi:10.1145/800296.811515.

[CCT09] J. Cheney, L. Chiticariu, and W.C. Tan. *Provenance in Databases: Why, How, and Where.* Foundations and Trends in Databases, Issue 4. Now Publishers, 2009. URL: https://books.google.de/books?id=0HD42deeWWEC.

[GKT07] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance Semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, page 31–40, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1265530.1265535.

[Hir17] Denis Hirn. Compilation of SQL into KL. B.Sc. Thesis, 2017. URL: https://db.inf.uni-tuebingen.de/attachments/thesis-hirn-2017.pdf.

[Jon03] Simon Peyton Jones. Haskell 98 Languages and Libraries. 2003. URL: https://www.haskell.org/definition/haskell98-report.pdf.

[Lud03] Bertram Ludäscher. A Brief Tour through Provenance in Scientific Workflows and Databases. 2003. URL: http://hdl.handle.net/2142/89717.

[MDG18] Tobias Müller, Benjamin Dietrich, and Torsten Grust. You Say 'What', I Hear 'Where' and 'Why' — (Mis-)Interpreting SQL to Derive Fine-

Grained Provenance. pages 1536–1549, Rio de Janeiro, Brazil, 2018. URL: http://www.vldb.org/pvldb/vol11/p1536-muller.pdf.

[OMG18] Daniel O'Grady, Tobias Müller, and Torsten Grust. How 'How' Explains What 'What' Computes — How-Provenance for SQL and Query Compilers. London, UK, 2018. URL: https://www.usenix.org/system/files/conference/tapp2018/tapp2018-paper-ogrady.pdf.

[Par18] Gabriel Paradzik. Language-Level Provenance Analysis of SQL. B.Sc. Thesis, 2018. URL: https://db.inf.uni-tuebingen.de/attachments/paradzik-2018.pdf.

[pos] PostgreSQL Documentation. URL: https://www.postgresql.org/docs/.

All *URL*s have been checked on August 13, 2020.

Appendices

A | List of Translation Rules

This chapter offers a complete list of all formal translation rules implemented in the scope of this work. As a stepping stone for further work the Y–operator is included in these rules. With it, data provenance can easily be turned into why–Provenance. In the scope of this work is only how–Provenance, hence for now it is implemented as identity.

A.1 Literals

LITERAL

$$\ell^{\alpha} \Rrightarrow^{cell} (\ell, \{\alpha\})$$

A.2 Operators

BINOP SCALAR

$$\frac{ec_1^{\alpha_1} \Rrightarrow^{cell} (ec_1^{\mathbb{1}}, ec_1^{\mathbb{2}}) \qquad ec_2^{\alpha_2} \Rrightarrow^{cell} (ec_2^{\mathbb{1}}, ec_2^{\mathbb{2}})}{(ec_1^{\alpha_1} \circledast ec_2^{\alpha_2})^{\alpha} \Rrightarrow^{cell} (ec_1^{\mathbb{1}} \circledast ec_2^{\mathbb{1}}, ec_1^{\mathbb{2}} \cup ec_2^{\mathbb{2}} \cup \{\alpha\})}$$

A.3 Column References

COLUMN

$$\frac{ec^{\alpha} \Rrightarrow^{cell} (ec^{\mathbb{1}}, ec^{\mathbb{2}})}{(ec^{\alpha} . col^{\alpha_2})^{\alpha_0} \Rrightarrow^{cell} (ec_1^{\mathbb{1}} . col, \Psi(ec_1^{\mathbb{2}} . col, \{\alpha_0, \alpha_2\}))}$$

A.4 Case

CASE

$$\frac{\left| \; ec_{w_i}^{\alpha_{w_i}} \Rrightarrow^{cell} (ec_{w_i}^{\mathbb{1}}, ec_{w_i}^{\mathbb{2}}) \right|_{i=1\ldots} \quad \left| \; ec_{t_i}^{\alpha_{t_i}} \Rrightarrow^{cell} (ec_{t_i}^{\mathbb{1}}, ec_{t_i}^{\mathbb{2}}) \right|_{i=0\ldots} \quad \{f_1,\ldots\} := \mathsf{fv}(e_{in}) \quad ⓛ := \mathsf{tolocation}(\alpha) \quad X^{\mathbb{1}} := \texttt{CASE writeCase}(ⓛ, \texttt{CASE WHEN } ec_{w_1}^{\mathbb{1}} \texttt{ THEN } 1 \; \vdots \; \texttt{ELSE } 0 \texttt{ END}, f_1.\rho, \ldots) \texttt{ WHEN } 1 \texttt{ THEN } ec_{t_1}^{\mathbb{1}} \; \vdots \; \texttt{ELSE } ec_{t_0}^{\mathbb{1}} \texttt{ END} \quad X^{\mathbb{2}} := \texttt{CASE readCase}(ⓛ, f_1.\rho, \ldots) \texttt{ WHEN } 1 \texttt{ THEN } \Psi(ec_{t_1}^{\mathbb{2}}, \Upsilon(ec_{w_1}^{\mathbb{2}})) \; \vdots \; \texttt{WHEN } i \texttt{ THEN } \Psi(ec_{t_i}^{\mathbb{2}}, \Upsilon(ec_{w_1}^{\mathbb{2}} \cup \ldots \cup ec_{w_i}^{\mathbb{2}})) \; \vdots \; \texttt{ELSE } \Psi(ec_{t_0}^{\mathbb{2}}, \Upsilon(ec_{w_1}^{\mathbb{2}} \cup \ldots)) \texttt{ END}}{e_{in} := \left(\texttt{CASE WHEN } ec_{w_1}^{\alpha_{w1}} \texttt{ THEN } ec_{t_1}^{\alpha_{t1}} \; \vdots \; \texttt{ELSE } ec_{t_0}^{\alpha_{t0}} \texttt{ END} \right)^{\alpha} \Rrightarrow^{cell} (X^{\mathbb{1}}, \Psi(X^{\mathbb{2}}, \{\alpha\}))}$$

A.5 Row Constructors

Row

$$\frac{\left| \; ec_i^{\alpha_i} \Rrightarrow^{cell} (ec_i^{\mathbb{1}}, ec_i^{\mathbb{2}}) \right|_{i=1\ldots} \quad \rho := \mathsf{totuid}(\alpha_0)}{(\texttt{ROW}(ec_1^{\alpha_1}, \ldots))^{\alpha} \Rrightarrow^{cell} (\texttt{ROW}(\rho, ec_1^{\mathbb{1}}, \ldots), \Psi(\texttt{ROW}(\rho, ec_1^{\mathbb{2}}, \ldots), \{\alpha\}))}$$

A.6 Values

Values

$$\frac{\left|\ ec_i^{\alpha_i} \Mapsto^{cell} (ec_i^{\mathbb{1}}, ec_i^{\mathbb{2}})\ \right|_{i=1\ldots}}{(\texttt{VALUES}\ \ (ec_1^{\alpha_1},\ldots))^{\alpha} \Mapsto^{table} (\texttt{VALUES}\ \ (ec_1^{\mathbb{1}},\ldots), \Psi(\texttt{VALUES}\ \ (ec_1^{\mathbb{2}},\ldots),\{\alpha\}))}$$

A.7 Union All

Union ALL

$$\frac{\begin{array}{c}
\textcircled{l} \coloneqq \mathsf{tolocation}(\alpha) \quad \{f_1,\ldots\} \coloneqq \mathsf{fv}(et_1^{\alpha_1}) \\
[col_1,\ldots] \coloneqq \mathsf{columns}(et_1) \\
et_1^{\alpha_1} \Mapsto^{table} (et_1^{\mathbb{1}}, et_1^{\mathbb{2}}) \qquad et_2^{\alpha_2} \Mapsto^{table} (et_2^{\mathbb{1}}, et_2^{\mathbb{2}}) \\
X^{\mathbb{1}} \coloneqq \begin{array}{l} \texttt{SELECT}\ \ \texttt{writeUnion}\ (\textcircled{l}, var.\rho, f_1.\rho, ,\ldots)\ \ \texttt{AS}\ \rho, \\ \qquad var.col_1,\ldots \\ \texttt{FROM}\ et_1^{\mathbb{1}}\ \texttt{AS}\ var \end{array} \\
X^{\mathbb{2}} \coloneqq \begin{array}{l} \texttt{SELECT}\ \ \texttt{readUnion}\ (\textcircled{l}, var.\rho, f_1.\rho, ,\ldots)\ \ \texttt{AS}\ \rho, \\ \qquad var.col_1,\ldots \\ \texttt{FROM}\ et_1^{\mathbb{2}}\ \texttt{AS}\ var \end{array}
\end{array}}{(et_1^{\alpha_1}\ \texttt{UNION}\ \ \texttt{ALL}\ et_2^{\alpha_2})^{\alpha} \Mapsto^{table} (X^{\mathbb{1}}\ \texttt{UNION}\ \ \texttt{ALL}\ et_2^{\mathbb{1}}, \Psi(X^{\mathbb{2}}\ \texttt{UNION}\ \ \texttt{ALL}\ et_2^{\mathbb{2}},\{\alpha\}))}$$

A.8 With

With

$$\frac{\begin{array}{c}
\left|\ et_i^{\alpha_i} \Mapsto^{table} (et_i^{\mathbb{1}}, et_i^{\mathbb{2}})\ \right|_{i=0,\ldots} \\
X^{\mathbb{1}} \coloneqq \texttt{WITH}\ \ (tab\mathbb{1}_1\ \texttt{AS}\ et_1^{\mathbb{1}},\ldots)\ et_0^{\mathbb{1}} \\
X^{\mathbb{2}} \coloneqq \texttt{WITH}\ \ (tab\mathbb{2}_1\ \texttt{AS}\ et_1^{\mathbb{2}},\ldots)\ et_0^{\mathbb{2}}
\end{array}}{(\texttt{WITH}\ \ (tab_1\ \texttt{AS}\ et_1^{\alpha_1},\ldots)\ et_0^{\alpha_0})^{\alpha} \Mapsto^{table} (X^{\mathbb{1}}, \Psi(X^{\mathbb{2}},\{\alpha\}))}$$

A.9 SFW: Plain Select From

SFW-Map

$$\frac{\begin{array}{c} \left| \, ec_i^{\alpha_i} \Mapsto^{cell} (ec_i^{\mathbb{1}}, ec_i^{\mathbb{2}}) \, \right|_{i=1..n} \\ et^{\alpha_0} \Mapsto^{table} (et^{\mathbb{1}}, et^{\mathbb{2}}) \\ X^{\mathbb{1}} := \begin{array}{rl} \texttt{SELECT} & var.\rho\ \texttt{AS}\ \rho \\ & \quad ec_1^{\mathbb{1}}\ \texttt{AS}\ col_1, \ldots \\ \texttt{FROM} & et^{\mathbb{1}}\ \texttt{AS}\ var \end{array} \\ X^{\mathbb{2}} := \begin{array}{rl} \texttt{SELECT} & var.\rho\ \texttt{AS}\ \rho \\ & \quad ec_1^{\mathbb{2}}\ \texttt{AS}\ col_1, \ldots \\ \texttt{FROM} & et^{\mathbb{2}}\ \texttt{AS}\ var \end{array} \end{array}}{\left(\begin{array}{rl} \texttt{SELECT} & ec_1^{\alpha_1}\ \texttt{AS}\ col_1, \ldots \\ \texttt{FROM} & et^{\alpha_0}\ \texttt{AS}\ var \end{array}\right)^{\alpha} \Mapsto^{table} (X^{\mathbb{1}}, \Psi(X^{\mathbb{2}}, \{\alpha\}))}$$

A.10 SFW: n–way join

SFW-Join

$$\frac{\begin{array}{c} \text{\textcircled{ℓ}} := \mathsf{cast}(\alpha) \quad \{f_1, \ldots\} := \mathsf{fv}(et_{inp}) \\ \left| \, ec_i^{\alpha_i} \Mapsto^{cell} (ec_i^{\mathbb{1}}, ec_i^{\mathbb{2}}) \, \right|_{i=0,..n} \\ \left| \, et_i^{\alpha_i} \Mapsto^{table} (et_i^{\mathbb{1}}, et_i^{\mathbb{2}}) \, \right|_{i=(n+1),..(n+m)} \\ X^{\mathbb{1}} := \begin{array}{rl} \texttt{SELECT} & \texttt{writeJoin}(\text{\textcircled{ℓ}}, var_1.\rho, \ldots, f_1.\rho, \ldots)\ \texttt{AS}\ \rho \\ & ec_1^{\mathbb{1}}\ \texttt{AS}\ col_1, \ldots \\ \texttt{FROM} & et_1^{\mathbb{1}}\ \texttt{AS}\ var_1, \ldots \\ \texttt{WHERE} & ec_0^{\mathbb{1}} \end{array} \\ X^{\mathbb{2}} := \begin{array}{rl} \texttt{SELECT} & log.\rho\ \texttt{AS}\ \rho \\ & \Psi(ec_1^{\mathbb{2}}, \Upsilon(ec_0^{\mathbb{2}}))\ \texttt{AS}\ col_1, \ldots \\ \texttt{FROM} & et_1^{\mathbb{2}}\ \texttt{AS}\ var_1, \ldots, \\ & \texttt{readJoin}(\text{\textcircled{ℓ}}, var_1.\rho, \ldots, f_1.\rho, \ldots)\ \texttt{AS}\ log \end{array} \end{array}}{et_{inp} := \left(\begin{array}{rl} \texttt{SELECT} & ec_1^{\alpha_1}\ \texttt{AS}\ col_1, \ldots \\ \texttt{FROM} & et_{n+1}^{\alpha_{n+1}}\ \texttt{AS}\ var_{n+1}, \ldots \\ \texttt{WHERE} & ec_0^{\alpha_0} \end{array}\right)^{\alpha} \Mapsto^{table} (X^{\mathbb{1}}, \Psi(X^{\mathbb{2}}, \{\alpha\}))}$$

A.11 SFW: Aggregates

SFW-Agg

$$\text{Ⓛ} := \mathsf{tolocation}(\alpha) \quad et^{\alpha_0} \Rrightarrow^{table} (et^{\mathbb{1}}, et^{\mathbb{2}}) \quad \{f_1, \ldots\} := \mathsf{fv}(et_{inp})$$

$$\Big|\ ec_i^{\alpha_i} \Rrightarrow^{cell} (ec_i^{\mathbb{1}}, ec_i^{\mathbb{2}})\ \Big|_{i=1..n} \quad \Big|\ ec_{gr_i}^{\alpha_{gri}} \Rrightarrow^{cell} (ec_{gr_i}^{\mathbb{1}}, ec_{gr_i}^{\mathbb{2}})\ \Big|_{i=1..n}$$

$$X^{\mathbb{1}} := \begin{array}{l} \textbf{SELECT}\ \ \texttt{writeAgg}\,(\text{Ⓛ}, \textbf{ARRAY\_AGG}\,(var.\rho), f_1.\rho, \ldots)\ \textbf{AS}\ \rho, \\ \qquad \circledast(ec_1^{\mathbb{1}})\ \textbf{AS}\ col_1, \ldots \\ \textbf{FROM}\ et^{\mathbb{1}}\ \textbf{AS}\ var \\ \textbf{GROUP BY}\ ec_{gr_1}^{\mathbb{1}}, \ldots \end{array}$$

$$X^{\mathbb{2}} := \begin{array}{l} \textbf{SELECT}\ \textbf{THE}(log.\rho)\ \textbf{AS}\ \rho, \\ \qquad \bigcup \Psi(ec_1^{\mathbb{2}}, \Upsilon(ec_{gr_1}^{\mathbb{2}} \cup \ldots))\ \textbf{AS}\ col_1, \ldots \\ \textbf{FROM}\ et^{\mathbb{2}}\ \textbf{AS}\ var, \\ \qquad \texttt{readAgg}\,(\text{Ⓛ}, var.\rho, f_1.\rho, \ldots)\ \textbf{AS}\ log \\ \textbf{GROUP BY}\ log.\rho \end{array}$$

$$et_{inp} := \left(\begin{array}{l} \textbf{SELECT}\ \circledast(ec_1^{\alpha_1})^{\alpha_{agg1}}\ \textbf{AS}\ col_1, \ldots \\ \textbf{FROM}\ et^{\alpha_0}\ \textbf{AS}\ var \\ \textbf{GROUP BY}\ ec_{gr_1}^{\alpha_{gr1}}, \ldots \end{array} \right)^{\alpha} \Rrightarrow^{table} (X^{\mathbb{1}}, \Psi(X^{\mathbb{2}}, \{\alpha\}))$$

A.12 SFW: Order By

SFW-ORDERBY

$$\textcircled{\ell} \coloneqq \mathsf{tolocation}(\alpha) \qquad et^{\alpha_0} \Mapsto^{table} (et^{\mathbb{1}}, et^{\mathbb{2}}) \qquad \{f_1, \ldots\} \coloneqq \mathsf{fv}(et_{inp})$$

$$\Big|\, ec_i^{\alpha_i} \Mapsto^{cell} (ec_i^{\mathbb{1}}, ec_i^{\mathbb{2}}) \,\Big|_{i=1..} \qquad \Big|\, ec_{dis_i}^{\alpha_{dis_i}} \Mapsto^{cell} (ec_{dis_i}^{\mathbb{1}}, ec_{dis_i}^{\mathbb{2}}) \,\Big|_{i=1..}$$

$$X^{\mathbb{1}} \coloneqq \begin{array}{l}
\textbf{SELECT}\ \texttt{writeFilter}\,(\textcircled{\ell}, var''.\rho, f_1.\rho, \ldots)\ \textbf{AS}\ \rho \\
\qquad var''.col_1\ \textbf{AS}\ col_1, \ldots \\
\textbf{FROM}\ (\\
\qquad \textbf{SELECT}\ var'.\rho\ \textbf{AS}\ \rho \\
\qquad\qquad ec_1^{\mathbb{1}}\ \textbf{AS}\ col_1, \ldots \\
\qquad \textbf{FROM}\ et^{\mathbb{1}}\ \textbf{AS}\ var' \\
\qquad \textbf{ORDER BY}\ col_{ord_1}\ \textbf{ASC}\,|\,\textbf{DESC}\ , \ldots \\
\qquad \textbf{DISTINCT ON}\ ec_{dis_1}^{\mathbb{1}}\ , \ldots \\
\qquad \textbf{LIMIT}\ \ell_l \\
\qquad \textbf{OFFSET}\ \ell_o \\
)\ \textbf{AS}\ var'' \\
\textbf{ORDER BY}\ var''.col_{ord_1}\ \textbf{ASC}\,|\,\textbf{DESC}\ , \ldots
\end{array}$$

$$X^{\mathbb{2}} \coloneqq \begin{array}{l}
\textbf{SELECT}\ log.\rho\ \textbf{AS}\ \rho \\
\qquad \Psi(ec_1^{\mathbb{2}}, \mathbb{p})\ \textbf{AS}\ col_1, \ldots \\
\textbf{FROM}\ et^{\mathbb{2}}\ \textbf{AS}\ var, \ldots, \\
\qquad \texttt{readFilter}\,(\textcircled{\ell}, var.\rho, f_1.\rho, \ldots)\ \textbf{AS}\ log
\end{array}$$

$$\mathbb{p} \coloneqq \{\alpha\} \cup \Upsilon(\{\alpha_{ord_1}\} \cup \ldots \cup var.col_{ord_1} \cup \ldots \cup ec_{dis_1}^{\mathbb{2}} \cup \ldots)$$

$$et_{inp} \coloneqq \left(\begin{array}{l}
\textbf{SELECT}\ ec_1^{\alpha_1}\ \textbf{AS}\ col_1, \ldots \\
\textbf{FROM}\ et^{\alpha_0}\ \textbf{AS}\ var \\
\textbf{ORDER BY}\ col_{ord_1}^{\alpha_{ord_1}}\ \textbf{ASC}\,|\,\textbf{DESC}\ , \ldots \\
\textbf{DISTINCT ON}\ ec_{dis_1}^{\alpha_{dis_1}}\ , \ldots \\
\textbf{LIMIT}\ \ell_l \\
\textbf{OFFSET}\ \ell_o
\end{array} \right)^{\alpha} \Mapsto^{table} (X^{\mathbb{1}}, X^{\mathbb{2}})$$

B | Sample Output

The following listing shows the output of the compiler presented in this work with the query in Listing 1.1 as input. This generated *SQL* code can be executed by Postgres and results in the desired Provenance data.

```sql
-- view definitions
CREATE MATERIALIZED VIEW flights1 AS
    SELECT nextval('tuid_seq') AS tuid,
           flights.origin AS origin,
           flights.destination AS destination,
           flights.distance AS distance,
           flights.price AS price
    FROM flights;
CREATE MATERIALIZED VIEW flights2 AS
    SELECT flights1.tuid AS tuid,
           ARRAY[] :: int4[] AS origin,
           ARRAY[] :: int4[] AS destination,
           ARRAY[] :: int4[] AS distance,
           ARRAY[] :: int4[] AS price
    FROM flights1;

-- phase 1 - query:
(SELECT writeunion(20, subquery1.tuid :: int4) AS tuid,
        subquery1.destination AS destination
 FROM (SELECT writejoin(8, RTE0.tuid :: int4) AS tuid,
              RTE0.destination AS destination
       FROM flights1 AS RTE0(tuid,
                              origin,
                              destination,
                              distance,
                              price)
       WHERE RTE0.price < 100
      ) AS subquery1(tuid, destination))
  UNION ALL
(SELECT writejoin(17, RTE2.tuid :: int4) AS tuid,
        RTE2.destination AS destination
 FROM flights1 AS RTE2(tuid,
                        origin,
                        destination,
```

```
                         distance,
                         price)
 WHERE RTE2.price > 200);

-- phase 2 - udf and type definitions:
DROP TYPE IF EXISTS row4 CASCADE;
CREATE TYPE row4 AS (tuid int4, origin int4[], destination int4[], distance int4[],
    price int4[]);
DROP TYPE IF EXISTS row8 CASCADE;
CREATE TYPE row8 AS (tuid int4, destination int4[]);
DROP TYPE IF EXISTS row10 CASCADE;
CREATE TYPE row10 AS (tuid int4, destination int4[]);
DROP TYPE IF EXISTS row13 CASCADE;
CREATE TYPE row13 AS (tuid int4, origin int4[], destination int4[], distance int4[],
    price int4[]);
DROP TYPE IF EXISTS row17 CASCADE;
CREATE TYPE row17 AS (tuid int4, destination int4[]);
DROP TYPE IF EXISTS row19 CASCADE;
CREATE TYPE row19 AS (tuid int4, destination int4[]);
DROP TYPE IF EXISTS row21 CASCADE;
CREATE TYPE row21 AS (tuid int4, destination int4[]);
CREATE OR REPLACE FUNCTION push4(flights2, int4[]) RETURNS flights2
AS
$$
    SELECT ($1 :: flights2).tuid,
           (($1 :: flights2).origin) || $2,
           (($1 :: flights2).destination) || $2,
           (($1 :: flights2).distance) || $2,
           (($1 :: flights2).price) || $2
$$ LANGUAGE SQL;
CREATE OR REPLACE FUNCTION push2(int4[], int4[]) RETURNS int4[]
AS
$$
    SELECT $1 || $2
$$ LANGUAGE SQL;
CREATE OR REPLACE FUNCTION push5(int4[], int4[]) RETURNS int4[]
AS
$$
    SELECT $1 || $2
$$ LANGUAGE SQL;
CREATE OR REPLACE FUNCTION push8(row8, int4[]) RETURNS row8
AS
$$
    SELECT ($1 :: row8).tuid, (($1 :: row8).destination) || $2
$$ LANGUAGE SQL;
CREATE OR REPLACE FUNCTION push10(row10, int4[]) RETURNS row10
AS
$$
    SELECT ($1 :: row10).tuid, (($1 :: row10).destination) || $2
$$ LANGUAGE SQL;
CREATE OR REPLACE FUNCTION push13(flights2, int4[]) RETURNS flights2
```

```
AS
$$
    SELECT ($1 :: flights2).tuid,
           (($1 :: flights2).origin) || $2,
           (($1 :: flights2).destination) || $2,
           (($1 :: flights2).distance) || $2,
           (($1 :: flights2).price) || $2
$$ LANGUAGE SQL;
CREATE OR REPLACE FUNCTION push11(int4[], int4[]) RETURNS int4[]
AS
$$
    SELECT $1 || $2
$$ LANGUAGE SQL;
CREATE OR REPLACE FUNCTION push14(int4[], int4[]) RETURNS int4[]
AS
$$
    SELECT $1 || $2
$$ LANGUAGE SQL;
CREATE OR REPLACE FUNCTION push17(row17, int4[]) RETURNS row17
AS
$$
    SELECT ($1 :: row17).tuid, (($1 :: row17).destination) || $2
$$ LANGUAGE SQL;
CREATE OR REPLACE FUNCTION push19(row19, int4[]) RETURNS row19
AS
$$
    SELECT ($1 :: row19).tuid, (($1 :: row19).destination) || $2
$$ LANGUAGE SQL;
CREATE OR REPLACE FUNCTION push21(row21, int4[]) RETURNS row21
AS
$$
    SELECT ($1 :: row21).tuid, ($1 :: row21).destination || $2
$$ LANGUAGE SQL;

-- phase 2 - query:
SELECT call21.*
FROM ((SELECT readunion(20, subquery1.tuid :: int4) AS tuid,
              subquery1.destination AS destination
       FROM (SELECT call10.*
             FROM (SELECT call8.*
                   FROM (SELECT log.tuid AS tuid,
                                (push2(RTE0.destination, ARRAY[2] :: int4[]))
                                ||
                                (((push5(RTE0.price, ARRAY[5] :: int4[]))
                                  ||
                                  (ARRAY[6] :: int4[]))
                                 ||
                                 (ARRAY[7] :: int4[])) AS destination
                         FROM (SELECT call4.*
                               FROM flights2 AS arg4(tuid,
                                                       origin,
```

```
                                                       destination,
                                                       distance,
                                                       price),
                                    LATERAL push4(arg4,
                                                  ARRAY[4] :: int4[]) AS call4(tuid,
                                                                                 origin
    ,
    destination,
    distance,
                                                                                 price)
                              ) AS RTE0,
                              readjoin(8, RTE0.tuid :: int4) AS log
                        ) AS arg8,
                        LATERAL push8(arg8, ARRAY[8] :: int4[]) AS call8(tuid,
    destination)
                  ) AS arg10(tuid, destination),
                  LATERAL push10(arg10, ARRAY[10] :: int4[]) AS call10(tuid,
                                                                           destination)
            ) AS subquery1)
        UNION ALL
      (SELECT call19.*
       FROM (SELECT call17.*
             FROM (SELECT log.tuid AS tuid,
                          (push11(RTE2.destination, ARRAY[11] :: int4[]))
                          ||
                          (((push14(RTE2.price, ARRAY[14] :: int4[]))
                            ||
                            (ARRAY[15] :: int4[]))
                           ||
                           (ARRAY[16] :: int4[])) AS destination
                   FROM (SELECT call13.*
                         FROM flights2 AS arg13(tuid,
                                                  origin,
                                                  destination,
                                                  distance,
                                                  price),
                              LATERAL push13(arg13,
                                             ARRAY[13] :: int4[]) AS call13(tuid,
                                                                              origin,
    destination,
                                                                              distance,
                                                                              price)
                        ) AS RTE2,
                        readjoin(17, RTE2.tuid :: int4) AS log
                  ) AS arg17,
                  LATERAL push17(arg17, ARRAY[17] :: int4[]) AS call17(tuid,
                                                                           destination)
```

```
            ) AS arg19(tuid, destination),
            LATERAL push19(arg19, ARRAY[19] :: int4[]) AS call19(tuid,
                                                                 destination))
    ) AS arg21,
    LATERAL push21(arg21,
                ARRAY[21] :: int4[]) AS call21(tuid,destination);
```

Listing B.1: Output of the compiler for **UNION ALL** query

Acknowledgements

Foremost, I would like to to express my gratitude to my supervisor Dr. Tobias Müller for his continuous support and for providing and explaining the theoretical input for this work, most notably the translation rules.
Moreover I have to thank Denis Hirn for the implementation of the *LogParser* and his support for understanding the usage of his libraries and some general Haskell–related topics.
Lastly, my gratitude goes to my brother Florian Engel and my friend Holly Ransom for proof reading this work.

Author’s Declaration

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

Datum, Ort, Unterschrift