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

Bachelorarbeit Informatik

Language-Level Provenance Analysis of SQL

Gabriel Paradzik

24. März 2018

Gutachter

Prof. Dr. Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Eberhard Karls Universität Tübingen

Betreuer

Tobias Müller
Wilhelm-Schickard-Institut für Informatik
Eberhard Karls Universität Tübingen

Paradzik, Gabriel:
Language-Level Provenance Analysis of SQL
Bachelorarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 01.12.2017 - 31.03.2018

Abstract

Zur Berechnung der Data Provenance einer SQL Query wird in dieser Bachelorarbeit der Ansatz der *Language-Level Provenance Analysis* verfolgt. Dieser Ansatz erfordert Transformationen der zu untersuchenden SQL Query basierend auf ihrer Semantik. Im Rahmen dieser Arbeit wurde ein Programm entwickelt, welches die dafür benötigten Transformationen automatisiert.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Aufbau 2

2 Grundlagen **3**

2.1 Tools 3

2.1.1 Haskell 3

2.1.2 LogParser 4

2.1.3 AST PrettyPrinter 4

2.1.4 PostgreSQL 5

2.1.4.1 Sequenzen 5

2.1.4.2 Sichten 5

2.1.4.3 User Defined Functions 6

2.2 SQL 6

2.2.1 Normalisierung 8

2.3 Data Provenance 9

2.3.1 Vorbereitung 11

2.3.2 Phasen der Provenance 11

2.3.2.1 Phase 1 13

2.3.2.2 Phase 2 14

2.3.3 Typen der Provenance 15

2.3.3.1 Where-Provenance 15

2.3.3.2 Why-Provenance 16

2.4 Provenance Toolchain 16

3 Implementierung **19**

3.1 Aufbau 19

3.1.1 Bedienung 20

3.2 Abstract Syntax Tree 20

3.3 Provenance-Sets 22

3.4 Regelwerk 25

3.4.1 Vorbereitung 28

3.4.2 Regelwerk Expr . . . 29
3.4.2.1 Literal . . . 29
3.4.2.2 Spaltenverweis . . . 29
3.4.2.3 Tabellenverweis . . . 29
3.4.2.4 Operator . . . 29
3.4.2.5 Tabellenwertige Funktion . . . 30
3.4.3 Regelwerk Query . . . 31
3.4.3.1 Select . . . 31
3.4.3.2 Join . . . 31
3.4.3.3 Group By . . . 32
3.4.3.4 Select From . . . 33
3.4.3.5 Order By . . . 33
3.4.3.6 Window . . . 34
3.4.3.7 With . . . 35

4 Performance Analyse **37**
4.1 Transaction Processing Performance Council . . . 37
4.2 Durchführung . . . 37

5 Ausblick **41**
5.1 Fazit . . . 41
5.2 Future Work . . . 41
5.2.1 Darstellung der Provenance-Sets . . . 41
5.2.2 Unterstützung weiterer Queries . . . 42
5.2.3 Visualisierung . . . 42

Literaturverzeichnis **45**

Abkürzungsverzeichnis

ADT Algebraischer Datentyp.

AST Abstract Syntax Tree.

KL Kernel Language.

TPC Transaction Processing Performance Council.

UDF User Defined Function.

1 Einleitung

Datenbanken werden weltweit zur persistenten Speicherung von Daten verwendet. Die große Mehrheit davon sind relationale Datenbankmanagementsysteme, welche strukturierte Daten in tabellarischer Form speichern. SQL ist eine standardisierte Programmiersprache zum Lesen und Manipulieren von Tabellen einer Datenbank. Eine lesende SQL Query, die `SELECT`-Query, beantwortet nur eine Frage: *Wie lautet das Ergebnis*? Doch nicht immer ist das pure Ergebnis von Interesse, sondern auch der Weg dorthin. Fragen wie „*Welche* Werte haben zur Berechnung des Ergebnisses beigetragen? “ oder „*Warum* sind bestimmte Daten im Ergebnis vorhanden? “ liegen außerhalb des Aufgabenbereichs von SQL und sind Teil der *Data Provenance*.

Die *Data Provenance* betrachtet die Herkunft von Daten und spielt besonders im Bereich der Datenbanken und Big Data eine große Bedeutung. Dadurch können Ergebnisse einer Query auf ihre Korrektheit geprüft werden und Fehler bis hin zur Quelle zurückverfolgt werden.

Abbildung 1.1 zeigt im linken Teil die Tabelle `apartments`, welche Informationen über Wohnungen in Tübingen und Hamburg enthält. Die Query in Listing 1.1 berechnet für jede Stadt den Durchschnittspreis der Wohnungen. Das Ergebnis der Query ist im rechten Teil der Abbildung 1.1 zu sehen. Von Interesse ist nun die Beantwortung der beiden oben formulierten Fragen in Bezug auf den Durchschnittspreis aller Wohnungen aus Tübingen.

Alle Werte in der Ursprungstabelle, welche direkt bei der Berechnung des Durchschnittspreises beteiligt sind, sind aqua eingefärbt. Diese entsprechen den einzelnen Wohnungspreisen aller Wohnungen aus Tübingen.

Da die Datensätze aus `apartments` nach dem Stadtnamen gruppiert werden, sind diese indirekt in Form von Kontrollentscheidungen an der Berechnung beteiligt. Diese Tabellenzellen sind beige eingefärbt und entsprechen allen Vorkommen des Stadtnamens Tübingen.

```
SELECT city       AS city,
       AVG(price) AS "∅ price"
FROM   apartments
GROUP BY city
```

Listing 1.1: Query Q

apartments

city	area	price
Tübingen	60	773,00 €
Hamburg	77	1.350,00 €
Tübingen	120	1.004,00 €
Tübingen	45	530,00 €
Hamburg	50	999,00 €

Result

city	Ø price
Hamburg	1.174,50 €
Tübingen	769,00 €

Abbildung 1.1: Data Provenance mit *ProvVis*[1] visualisiert

Es gibt zahlreiche Möglichkeiten die *Data Provenance* einer Query zu berechnen. Am Lehrstuhl für Datenbanken der Universität Tübingen wurde ein Verfahren dafür entwickelt, welches auf Transformationen der Eingabe Query beruht.

Ziel der Bachelorarbeit ist die Entwicklung des Programms *SQLProv*, welches die auf einem Regelwerk basierenden Transformationen automatisiert. Das implementierte Regelwerk soll alle gängigen SQL Konstrukte umfassen (`GROUP BY`, `WINDOW`, `WITH`, . . .). Außerdem soll die Implementierung der *Provenance-Sets* abstrakt geschehen, so dass eine Änderung dieser nur eine minimale Anpassung an *SQLProv* erfordert.

1.1 Aufbau

Die Arbeit ist in 5 Kapitel unterteilt. In Kapitel 2 werden die verwendeten Tools, welche im Zusammenhang mit *SQLProv* verwendet wurden, vorgestellt und ein Einblick in die Berechnung der *Data Provenance* gegeben. Zum Schluss des Kapitels, wird *SQLProv* in die existierende Provenance Toolchain eingereiht. Dies bildet die Grundlage für Kapitel 3, welches auf die Implementierung und Features des Programms eingeht. Anschließend wird eine Performance Analyse der aus *SQLProv* resultierenden Queries durchgeführt. Im letzten Kapitel wird ein Fazit gezogen und mögliche Weiterentwicklungen des Projekts dargestellt.

[1] ProvVis ist eine Visualisierung der *Data Provenance* im Browser und ist nebenbei als Hilfstool entstanden. Sie ist im Repository unter `PgLlProvenance/ProvVis/` zu finden.

2 Grundlagen

Dieses Kapitel beginnt mit einen Überblick über die Abhängigkeiten und verwendeten Tools von *SQLProv*. Anschließend wird auf die Semantik von `SELECT` Abfragen und die Normalisierung dieser eingangen. Zum Schluss wird die Vorgehensweise zur Berechnung der *Data Provenance* näher erläutert und *SQLProv* in die existierende Toolchain eingeordnet.

2.1 Tools

2.1.1 Haskell

Haskell ist eine rein funktionale Programmiersprache, welche im Jahr 1990 in der ersten Version veröffentlicht wurde. Am Lehrstuhl für Datenbanken wird Haskell für eine Vielzahl von Forschungsprojekten eingesetzt. Gerade für die Forschung im Bereich der Provenance existierte bereits eine umfangreiche Codebasis in Haskell, weshalb die Umsetzung von *SQLProv* in Haskell eine Voraussetzung war. Folgende Features zeichnen Haskell aus:

- *Pure*: Alle Funktionen sind im mathematischen Sinne *pur*. Das bedeutet, dass der Rückgabewert einer Funktion ausschließlich von den Eingabewerten abhängt. Alle Seiteneffekte (z.B. das Lesen einer Datei oder das Schreiben eines Outputs) sind nur im *IO Monad* möglich. Dadurch lässt sich einer Funktion anhand der Signatur ansehen, ob diese seiteneffektbehaftet ist oder nicht. Damit findet eine klare Trennung zwischen *purem* und *nicht-purem* Code statt.
- *Statisch typisiert*: Das Programm wird während der Kompilation auf Typfehler geprüft. Damit ist das Programm frei von Typfehlern zur Laufzeit. Außerdem fallen dadurch viele potentielle Fehler früh in der Entwicklung auf.
- *Lazy*: Die Evaluation von Ausdrücken passiert nur bei Bedarf. Dadurch können in der Sprache Berechnungen auf unendlichen Datenstrukturen durchgeführt werden, solange diese nur einen endlichen Teil betrachten.
- *Algebraische Datentypen (ADTs)*: Haskell unterstützt die Definition von ADTs. Damit lassen sich Datenstrukturen auf natürliche Weise festhalten.

- *modular*: Ein Programm kann intern auf verschiedene Module aufgeteilt werden, um eigenständige Komponenten voneinander zu trennen. Es ist ebenfalls möglich externe Module (Programmbibliotheken) in ein Programm zu importieren.

Beim Design von Haskell wurde besonders viel Wert daraufgelegt, den syntaktischen Overhead minimal zu halten. Das macht Haskell Programme zugleich kompakt und lesbar. Um den Programmcode auf das Wesentliche zu beschränken ist ebenfalls eine geeignete Auswahl an Programmbibliotheken wichtig. Neben Standardmodulen wurden in *SQLProv* einige weitere nennenswerte Module verwendet.

- *optparse-applicative*[1] ist eine Programmbibliothek zum Parsen vom Kommandozeilenargumenten. Die Definition eines Parsers geschieht dabei mit Hilfe der Typklassen *Applicative* und *Alternative*. Das Ergebnis des Parsers ist ein benutzerdefinierter ADT, welcher alle Programmargumente in gewünschter Darstellung enthält.
- *lens*[2] ist eine Programmbibliothek, welche das Arbeiten mit tief verschachtelten Datenstrukturen vereinfacht. Damit lassen sich Werte innerhalb einer Datenstruktur mit einer kompakten Syntax auslesen und modifizieren.

2.1.2 LogParser

[Hir17] Der *LogParser* wurde am Lehrstuhl für Datenbanken der Universität Tübingen entwickelt und übersetzt eine SQL Query in einen Abstract Syntax Tree (AST). Dabei wurde eine Erweiterung für den PostgreSQL Server entwickelt, welche sich den internen Queryparser zunutze macht, um eine Query zu interpretieren und die Postgres-interne Queryrepräsentation zu extrahieren. Diese interne Darstellung wird schlussendlich vom *LogParser* in eine Haskell-interne Darstellung, dem AST, übersetzt. Dieser AST bildet die Grundlage für *SQLProv* zur Transformation von SQL Queries.

2.1.3 AST PrettyPrinter

Der *AST PrettyPrinter* ist ein Haskell Modul, welches die AST Darstellung des LogParsers zurück in SQL Quelltext übersetzt. Um die menschliche Lesbarkeit zu steigern, beinhaltet die Ausgabe Einrückungen und ANSI Farbcodes.

[1] Siehe `https://hackage.haskell.org/package/optparse-applicative`.
[2] Siehe `https://hackage.haskell.org/package/lens`.

2.1.4 PostgreSQL

PostgreSQL[3] ist ein objektrelationales Datenbanksystem und erschien 1997 in der ersten Version. In der aktuellen Version 10 unterstützt Postgres den SQL Standard in großen Teilen und bietet ein umfangreiches Interface für Erweiterungen. Aus diesem Grund ist *Postgres* am Lehrstuhl das Datenbanksystem der Wahl. Während das in dieser Arbeit benutzte Verfahren zur Berechnung der Provenance unabhängig vom Datenbanksystem ist, werden einige nützliche Funktionen von Postgres verwendet, welche im Folgenden beschrieben werden.

2.1.4.1 Sequenzen

Innerhalb einer Datenbank können *Sequenzen* erstellt werden, welche einen Startwert, einen Endwert und eine Schrittweite besitzen. Standardmäßig startet eine *Sequenz* bei 1 mit einer Schrittweite von +1 und endet beim Erreichen des Limits eines 64-bit Integers. Beim Erreichen des Endwertes wird die Sequenz standardmäßig beim Startwert fortgesetzt. Nach Definition einer Sequenz, kann mit Hilfe der Funktion `nextval` der nächste Wert der Sequenz bestimmt werden.

```
CREATE SEQUENCE my_seq;
SELECT nextval('my_seq'); -- Result: 1
SELECT nextval('my_seq'); -- Result: 2
```

Listing 2.1: Beispielhafte Verwendung einer Sequenz

2.1.4.2 Sichten

Postgres unterstützt die Erstellung von *Sichten*. Damit kann eine beliebige Query an einen Namen gebunden werden, welche in anderen Queries als Tabellenreferenz benutzt werden kann. Das Ergebnis dieser Query wird bei jedem Aufruf erneut berechnet. Bei der Erstellung einer *materialisierten Sicht* wird das Ergebnis der Query bei der Erstellung der Sicht berechnet und für zukünftige Benutzungen zwischengespeichert.

```
CREATE [MATERIALIZED] VIEW v (c_1, ..., c_n)
  AS q;
```

Listing 2.2: Vereinfachte `CREATE VIEW` Syntax

[3] Im Folgenden auch *Postgres* genannt

```
CREATE VIEW my_view (col)
  AS SELECT 1;
SELECT col FROM my_view; -- Result: 1
```

Listing 2.3: Beispielhafte Verwendung einer Sicht

2.1.4.3 User Defined Functions

Unter *Postgres* lassen sich User Defined Functions (UDFs) definieren, welche eine feste Anzahl an Parametern entgegennehmen und einen Wert bzw. eine Tabelle zurückgeben. Der Funktionskörper enthält eine Abfolge von Queries, welche sowohl lesend (`SELECT`), als auch schreibend (`INSERT`, `UPDATE`, ...) sein dürfen. Das Ergebnis der UDF entspricht dem Ergebnis der letzten Query q_n.

```
CREATE FUNCTION f(x_1 τ_1, ..., x_n τ_n) RETURNS τ AS
$$
  q_1;
  ...
  q_n
$$
LANGUAGE SQL;
```

Listing 2.4: Vereinfachte `CREATE FUNCTION` Syntax

2.2 SQL

[Don74] *SQL*, früher *SEQUEL*, wurde als eine Abfragesprache für Datenbanken entwickelt. Mit besonders verbosen Sprachkonstrukten und dem Ziel, dass sich eine Abfrage wie ein englischer Satz liest, sollte diese Sprache auch technisch unversierten Personen zugänglich sein. Durch die Weiterentwicklung des SQL Standards wurden viele neue Sprachfeatures hinzugefügt, unter anderem: Window Functions, rekursive Queries und JSON Support. Besonders die rekursiven Queries bezeichnen einen Meilenstein, da SQL dadurch Turing-vollständig ist.

Listing 2.5 zeigt eine vereinfachte Form der Grammatik einer `SELECT` Query. Da der Aufbau einer solchen Query an den Aufbau der englischen Sprache angelehnt ist, werden die einzelnen Sprachkonstrukte vom Datenbanksystem nicht in Lesereihenfolge ausgeführt, sondern wie folgt.

1. Alle Tabellen in der `WITH`-Anweisung werden berechnet und temporär im Speicher gehalten.

2. Es wird das *kartesische Produkt* aller Tabellen der `FROM`-Klausel gebildet. Die einzelnen t_i sind dabei tabellenwertige Ausdrücke.

3. Die `WHERE`-Klausel entfernt alle Einträge aus dem Ergebnis des vorherigen Schritts, welche der Bedingung nicht entsprechen.

4. Das `GROUP BY` gruppiert das Ergebnis nach den angegebenen Spalten und berechnet die Aggregate, welche in den Ausdrücken der `SELECT`-Klausel vorkommen.

5. Die definierten *Windows* werden berechnet und die dazugehörigen Aggregate `... OVER ...` der `SELECT`-Klausel ausgeführt.

6. Nun werden alle Ausdrücke auf Basis der aktuellen Ergebnistabelle im `SELECT` berechnet. Diese Ausdrücke können einfache Spaltenreferenzen, aber auch Queries sein, welche einen atomaren Wert berechnen. In *SQL* ist eine einzeilige und einspaltige Tabelle gleichwertig mit einem atomaren Wert.

7. Die Ergebnistabelle wird nach den Ausdrücken der `ORDER BY`-Klausel sortiert.

8. Zum Schluss werden die Zeilen der Ergebnistabelle mit der `LIMIT` / `OFFSET`-Klausel limitiert. Man beachte, dass ein `LIMIT` / `OFFSET` ohne eine `ORDER BY`-Klausel unvorhersehbare Ergebnisse liefert.

```
1  [ WITH [ RECURSIVE ]
2    t'_1 (c_11, ...) AS (q_1),
3    ...
4    t'_n (c_n1, ...) AS (q_n) ]
5  SELECT e_1, ...
6  [ FROM     t_1, ... ]
7  [ WHERE    p_w ]
8  [ GROUP BY g_1, ... ]
9  [ HAVING   p_h ]
10 [ WINDOW   w AS ... ]
11 [ ORDER BY o_1, ... ]
12 [ LIMIT    l ]
13 [ OFFSET   m ]
```

Listing 2.5: `SELECT`-Block Grammatik

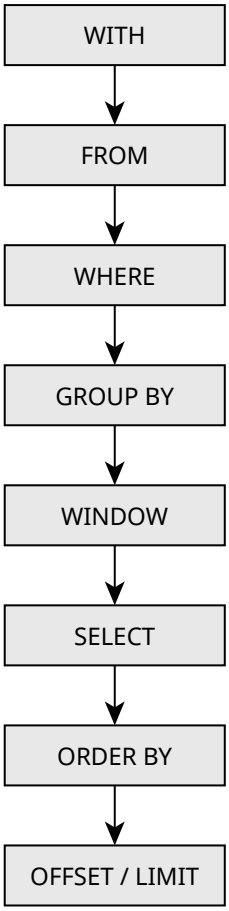

Abbildung 2.1: SELECT: Reihenfolge der Ausführung

2.2.1 Normalisierung

SQL besitzt eine Vielzahl an verschiedenen Konstrukten, um Queries zu formulieren. Diese Konstrukte sind meist nicht voneinander abhängig, um eine valide Query zu schreiben. Um die Provenance einer Query zu berechnen wird diese basierend auf ihren verwendeten Konstrukten transformiert. Es wird schnell ersichtlich, dass sich aus der in Listing 2.5 gezeigten Grammatik eine Vielzahl von gültigen Queries ableiten lassen. Zu viele, um für jede Kombination der Sprachkonstrukte eine eigene Transformationsregel zu entwerfen.

```
1 SELECT e_1, ..., e_n
2 FROM t
3 WHERE p
4 GROUP BY g
5 ORDER BY o_1, ..., o_n
```

Listing 2.6: Nicht normalisierte Query

```
1 SELECT e_11, ..., e_1n
2 FROM (
3     SELECT e_21, ..., e_2n
4     FROM (
5         SELECT e_31, ..., e_3n
6         FROM t
7         WHERE p) AS t
8     GROUP BY g) AS t
9 ORDER BY o_1, ..., o_n
```

Listing 2.7: Normalisierte Query

Das Ziel der Normalisierung ist die Minimierung der notwendigen Transformationsregeln ohne Einschränkung der Funktionalität der Eingabe Query. Möglich wird das durch die Verlagerung von Teilen der Query in Subqueries, während die ursprüngliche Semantik unverändert bleibt. Listing 2.6 zeigt eine Query, welche die Eingabetabelle filtert, gruppiert und zum Schluss sortiert. Diese kann, wie in Listing 2.7 gezeigt, normalisiert werden, indem jede dieser drei Schritte in eine Subquery verlagert wird. Es reichen nun drei Transformationsregeln, um die normalisierte Query zu übersetzen. Zusätzlich dazu kann jede beliebige Kombination dieser Subquery mithilfe dieser Transformationsregeln übersetzt werden. Es sei an dieser Stelle noch angemerkt, dass die Reihenfolge der Aufteilung in Subqueries eine wichtige Rolle spielt. So muss zum Beispiel die Sortierung des Ergebnisses immer im äußersten Teil der Query stattfinden, da die Ordnung einer Tabelle in der `FROM`-Klausel nicht erhalten bleibt. Im Allgemeinen folgt die Aufsplittung in Reihenfolge der Auswertung der einzelnen Konstrukte (siehe dazu Abbildung 2.1).

Die Normalisierung einer Query ist nicht Teil von *SQLProv*. Um die Transformationen durchführen zu können, wird eine normalisierte Query als Eingabe erwartet.

2.3 Data Provenance

Unter *Data Provenance* (auch *Data Lineage* genannt) versteht man die Untersuchung der Herkunft von Daten. Für alle Prozesse, welche einen *Input* verarbeiten und basierend darauf einen *Output* berechnen, kann die Data Provenance betrachtet werden. Besitzen Input und Output eine gewisse Struktur, so können Beziehungen zwischen Teilstrukturen bis hin zu atomaren Einheiten einer Struktur hergestellt werden. Da in dieser Arbeit die Data Provenance von SQL in einem relationalen Datenbanksystem betrachtet wird, sind Ein- und Ausgabestrukturen Tabellen und die atomare Einheit eine Tabellenzelle.

Es gibt verschiedene Möglichkeiten die Provenance einer SQL Query zu berechnen. Am Lehrstuhl für Datenbanken werden zwei unterschiedliche Ansätze verfolgt.

- *KL-based Provenance Analysis*. [Mül16] Eine beliebige SQL Query wird basierend auf einem Regelwerk in die Programmiersprache Kernel Language (KL) übersetzt. KL ist eine am Lehrstuhl für diesen Zweck entworfene imperative Programmiersprache, welche keine Seiteneffekte zulässt. Mithilfe von bekannten Programmanalysetechniken wird anschließend die Provenance des KL-Programms und damit der SQL Query berechnet.
- *Language-Level Provenance Analysis*. Bei diesem Ansatz wird ausschließlich SQL verwendet, um die Provenance einer Query zu berechnen. Dazu wird eine Query in zwei Queries transformiert, welche Queries der Phase $\mathbb{1}$ und Phase $\mathbb{2}$ genannt werden. In Phase $\mathbb{1}$ wird die ursprüngliche Query ausgeführt und dabei Provenance-bezogene Daten protokolliert. Bei der Ausführung der Phase $\mathbb{2}$ Query werden die protokollierten Daten aus Phase $\mathbb{1}$ verwendet, um die Provenance zu berechnen.

Der Fokus dieser Arbeit liegt auf dem zweiten Ansatz, der *Language-Level Provenance Analysis*. Da die Berechnung der Provenance ausschließlich in SQL stattfindet, liegen die Ergebnisse in Tabellenform vor. Diese Ergebnistabelle enthält in jeder Zelle eine Menge von Verweisen auf Zellen der Eingabetabellen. Ein Verweis geschieht nicht etwa *by-value*, sondern über eine Zellenidentifikation[4] ϱ_i. Damit derartige Verweise möglich sind, müssen die von der Query verwendeten Tabellen vorbereitet werden. Die Vorbereitung der Eingabetabellen wird in Abschnitt 2.3.1 erklärt.

In den folgenden Unterkapiteln wird die Tabelle 2.1 für eine Veranschaulichung der *Data Provenance* verwendet. Diese beinhaltet die Stadt, die Fläche und den Preis von Wohnungen aus Deutschland.

apartments

city	area	price
Leipzig	64	525€
München	72	1370€
Tübingen	60	773€
Hamburg	77	1350€

Tabelle 2.1: Tabelle `apartments` mit Wohnungen

[4] Im Folgenden auch *Provenance-ID* genannt

2.3.1 Vorbereitung

Das Ziel der Vorbereitung ist eine Modifikation von `apartments`, welche für jede Zelle eine Provenance-ID ϱ_i enthält. Würden jedoch alle Zellen der `apartments` Tabelle mit ϱ_i durchnummeriert werden, fehlt die Zuordnung eines ϱ_i zu ihrem jeweiligen Wert, da eine Tabelle in einem relationalen Datenbanksystem keine definierte Ordnung besitzt. Aus diesem Grund erhalten beide Tabellen eine zusätzliche Spalte ϱ, welche die Zuordnung der Zeilen bewahrt. Die `apartments` Tabelle mit einer zusätzlichen Spalte zur Zeilenidentifikation wird `apartments`$^{\mathbb{1}}$ genannt und die Tabelle mit den Provenance-IDs ϱ_i lautet `apartments`$^{\mathbb{2}}$.

`apartments`$^{\mathbb{1}}$

ϱ	city	area	price
ϱ_1	Leipzig	64	525€
ϱ_2	München	72	1370€
ϱ_3	Tübingen	60	773€
ϱ_4	Hamburg	77	1350€

Tabelle 2.2: Phase $\mathbb{1}$ der Tabelle `apartments`

`apartments`$^{\mathbb{2}}$

ϱ	city	area	price
ϱ_1	$\{\varrho_5\}$	$\{\varrho_6\}$	$\{\varrho_7\}$
ϱ_2	$\{\varrho_8\}$	$\{\varrho_9\}$	$\{\varrho_{10}\}$
ϱ_3	$\{\varrho_{11}\}$	$\{\varrho_{12}\}$	$\{\varrho_{13}\}$
ϱ_4	$\{\varrho_{14}\}$	$\{\varrho_{15}\}$	$\{\varrho_{16}\}$

Tabelle 2.3: Phase $\mathbb{2}$ der Tabelle `apartments`

In den folgenden Tabellen der Phase $\mathbb{1}$ oder $\mathbb{2}$ wird die Spalte ϱ im Tabellenkopf weggelassen. Die Zeilenidentifikationen ϱ_i befinden sich als Annotation links von der Tabelle.

2.3.2 Phasen der Provenance

Die Idee, welche bei der *Language-Level Provenance Analysis of SQL* verfolgt wird, ist die ausschließliche Verwendung von SQL zur Berechnung der Provenance einer Query. Der vorgestellte Ansatz basiert vor allem auf der Verwendung von UDFs in Verbindung mit intern geführten Log-Tabellen. UDFs lassen Seiteneffekte zu, wie z.B. das Lesen, Einfügen und Aktualisieren von Zeilen in beliebigen Tabellen. Diese Eigenschaften einer UDF werden in den Phasen $\mathbb{1}$ und $\mathbb{2}$ benutzt, um Meta-Informationen zu speichern bzw. zu lesen. Die Normalisierung einer Query verwendet Subqueries um die ursprüngliche Query in einzelne Schritte zu verlagern, welche jeweils genau eine Operation ausführen (Filterung, Join, Aggregierung, ...). Zu jeder solchen Operation existiert eine `read`- und eine `write`-UDF (`readFilter`, `writeFilter`, ...) sowie eine dazugehörige Tabelle, welche zum Speichern bzw. Lesen dieser Informationen genutzt wird. Im Folgenden werden diese Tabellen *Log-Tabellen* genannt.

In Phase $\mathbb{1}$ wird die Query mit `write`-UDFs annotiert, um Meta-Informationen zu den Operationen in den *Log-Tabellen* zu speichern. Die Ergebnistabelle erhält dabei eine zusätzliche Spalte ϱ mit Zeilenidentifikationen ϱ_i, um eine Zuordnung zum Ergebnis aus Phase $\mathbb{2}$ herzustellen.

In Phase 2 werden die *Log-Tabellen* verwendet, um die ursprüngliche Query nachzuspielen. Anders als in Phase 1, arbeiten nun alle Berechnungen innerhalb der Query mit *Provenance-Sets*. Da in dieser Phase die konkreten Werte nicht vorhanden sind, müssen zum Beispiel Filterungen oder Aggregierungen mit Hilfe von UDFs und den protokollierten Meta-Informationen aus Phase 1 simuliert werden. Das Ergebnis der Phase 2 ist eine Tabelle, welche dieselbe Anzahl an Zeilen und Spalten wie die Ergebnistabelle aus Phase 1 besitzt. Anstatt Werten, enthält jede Tabellenzelle ein *Provenance-Set*, welches die Abhängigkeiten zu den Ursprungstabellen beschreibt.

Die beschriebenen Zusammenhänge sind in Abbildung 2.2 dargestellt.

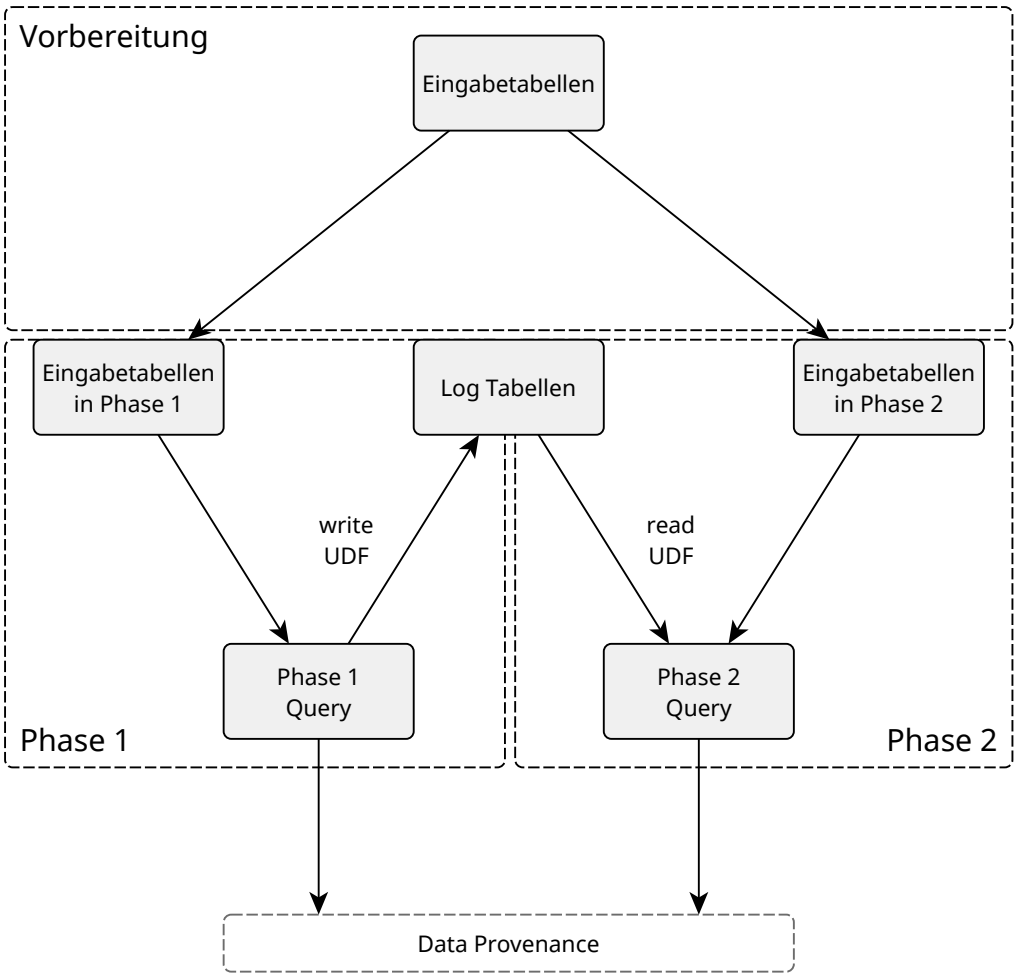

Abbildung 2.2: Übersicht über SQL-basierte Berechnung der Provenance

In den folgenden beiden Unterkapiteln wird das eben beschriebene Verfahren an einer beispielhaften Query Q näher erläutert. Q gibt alle Wohnungen zurück, dessen Miete günstiger als 1000€ ist.

```
1 SELECT city,
2        area,
3        price
4 FROM   apartments
5 WHERE price < 1000
```

Listing 2.8: Query Q

2.3.2.1 Phase 1

Die Query Q ist eine *filternde Query*. Das Ziel der Phase $\mathbb{1}$ ist die Protokollierung aller Zeilen, welche der in der `WHERE`-Klausel aufgeführten Bedingung entsprechen, damit dieses Verhalten in Phase $\mathbb{2}$ simuliert werden kann. Zur Identifizierung einer Zeile wird die dafür vorgesehene Spalte ϱ der Tabelle `apartments`$^{\mathbb{1}}$ verwendet. Durch das Hinzufügen von `writeFilter` im `SELECT` wie in Listing 2.9, wird diese für jede Zeile, welche die Bedingung in der `WHERE`-Klausel erfüllt, ausgeführt. Die `writeFilter`-UDF akzeptiert zwei Argumente. Das erste Argument dient als Identifizierung eines `writeFilter` Aufrufs. Falls in einer verschachtelten SQL Query mehrere Aufrufe getätigt werden, muss eine Unterscheidung zwischen diesen stattfinden, welche über das erste Argument mittels eines einfachen Integers stattfindet. Das zweite Argument ist die Zeilenidentifikation der zu filternden Tabelle. `writeFilter` speichert beide Argumente in der `logfilter` Tabelle und gibt das zweite Argument unverändert zurück. Das Ergebnis von $Q^{\mathbb{1}}$ ist in Tabelle 2.4 dargestellt.

```
1 SELECT writeFilter(1, a.ϱ) AS ϱ,
2        a.city,
3        a.area,
4        a.price
5 FROM   apartments¹ AS a(ϱ, city, area, price)
6 WHERE  a.price < 1000
```

Listing 2.9: Phase $\mathbb{1}$ der Query Q

$Q^{\mathbb{1}}$

	city	area	price
ϱ_1	Leipzig	64	525€
ϱ_3	Tübingen	60	773€

Tabelle 2.4: Ergebnis von $Q^{\mathbb{1}}$

Die Tabelle `logfilter`, welche vor der Ausführung von $Q^{\mathbb{1}}$ leer war, enthält nun die Zeilenidentifikationen der Zeilen, welche das `WHERE`-Prädikat erfüllen.

logfilter	
location	tuid
1	ϱ_1
1	ϱ_3

Tabelle 2.5: Tabelle `logfilter` nach Ausführung von Q

2.3.2.2 Phase 2

Ziel der Phase $\mathbb{2}$ ist das Lesen der Logs aus Phase $\mathbb{1}$, um das Verhalten der Query Q nachzustellen. Dazu wird die Query Q mit einem Aufruf der `readFilter`-UDF im `FROM`, in Form eines `LATERAL JOIN`s, modifiziert. Das `LATERAL` Keyword bewirkt, dass sich die Argumente der `readFilter`-UDF auf Spalten der Tabelle `apartments`$^{\mathbb{2}}$ beziehen dürfen. Für jede Zeile r aus `apartments`$^{\mathbb{2}}$ wird die UDF `readFilter` ausgeführt und das kartesische Produkt mit $\{r\}$ berechnet. Das Gesamtergebnis wird nun aus der Vereinigung aller kartesischen Produkte über alle Zeilen berechnet. Ist das Resultat der UDF eine leere Tabelle, so ist das kartesische Produkt ebenfalls leer. Das kann dazu führen, dass bestimmte Zeilen aus `apartments`$^{\mathbb{2}}$ nicht im Gesamtergebnis auftauchen. Dieses Verhalten ist bei einem `NATURAL JOIN` nicht möglich.

`readFilter` akzeptiert genau wie `writeFilter` eine *ID* und eine Zeilenidentifikation. Existieren diese in der `logfilter` Tabelle, so wird eine einzeilige und einspaltige Tabelle mit der Zeilenidentifikation zurückgegeben. Falls nicht, ist das Ergebnis eine leere Tabelle. Das bringt den gewünschten filternden Effekt, da `readFilter` anhand der übergebenen Argumente entscheidet, ob eine bestimmte Zeile aus `apartments`$^{\mathbb{2}}$ im Gesamtergebnis erhalten bleibt oder gefiltert wird. Der entscheidende Unterschied ist, dass in Phase $\mathbb{1}$ anhand von tatsächlichen Werten gefiltert wurde und in Phase $\mathbb{2}$ anhand der Zeilenidentifikationen.

```
1  SELECT filter.ϱ AS ϱ,
2         a.city,
3         a.area,
4         a.price
5  FROM   apartments^2 AS a(ϱ, city, area, price),
6         LATERAL readFilter(1, a.ϱ) AS filter(ϱ)
```

Listing 2.10: Phase $\mathbb{2}$ der Query Q

$Q^{\mathbb{2}}$	city	area	price
ϱ_1	$\{\varrho_5\}$	$\{\varrho_6\}$	$\{\varrho_7\}$
ϱ_3	$\{\varrho_{11}\}$	$\{\varrho_{12}\}$	$\{\varrho_{13}\}$

Tabelle 2.6: Ergebnis von $Q^{\mathbb{2}}$

2.3.3 Typen der Provenance

Ist die Rede von *Beziehungen zwischen Tabellenzellen*, können verschiedene Dinge gemeint sein. Aus welchen Eingabewerten wurde ein Ausgabewert berechnet (*Where-Provenance*)? Welche Eingabewerte haben dafür gesorgt, dass ein bestimmter Wert in der Ausgabetabelle vorhanden ist (*Why-Provenance*)? Diese beiden Provenance-Typen werden im Folgenden genauer betrachtet und anhand der `apartments` Tabelle verdeutlicht.

2.3.3.1 Where-Provenance

Die *Where-Provenance* beschreibt, durch welche Eingabewerte ein bestimmter Ausgabewert beeinflusst wurde. Dazu zählen zum Beispiel arithmetische und boolesche Operationen. Zur Verdeutlichung der *Where-Provenance* wird die `apartments` Tabelle verwendet. Diese enthält Informationen über verfügbare Wohnungen in verschiedenen Städten. Uns interessieren nun die Kosten pro Quadratmeter und die daraus resultierende *Where-Provenance*.

```
1 SELECT city          AS city,
2        area          AS area,
3        price         AS price,
4        price / area AS "price per m²"
5 FROM   apartments
```

Listing 2.11: Query Q_1

Q_1

city	area	price	price per m²
Leipzig	64	525€	8.20€
München	72	1370€	19.03€
Tübingen	60	773€	12.88€
Hamburg	77	1350€	17.53€

Tabelle 2.7: Ergebnis von Q_1

Es ist offensichtlich, dass der Preis pro Quadratmeter von der jeweiligen Fläche und dem Gesamtpreis abhängt. Die Notation der *Where-Provenance* geschieht mithilfe von *Provenance-Sets*. In den Vorbereitungen wurde jeder Zelle in der `apartments` Tabelle eine Identifikation ϱ_i zugewiesen. Die Betrachtung der *Where-Provenance* von Q_1 geschieht ebenfalls in Form einer Tabelle. Dabei wird jeder Zelle eine Menge von Identifikationen ϱ_i, welche das entsprechende Ergebnis beeinflusst haben, zugewiesen.

$Q_1^{\mathbb{1}}$

city	area	price	price per m²
Leipzig	64	525€	8.20€
München	72	1370€	19.03€
Tübingen	60	773€	12.88€
Hamburg	77	1350€	17.53€

$Q_1^{\mathbb{2}}$

	city	area	price	price per m²
ϱ_1	$\{\varrho_5\}$	$\{\varrho_6\}$	$\{\varrho_7\}$	$\{\varrho_6, \varrho_7\}$
ϱ_2	$\{\varrho_8\}$	$\{\varrho_9\}$	$\{\varrho_{10}\}$	$\{\varrho_9, \varrho_{10}\}$
ϱ_3	$\{\varrho_{11}\}$	$\{\varrho_{12}\}$	$\{\varrho_{13}\}$	$\{\varrho_{12}, \varrho_{13}\}$
ϱ_4	$\{\varrho_{14}\}$	$\{\varrho_{15}\}$	$\{\varrho_{16}\}$	$\{\varrho_{15}, \varrho_{16}\}$

Tabelle 2.8: Ergebnis der Phasen $\mathbb{1}$ und $\mathbb{2}$ bei der Berechnung der *Where-Provenance*

2.3.3.2 Why-Provenance

Die *Why-Provenance* beschreibt, welche Eingabewerte die Existenz eines Wertes in der Ergebnistabelle beeinflusst haben. Hauptsächlich wird Why-Provenance durch Filterentscheidungen (`WHERE`) generiert, aber auch durch Gruppierungen (`GROUP BY`) und Window Functions. Zur Demonstration untersuchen wir die *Why-Provenance* einer Query Q_2, welche alle Wohnungen mit einem Preis pro Quadratmeter kleiner 15€ berechnet.

```
1 SELECT city          AS city,
2        area          AS area,
3        price         AS price
4 FROM   apartments
5 WHERE  price / area < 15
```

Listing 2.12: Query Q_2

Q_2

city	area	price
Leipzig	64	525€
Tübingen	60	773€

Tabelle 2.9: Ergebnis der Query Q_2

In dieser Query hängt die Existenz jeder Zeile in der Ergebnistabelle, von allen Spaltenreferenzen in der `WHERE`-Klausel ab. Konkret sind die Spalten `area` und `price` gemeint. Daher ist die Provenance für alle Spalten einer Zeile identisch.

$Q_2^{\mathbb{1}}$

city	area	price	
Leipzig	64	525€	ϱ_1
Tübingen	60	773€	ϱ_3

$Q_2^{\mathbb{2}}$

city	area	price
$\{\varrho_6, \varrho_7\}$	$\{\varrho_6, \varrho_7\}$	$\{\varrho_6, \varrho_7\}$
$\{\varrho_{12}, \varrho_{13}\}$	$\{\varrho_{12}, \varrho_{13}\}$	$\{\varrho_{12}, \varrho_{13}\}$

Tabelle 2.10: Ergebnis der Phasen $\mathbb{1}$ und $\mathbb{2}$ bei der Berechnung der *Why-Provenance*

2.4 Provenance Toolchain

In Kapitel 2.1 wurden die Abhängigkeiten von *SQLProv* vorgestellt. Abbildung 2.3 verschafft einen Überblick über das Zusammenspiel von *SQLProv* mit den Abhängigkeiten beim Übersetzungsvorgang einer SQL Query.

Startpunkt der Betrachtung ist die zu übersetzende SQL Query. Wie in Abschnitt 2.2.1 erläutert, akzeptiert *SQLProv* nur normalisierte Queries, um die Anzahl der benötigten Regeln so gering wie möglich zu halten. Erhält *SQLProv* eine normalisierte Query, wird diese an den *LogParser* übergeben, welcher diese Query mit Hilfe der Postgres Erweiterung analysiert und eine AST Repräsentation dieser Query zurückgibt. Den Kern von *SQLProv* stellt der Übersetzungsvorgang dieses ASTs in die Phasen $\mathbb{1}$ und $\mathbb{2}$ dar. Anschließend werden die beiden

ASTs mit Hilfe des *AST PrettyPrinters* in SQL Quelltext übersetzt und ausgegeben. Zusätzlich werden SQL Statements zur Übersetzung der verwendeten Tabellen in die Phasen 𝟙 und 𝟚 generiert.

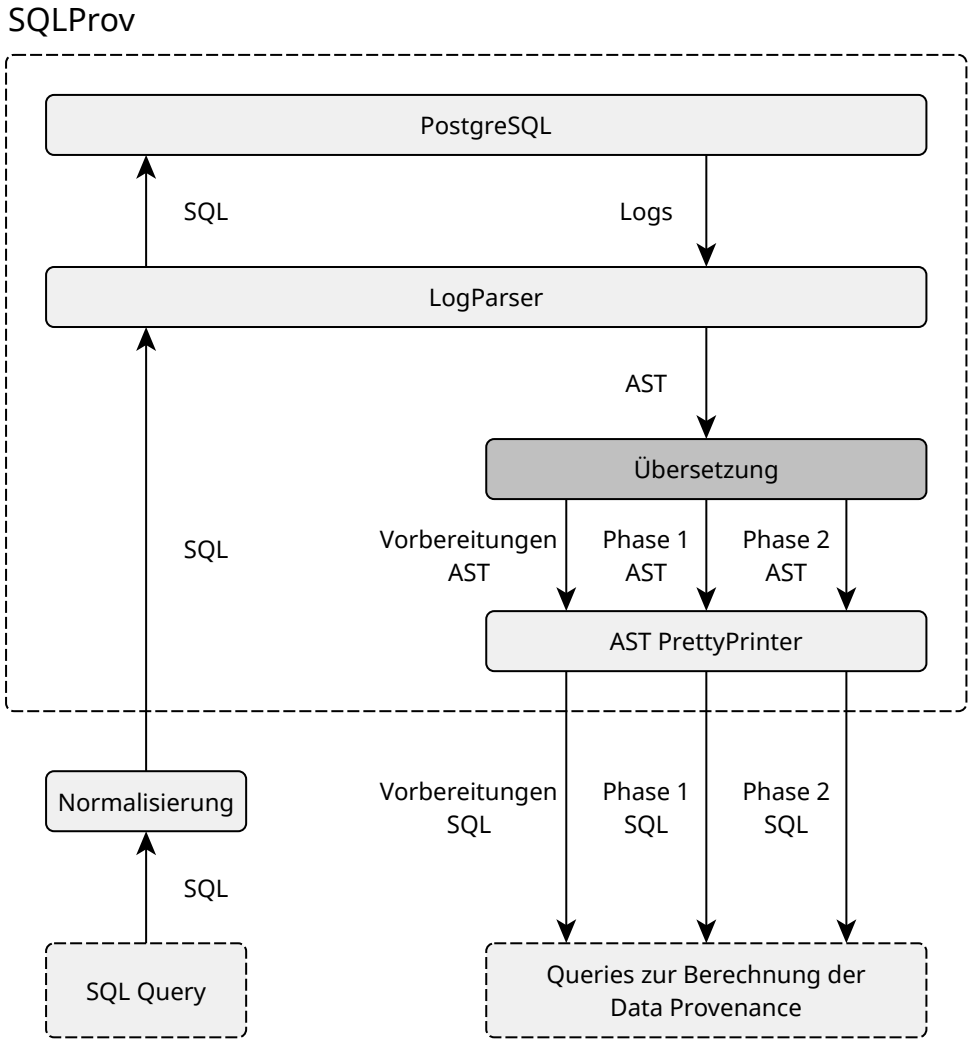

Abbildung 2.3: Einordnung von *SQLProv* in die SQL Provenance Toolchain

3 Implementierung

Dieses Kapitel beschäftigt sich mit dem internen Aufbau und der Funktionsweise von *SQLProv*. Zu Beginn wird eine Übersicht der internen Module vermittelt und die jeweiligen Aufgaben der Module erläutert. Anschließend werden einige interessante Implementierungsdetails hervorgehoben und näher beschrieben. Der letzte Abschnitt enthält das Regelwerk zur Übersetzung einer Query in die Phasen $\mathbb{1}$ und $\mathbb{2}$. Zudem wird die Implementation des Regelwerks anhand einer Transformationsregel näher beschrieben.

3.1 Aufbau

Haskell bietet die Möglichkeit ein Programm intern auf mehrere Module aufzuteilen, um eine Aufteilung in logische Teilprogramme zu erhalten. Davon wurde bei der Implementierung von *SQLProv* Gebrauch gemacht und ist in Abbildung 3.1 in Form eines Abhängigkeitsdiagramms dargestellt. Die Aufgaben der einzelnen Module werden im Folgenden näher erläutert.

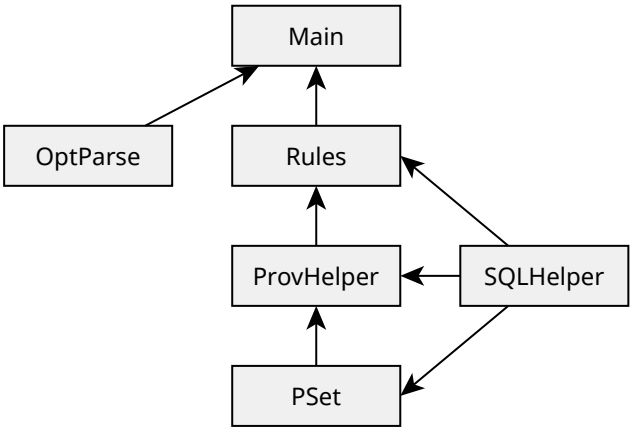

Abbildung 3.1: SQLProv Modulübersicht

- Das *Main* Modul beinhaltet das Hauptprogramm, welches die Programmargumente interpretiert und die gewünschten Aktionen ausführt.

- Das *OptParse* Modul ist für das Parsen der Programmargumente in eine interne Datenstruktur zuständig. Dafür wird die externe Programmbibliothek *optparse-applicative* verwendet.
- Das Modul *SQLHelper* enthält eine Sammlung von Funktionen, welche für das Arbeiten mit dem AST des *LogParsers* hilfreich sind.
- Das *PSet* Modul enthält die AST Definitionen für die Implementierung von *Provenance-Sets*. In Kapitel 3.3 wird näher auf dieses Modul eingegangen.
- Das *ProvHelper* Modul ist ähnlich wie *SQLHelper* dafür zuständig, das Arbeiten mit dem AST zu erleichtern. Dieses Modul enthält allerdings nur Funktionen für Provenance-bezogene AST-Operationen und benötigt daher die Abhängigkeit zum *PSet* Modul.
- Das *Rules* Modul bildet das Kernstück von *SQLProv* und enthält die Übersetzungsregeln. Die Regeln sind in Kapitel 3.4 beschrieben.

3.1.1 Bedienung

Eine Installationsanleitung und Beschreibung der verfügbaren Programmargumente ist im Repository unter `PgLlProvenance/SQLProv/README.md` zu finden.

3.2 Abstract Syntax Tree

Ein Abstract Syntax Tree (AST) ist eine für die Maschine optimierte Darstellung einer Programmiersprache. Für eine Sprache kann es, je nach Anwendungszweck, mehrere mögliche Repräsentationen geben. Der *LogParser* enthält bereits einen eigenen AST für SQL. Dieser enthält einige Informationen, welche für den Anwendungszweck von *SQLProv* irrelevant sind, wie zum Beispiel die Position eines AST Elements im ursprünglichen Quelltext.

Ein AST Element des *LogParsers* enthält oft viele Felder, sodass sich bei der Konstruktion eines solchen Elements die Bedeutung der einzelnen Argumente nicht sofort erschließt. Dieses Problem ist in Listing 3.1 und 3.2 veranschaulicht.

```
data Expr =
    -- ...
    | EAggRef
          { args        :: [Expr]
          , proname     :: String
          , aggorder    :: [SortEx]
          , aggdistinct :: [SortEx]
          , aggfilter   :: Maybe Expr
          , typ         :: Type
          , aggstar     :: Bool
          , agglevelsup :: Integer
          , location    :: Maybe Integer
          }
    -- ...
```

Listing 3.1: AST Definition eines `EAggRef`

```
concatAgg :: Expr -> Expr
concatAgg x =
    EAggRef [x]
            (pgProc "concat_agg")
            []
            []
            Nothing
            ([sqlTypeAnyArray] ~+> sqlTypeAnyArray)
            False
            0
            Nothing
```

Listing 3.2: Konstruktion eines `EAggRef`

Listing 3.3 präsentiert eine Lösung für die beiden oben genannten Probleme, welche in *SQLProv* verwendet wird. Im *SQLHelper* Modul ist zu jedem verwendeten AST Element ein dazugehöriger *Standardwert* mit dem Prefix `default` definiert. Damit lassen sich, wie in Listing 3.4, neue AST Elemente auf eine lesbare Art und Weise erstellen.

```
1  defaultAggRef :: A.Expr
2  defaultAggRef =
3      A.EAggRef []              -- args
4                ""              -- aggname
5                []              -- aggorder
6                []              -- aggdistinct
7                Nothing         -- aggfilter
8                sqlTypeDefault  -- typ
9                False           -- aggstar
10               0               -- agglevelsup
11               Nothing         -- location
```

Listing 3.3: Definition von `defaultEAggRef`

```
1  concatAgg :: Expr -> Expr
2  concatAgg x =
3      defaultEAggRef
4          { args    = [x]
5          , aggname = pgProc "concat_agg"
6          , typ     = [sqlTypeAnyArray] ~+> sqlTypeAnyArray
7          }
```

Listing 3.4: Konstruktion eines `EAggRef` mit `defaultEAggRef`

3.3 Provenance-Sets

Provenance-Sets beschreiben die Abhängigkeiten einer Zelle zu den Eingabedaten. Sie spielen in der Phase $\mathbb{2}$ Query eine zentrale Rolle. Während in Phase $\mathbb{1}$ die Abhängigkeiten protokolliert werden, werden in Phase $\mathbb{2}$ diese Protokolle ausgelesen und aggregiert. Optimalerweise verhalten sie sich dabei wie Mengen aus mathematischer Sicht (duplikatfrei). Leider bietet Postgres keinen solchen Mengen Datentyp, weshalb in der aktuellen Implementierung Arrays verwendet werden. Für die *Provenance-IDs* ϱ_i werden Ganzzahlen verwendet. Diese haben einen geringen Speicherverbrauch und lassen sich durch in Postgres eingebaute Sequenzen fortlaufend generieren.

Listing 3.5 zeigt die zentralen Funktionen des PSet Moduls, welche bei einer Änderung der Darstellung von *Provenance-Sets* geändert werden müssen. Die abstrakte Bedeutung der Funktionen im *PSet* Modul lautet wie folgt.

- `sqlTypeRho` ist der Typ einer *Provenance-ID*.
- `sqlTypePSet` ist der Typ eines *Provenance-Sets*.

- `empty` erzeugt ein leeres *Provenance-Set*.
- `union` erzeugt eine SQL Operation, welche zwei *Provenance-Sets* vereinigt.
- `toWhy` erzeugt eine SQL Operation, welche ein *Where-Provenance-Set* in ein *Why-Provenance-Set* umwandelt.
- `singleton` gibt ein einelementiges *Provenance-Set* zurück, welches die übergebene *Provenance-ID* enthält.
- `concatAgg` gibt eine Aggregatfunktion zurück, welche alle *Provenance-Sets* einer Gruppe vereinigt.
- `concatAggWin` verhält sich genau wie `concatAgg` und ist für den Einsatz in *window functions* gedacht. Im verwendeten AST wird zwischen Aggregatfunktionen in `GROUP BY` und `WINDOW` unterschieden.

In Kapitel 2.3 wurden *Where-* und *Why-Provenance* getrennt betrachtet. Da in der Praxis die *Why-Provenance* aus der *Where-Provenance* berechnet wird, ist es sinnvoll beide Provenance Typen gleichzeitig zu berechnen und in einer Menge darzustellen. Da bislang nur der positive Zahlenbereich für die *Where-Provenance* verwendet wird, liegt es nahe die negativen Zahlen für die *Why-Provenance* zu benutzen. Ein Rückschluss auf die ursprüngliche *Provenance-ID* ist jederzeit durch erneute Negierung möglich.

```
1  import AST as A
2  import SQLHelper as H
3
4  sqlTypeRho :: A.Type
5  sqlTypeRho = H.sqlTypeInt
6
7  sqlTypePSet :: A.Type
8  sqlTypePSet = H.sqlTypeIntArray
9
10 empty :: A.Expr
11 empty = H.emptyIntArray
12
13 singleton :: A.Expr -> A.Expr
14 singleton = H.intArray . pure
15
16 union :: A.Expr -> A.Expr -> A.Expr
17 union s1 s2 = H.defaultEOpexpr{ oprleft  = Just s1
18                               , oprright = s2
19                               , oprname  = H.pgOperator "||"
20                               , typ      = sqlTypePSet
21                               }
22
23 concatAgg :: A.Expr -> A.Expr
24 concatAgg pset = H.defaultEAggRef{ args    = [pset]
25                                  , proname = pgProc "concat_agg"
26                                  , typ     = [sqlTypePSet] ~+> sqlTypePSet
27                                  }
28
29 concatAggWin :: Integer -> A.Expr -> A.Expr
30 concatAggWin winId pset =
31     H.defaultEWinFunc{ args    = [pset]
32                      , proname = pgProc "concat_agg"
33                      , winref  = winId
34                      , typ     = [sqlTypePSet] ~+> sqlTypePSet
35                      }
36
37 toWhy :: A.Expr -> A.Expr
38 toWhy pset = H.defaultEFuncCall{ args    = [pset]
39                                , proname = H.pgProc "toY"
40                                , typ     = [sqlTypePSet] ~+> sqlTypePSet
41                                }
```

Listing 3.5: Funktionen des *PSet* Moduls

3.4 Regelwerk

Das *Rules* Modul, welche das Regelwerk enthält, um eine Query in die Phasen $\mathbb{1}$ und $\mathbb{2}$ zu übersetzen, bildet das Herzstück von *SQLProv*. Der AST des *LogParsers* besitzt zwei zentrale Typen `Query` und `Expr`. `Query` repräsentiert eine `SELECT` Query und enthält Informationen über die Belegung der einzelnen SQL Konstrukte (`WHERE`, `GROUP BY`, . . .). Der Summentyp `Expr` repräsentiert einen Ausdruck und umfasst verschiedenste SQL Konstrukte. Diese reichen von einem einfachen Literal oder Spaltenverweis bis hin zu einer Subquery oder Aggregierung. Für jeweils beide dieser Typen wird eine Übersetzungfunktion benötigt.

```
1  data RuleConfig = RuleConfig { runYProv :: Bool }
2
3  data TableHead  = TableHead String  -- ^ name of table
4                              Type    -- ^ contains column names and types
5
6  --            input           state     env    out    log
7  --              |            ___|___ ____|_____ | _____|_____
8  type Rule a b = a -> OperSem Integer RuleConfig b [TableHead]
9
10 --             Input Phase 1 Phase 2
11 --             __|__  __|__   __|__
12 (~~>)  :: Rule Query (Query,  Query)
13 (~~~>) :: Rule Expr  (Expr ,  Expr )
```

Listing 3.6: Signatur der Übersetzungsfunktionen

Listing 3.6 zeigt die zentralen Definitionen und Signaturen des Regelwerks. Die Datenstruktur *RuleConfig* enthält die Programmoptionen, welche für die Übersetzung wichtig sind. Momentan existiert nur eine Option, um die Generierung der *Why-Provenance* in Phase $\mathbb{2}$ ein- bzw. auszuschalten. *TableHead* repräsentiert den *Kopf* einer Tabelle. Dazu gehören Name der Tabelle und die Liste der Namen und Datentypen der Spalten. Beide Übersetzungsfunktionen ~~> und ~~~> benutzen den *OperSem* Monad, welche eine *operationale Semantik* beschreibt. Dieser Monad besitzt ein *Environment*, einen *Log* und einen *State*, welche wie folgt belegt sind.

- Das *Environment* besteht aus der *RuleConfig*, damit diese innerhalb der Übersetzungsfunktionen gelesen werden kann.
- Der *State* besteht aus einer Ganzzahl, welche mit 1 initialisiert wird. Im Regelwerk wird ein Generator von Seiten des Programms benötigt, welcher eindeutige Zahlen liefert. Die Generatorfunktion lautet im Regelwerk *generateId()*.

- Das *Log* besteht aus einer Liste von *TableHeads*. Stößt die Übersetzungsfunktion auf einen Tabellenverweis, wird dieser protokolliert. Dies ist nötig, damit die Tabellen ebenfalls in Phase $\mathbb{1}$ und $\mathbb{2}$ übersetzt werden, da dies nicht Teil der eigentlichen Querytransformation ist. In Abschnitt 3.4.1 wird der Übersetzungsvorgang der Eingabetabellen beschrieben.

Listing 3.7 zeigt den grundlegenden Aufbau einer Übersetzungsfunktion, welche die JOIN-Regel[1] implementiert. Die Übersetzungsfunktion `~~>` akzeptiert ein Argument vom Typ `Query` und nutzt Pattern Matching, um zwischen den verschiedenen Typen einer Query zu unterscheiden. Da der Produkttyp `Query` mehr als 10 Felder besitzt, wurde die GHC Erweiterung *PatternSynonyms* verwendet, um die Übersichtlichkeit zu bewahren. Damit lassen sich Patterns an selbstdefinierte Namen binden und die Anzahl der Pattern-Argumente auf das Wesentliche beschränken. In der gezeigten Übersetzungsfunktion lautet das Pattern `PSelectFromNWhere` und bindet die gewünschten Ausdrücke an die Namen `selects`, `froms` und `whereEx`.

Im Funktionskörper werden zu Beginn die einzelnen Ausdrücke der Query in Phase $\mathbb{1}$ und $\mathbb{2}$ übersetzt und eine ID generiert. Anschließend wird für jede Tabelle t in der `FROM`-Klausel die Spaltenreferenz auf $t.\varrho$ in `fromRhoCols` gespeichert. Die Zeilen 13 - 15 sind für die Generierung der `writeJoin` und `readJoin` Aufrufe mit den benötigten Argumenten verantwortlich. In Anschluss werden in den Zeilen 17 - 23 die Berechnungen für die *Why-Provenance* durchgeführt. Die monadische Funktion `ifYProv` sorgt dafür, dass diese nur berechnet werden, wenn die dafür vorgesehene Option in `RuleConfig` gesetzt ist. Am Ende der Übersetzungsfunktion werden die ASTs der Phasen $\mathbb{1}$ und $\mathbb{2}$ mit Hilfe der zuvor berechneten Werte zusammengesetzt.

[1] Die JOIN-Regel ist in Abschnitt 3.4.3.2 definiert.

```
(~~>) inp@(PSelectFromNWhere selects froms whereEx) = do
    -- translate expressions in SELECT, FROM, WHERE
    (selects1, selects2) <- mapAndUnzipM (~~~>) selects
    (froms1, froms2)     <- mapAndUnzipM (~~~>) froms
    (where1, where2)     <- (~~~>) whereEx

    -- generate ID for readJoin/writeJoin
    l <- generateId

    -- extract rho colum refs
    let fromRhoCols = extractRhoColRef <$> froms

    -- generate readJoin/writeJoin
    let writeJoin = H.writeJoin l fromRhoCols
    let readJoin  = H.readJoin l fromRhoCols "join" [rhoColName]

    -- generate Y provenance table
    let yProv  = H.yProvTable where2 "y_join" "y_join"
    yProv'    <- ifYProv [yProv] []

    -- union every expression in phase 2 SELECT with Y provenance set
    let yCol   = H.genColRef' "y_join" "y_join"
    selects2' <- ifYProv (H.tmap (P.union yCol) <$> selects2) selects2

    return ( inp{ select  = writeJoin : selects1
                , from    = froms1
                , whereEx = Just where1
                }
           , inp{ select  = genRhoColRefTargetEx "join" : selects2'
                , from    = froms2 ++ [readJoin] ++ yProv'
                , whereEx = Nothing
                })
```

Listing 3.7: Implementierung der JOIN Übersetzungsregel

In Abschnitt 2.3.1 wurde die Übersetzung der Eingabetabellen in die Phasen $\mathbb{1}$ und $\mathbb{2}$ beschrieben. Die Implementierung dieser Vorbereitungen wird im folgenden Abschnitt behandelt. Anschließend wird ein Überblick über alle Transformationsregeln verschafft, welche in *SQLProv* implementiert sind.

3.4.1 Vorbereitung

Dieser Abschnitt beschreibt die benötigten SQL Statements zur Überführung der Eingabetabellen in die Phasen $\mathbb{1}$ und $\mathbb{2}$. In Kapitel 2.3.1 wurde die Vorbereitung der Eingabetabellen bereits im Kontext der *Data Provenance* erläutert. Die in diesem Abschnitt beschriebenen SQL Statements zur Vorbereitung der Eingabetabellen werden bei der Übersetzung einer Query mittels *SQLProv* mitgeneriert.

Ziel der Vorbereitung ist es, die Zellen der Eingabetabellen mit eindeutigen *Provenance-IDs* durchzunummerieren. Dafür bieten sich die in *Postgres* eingebauten *Sequenzen* an. Da jede *Provenance-ID* über alle verwendeten Tabellen eindeutig sein muss, geschieht die Erstellung der Sequenz nur einmal, unabhängig von der Anzahl der zu übersetzenden Tabellen.

```
CREATE SEQUENCE prov_id
```

Listing 3.8: Erstellung der Sequenz für die Generierung der *Provenance-IDs*

Für die Tabellen der Phase $\mathbb{1}$ und $\mathbb{2}$ werden *materialisierte Sichten* verwendet. Diese können bei einer Änderung der zugrundeliegenden Tabellen mit einem SQL Statement aktualisiert werden.

Zur Überführung einer Tabelle t mit den Spalten $c_1, \ldots, c_n$ und den Typen $\tau_1, \ldots, \tau_n$ in die Tabellen $t^{\mathbb{1}}$ und $t^{\mathbb{2}}$ werden die in Listing 3.9 und 3.10 gezeigten Statements generiert.

```
CREATE MATERIALIZED VIEW t^1 AS
(SELECT nextval('prov_id') AS ϱ,
        t.c_1 AS c_1,
        ...,
        t.c_n AS c_n
 FROM t)
```

Listing 3.9: CREATE Statement zur Erstellung von $t^{\mathbb{1}}$

```
CREATE MATERIALIZED VIEW t^2 AS
(SELECT t^1.ϱ AS ϱ,
        {nextval('prov_id')} AS c_1,
        ...,
        {nextval('prov_id')} AS c_n
 FROM t^1)
```

Listing 3.10: CREATE Statement zur Erstellung von $t^{\mathbb{2}}$

Die Notation $\{x\}$ in Listing 3.10 entspricht einem *Provenance-Set* mit einem Element x. Die Implementierung dieser Notation ist im *PSet* Modul unter dem Namen `singleton` zu finden. In der aktuellen Implementierung wird hierfür ein einelementiges Array verwendet (`ARRAY[x]`).

3.4.2 Regelwerk Expr

Dieser Abschnitt enthält die implementierten Übersetzungsregeln der Funktion ~~~>, welche ein AST Element vom Typ `Expr` in die Phasen $\mathbb{1}$ und $\mathbb{2}$ übersetzt. Die Notation der *Provenance-Sets* und der darauf basierenden Operationen geschieht abstrakt und sind im *PSet* Modul implementiert. Da die Darstellung von *Provenance-Sets* im *Rules* Modul abstrakt gehalten wurde.

3.4.2.1 Literal

$$\frac{}{l \rightmodels \langle l, \emptyset \rangle}$$

3.4.2.2 Spaltenverweis

$$\frac{}{t.c \rightmodels \langle t.c, t.c \rangle}$$

3.4.2.3 Tabellenverweis

$$\frac{}{t \rightmodels \langle t^{\mathbb{1}}, t^{\mathbb{2}} \rangle}$$

3.4.2.4 Operator

$$\frac{\oplus \in \{\cdot + \cdot, \cdot \ \mathbf{AND} \ \cdot, \cdot < \cdot, \dots\} \qquad \left| e_i \rightmodels \langle e_i^{\mathbb{1}}, e_i^{\mathbb{2}} \rangle \right|_{i=1,\dots,n}}{\oplus(e_1, \dots, e_n) \rightmodels \langle \oplus(e_1^{\mathbb{1}}, \dots, e_n^{\mathbb{1}}), e_1^{\mathbb{2}} \cup \dots \cup e_n^{\mathbb{2}} \rangle}$$

3.4.2.5 Tabellenwertige Funktion

$$f :: \tau_1 \times \cdots \times \tau_n \rightarrow \textbf{TABLE(}c_1\ \tau'_1, \ldots, c_m\ \tau'_m\textbf{)}$$

$$\left| e_i \rightarrowtail \langle e_i^{\mathbb{1}}, e_i^{\mathbb{2}} \rangle \right|_{i=1,\ldots,n} \qquad \lambda = \mathit{generateId()}$$

$$i^{\mathbb{1}} = \begin{array}{l} \texttt{(}\textbf{SELECT}\ \texttt{writeTblf(}\lambda\texttt{)}\ \textbf{AS}\ \varrho,\ t\texttt{.}c_1\ \textbf{AS}\ c_1\texttt{,}\ \ldots\texttt{,}t\texttt{.}c_m\ \textbf{AS}\ c_m \\ \textbf{FROM}\quad f\texttt{(}e_1^{\mathbb{1}}, \ldots, e_n^{\mathbb{1}}\texttt{)}\ \textbf{AS}\ t\texttt{(}c_1, \ldots, c_m\texttt{))} \end{array}$$

$$i^{\mathbb{2}} = \begin{array}{l} \texttt{(}\textbf{SELECT}\ \texttt{set.}\varrho\ \textbf{AS}\ \varrho,\ \texttt{args.D}\ \textbf{AS}\ c_1\texttt{,}\ \ldots\texttt{,args.D}\ \textbf{AS}\ c_m \\ \textbf{FROM}\quad \texttt{(}\textbf{VALUES}\ \texttt{(}e_1^{\mathbb{2}} \cup \cdots \cup e_n^{\mathbb{2}}\texttt{))}\ \textbf{AS}\ \texttt{args(D),} \\ \qquad\quad \texttt{readTblf(}\lambda\texttt{)}\ \textbf{AS}\ \texttt{set(}\varrho\texttt{))} \end{array}$$

$$\frac{}{f\texttt{(}e_1, \ldots, e_n\texttt{)} \rightarrowtail \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle}$$

3.4.3 Regelwerk Query

Dieser Abschnitt enthält die implementierten Übersetzungsregeln der Funktion ~~>, welche ein AST Element vom Typ `Query` in die Phasen $\mathbb{1}$ und $\mathbb{2}$ übersetzt. Der Teil einer Query, welcher die *Why-Provenance* berechnet, kann durch eine Programmoption deaktiviert werden und ist durch eine graue Schriftfarbe hervorgehoben.

3.4.3.1 Select

$$\left| e_i \rightarrowtail \langle e_i^{\mathbb{1}}, e_i^{\mathbb{2}} \rangle \right|_{i=1,\ldots,n}$$

$$i^{\mathbb{1}} = \texttt{SELECT } 1 \texttt{ AS } \varrho, e_1^{\mathbb{1}} \texttt{ AS } c_1, \ldots, e_n^{\mathbb{1}} \texttt{ AS } c_n$$

$$\frac{i^{\mathbb{2}} = \texttt{SELECT } 1 \texttt{ AS } \varrho, e_1^{\mathbb{2}} \texttt{ AS } c_1, \ldots, e_n^{\mathbb{2}} \texttt{ AS } c_n}{\texttt{SELECT } e_1 \texttt{ AS } c_1, \ldots, e_n \texttt{ AS } c_n \rightarrowtail \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle}$$

3.4.3.2 Join

$$\left| e_i \rightarrowtail \langle e_i^{\mathbb{1}}, e_i^{\mathbb{2}} \rangle \right|_{i=1,\ldots,n}$$

$$\left| q_i \rightarrowtail \langle q_i^{\mathbb{1}}, q_i^{\mathbb{2}} \rangle \right|_{i=1,\ldots,m} \qquad p \rightarrowtail \langle p^{\mathbb{1}}, p^{\mathbb{2}} \rangle \qquad \lambda = \mathit{generateId()}$$

$$i^{\mathbb{1}} = \begin{array}{l} \texttt{SELECT writeJoin}(\lambda, t_1.\varrho, \ldots, t_m.\varrho) \texttt{ AS } \varrho, e_1^{\mathbb{1}} \texttt{ AS } c_1, \ldots, e_n^{\mathbb{1}} \texttt{ AS } c_n \\ \texttt{FROM } q_1^{\mathbb{1}} \texttt{ AS } t_1, \ldots, [\texttt{LATERAL}]\ q_m^{\mathbb{1}} \texttt{ AS } t_m \\ \texttt{WHERE } p^{\mathbb{1}} \end{array}$$

$$i^{\mathbb{2}} = \begin{array}{l} \texttt{SELECT join.}\varrho \texttt{ AS } \varrho, e_1^{\mathbb{2}} \cup \texttt{y.}\upsilon \texttt{ AS } c_1, \ldots, e_n^{\mathbb{2}} \cup \texttt{y.}\upsilon \texttt{ AS } c_n \\ \texttt{FROM } q_1^{\mathbb{2}} \texttt{ AS } t_1, \ldots, [\texttt{LATERAL}]\ q_m^{\mathbb{2}} \texttt{ AS } t_m, \\ \quad \texttt{readJoin}(\lambda, t_1.\varrho, \ldots, t_m.\varrho) \texttt{ AS join}(\varrho), \\ \quad \texttt{toY}(p^{\mathbb{2}}) \texttt{ AS y}(\upsilon) \end{array}$$

$$\frac{}{\begin{array}{l} \texttt{SELECT } e_1 \texttt{ AS } c_1, \ldots, e_n \texttt{ AS } c_n \\ \texttt{FROM } q_1 \texttt{ AS } t_1, \ldots, [\texttt{LATERAL}]\ q_m \texttt{ AS } t_m \rightarrowtail \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle \\ \texttt{WHERE } p \end{array}}$$

3.4.3.3 Group By

$$\frac{e \rightrightarrows \langle e^{\mathbb{1}}, e^{\mathbb{2}} \rangle}{AGG(e) \rightrightarrows \langle AGG(e^{\mathbb{1}}), \bigcup e^{\mathbb{2}} \rangle} \qquad \frac{}{AGG(*) \rightrightarrows \langle AGG(*), \emptyset \rangle}$$

$$\left| e_i \rightrightarrows \langle e_i^{\mathbb{1}}, e_i^{\mathbb{2}} \rangle \right|_{i=1,\ldots,n}$$

$$q \rightrightarrows \langle q^{\mathbb{1}}, q^{\mathbb{2}} \rangle \qquad \left| g_i \rightrightarrows \langle g_i^{\mathbb{1}}, g_i^{\mathbb{2}} \rangle \right|_{i=1,\ldots,m} \qquad \lambda = generateId()$$

$$i^{\mathbb{1}} = \begin{array}{l} \textbf{SELECT} \quad \texttt{writeAgg}(\lambda, \bigcup\{t.\varrho\}) \textbf{ AS } \varrho, e_1^{\mathbb{1}} \textbf{ AS } c_1, \ldots, e_n^{\mathbb{1}} \textbf{ AS } c_n \\ \textbf{FROM} \quad q^{\mathbb{1}} \textbf{ AS } t \\ \textbf{GROUP BY } g_1^{\mathbb{1}}, \ldots, g_m^{\mathbb{1}} \end{array}$$

$$i^{\mathbb{2}} = \begin{array}{l} \textbf{SELECT} \quad \texttt{group.}\varrho \textbf{ AS } \varrho, e_1^{\mathbb{2}} \cup \bigcup\{\texttt{y.}\upsilon\} \textbf{ AS } c_1, \ldots, e_n^{\mathbb{2}} \cup \bigcup\{\texttt{y.}\upsilon\} \textbf{ AS } c_n \\ \textbf{FROM} \quad q^{\mathbb{2}} \textbf{ AS } t, \\ \qquad \texttt{readAgg}(\lambda, \texttt{ t.}\varrho) \textbf{ AS } \texttt{group}(\varrho), \\ \qquad \texttt{toY}(g_1^{\mathbb{2}} \cup \cdots \cup g_m^{\mathbb{2}}) \textbf{ AS } \texttt{y}(\upsilon) \\ \textbf{GROUP BY } \texttt{group.}\varrho \end{array}$$

$$\begin{array}{l} \textbf{SELECT} \quad e_1 \textbf{ AS } c_1, \ldots, e_n \textbf{ AS } c_n \\ \textbf{FROM} \quad q \textbf{ AS } t \qquad\qquad \rightrightarrows \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle \\ \textbf{GROUP BY } g_1, \ldots, g_m \end{array}$$

3.4.3.4 Select From

Diese Regel ist genau genommen ein Spezialfall der JOIN Regel. Trotzdem wurde diese Regel in *SQLProv* implementiert, um überflüssige Aufrufe der `writeJoin` und `readJoin` UDFs zu vermeiden.

$$\left| e_i \rightarrowtriangle \langle e_i^{\mathbb{1}}, e_i^2 \rangle \right|_{i=1,\ldots,n} \qquad q \rightarrowtriangle \langle q^{\mathbb{1}}, q^2 \rangle$$

$$i^{\mathbb{1}} = \begin{array}{l} \textbf{SELECT}\ t.\varrho\ \textbf{AS}\ \varrho,\ e_1^{\mathbb{1}}\ \textbf{AS}\ c_1, \ldots, e_n^{\mathbb{1}}\ \textbf{AS}\ c_n \\ \textbf{FROM}\quad q^{\mathbb{1}}\ \textbf{AS}\ t \end{array}$$

$$i^2 = \begin{array}{l} \textbf{SELECT}\ t.\varrho\ \textbf{AS}\ \varrho,\ e_1^2\ \textbf{AS}\ c_1, \ldots, e_n^2\ \textbf{AS}\ c_n \\ \textbf{FROM}\quad q^2\ \textbf{AS}\ t \end{array}$$

$$\frac{}{\begin{array}{l} \textbf{SELECT}\ e_1\ \textbf{AS}\ c_1, \ldots, e_n\ \textbf{AS}\ c_n \\ \textbf{FROM}\quad q\ \textbf{AS}\ t \end{array} \rightarrowtriangle \langle i^{\mathbb{1}}, i^2 \rangle}$$

3.4.3.5 Order By

$$\left| e_i \rightarrowtriangle \langle e_i^{\mathbb{1}}, e_i^2 \rangle \right|_{i=1,\ldots,n}$$

$$q \rightarrowtriangle \langle q^{\mathbb{1}}, q^2 \rangle \qquad \left| o_i \rightarrowtriangle \langle o_i^{\mathbb{1}}, o_i^2 \rangle \right|_{i=1,\ldots,m} \qquad \lambda = generateId()$$

$$i^{\mathbb{1}} = \begin{array}{l} \textbf{SELECT}\quad \texttt{writeOrderBy}(\lambda,\ t.\varrho,\ \texttt{ROW\_NUMBER() OVER (}w\texttt{)})\ \textbf{AS}\ \varrho, \\ \qquad\quad e_1^{\mathbb{1}}\ \textbf{AS}\ c_1, \ldots, e_n^{\mathbb{1}}\ \textbf{AS}\ c_n \\ \textbf{FROM}\quad q^{\mathbb{1}}\ \textbf{AS}\ t \\ \textbf{WINDOW}\quad w\ \textbf{AS}\ \texttt{(ORDER BY (}o_1^{\mathbb{1}} \cup \cdots \cup o_m^{\mathbb{1}}\texttt{))} \\ \textbf{ORDER BY}\ o_1^{\mathbb{1}}, \ldots, o_m^{\mathbb{1}} \\ \textbf{LIMIT}\quad l \end{array}$$

$$i^2 = \begin{array}{l} \textbf{SELECT}\ \texttt{orderBy.}\varrho\ \textbf{AS}\ \varrho,\ e_1^2 \cup \bigcup\{\texttt{y.}\upsilon\}\ \textbf{AS}\ c_1, \ldots, e_n^2 \cup \bigcup\{\texttt{y.}\upsilon\}\ \textbf{AS}\ c_n \\ \textbf{FROM}\quad q^2\ \textbf{AS}\ t, \\ \qquad\quad \texttt{readOrderBy(}\lambda,\ \texttt{t.}\varrho\texttt{)}\ \textbf{AS}\ \texttt{orderBy(}\varrho\texttt{)}, \\ \qquad\quad \texttt{toY(}o_1^2 \cup \cdots \cup o_m^2\texttt{)}\ \textbf{AS}\ \texttt{y(}\upsilon\texttt{)} \end{array}$$

$$\frac{}{\begin{array}{l} \textbf{SELECT}\quad e_1\ \textbf{AS}\ c_1, \ldots, e_n\ \textbf{AS}\ c_n \\ \textbf{FROM}\quad q\ \textbf{AS}\ t \\ \textbf{ORDER BY}\ o_1, \ldots, o_m \\ \textbf{LIMIT}\quad l \end{array} \rightarrowtriangle \langle i^{\mathbb{1}}, i^2 \rangle}$$

3.4.3.6 Window

In dieser WINDOW-Regel darf in der SELECT-Klausel nur eine Aggregatfunktion vorkommen.

$$\frac{\mathit{AGG}(e^{1}) \Rightarrow \langle a^{1}, a^{2}\rangle \qquad Y = \bigcup \texttt{y}.\upsilon\ \texttt{OVER}\ (\omega\ \phi)}{\mathit{AGG}(e)\ \texttt{OVER}\ (\omega\ \phi) \Rightarrow \langle a^{1}\ \texttt{OVER}\ (\omega\ \phi), a^{2}\ \texttt{OVER}\ (\omega\ \phi)\ \cup\ Y\rangle} \qquad \frac{}{\mathit{AGG}(*) \Rightarrow \langle \mathit{AGG}(*), \emptyset\rangle}$$

$$\left| e_i \Rightarrow \langle e_i^{1}, e_i^{2}\rangle \right|_{i=1,\dots,n} \qquad q \Rightarrow \langle q^{1}, q^{2}\rangle$$

$$\left| g_i \Rightarrow \langle g_i^{1}, g_i^{2}\rangle \right|_{i=1,\dots,m} \qquad \left| o_i \Rightarrow \langle o_i^{1}, o_i^{2}\rangle \right|_{i=1,\dots,k} \qquad \lambda = generateId()$$

$$i^{1} = \begin{array}{l} \texttt{SELECT writeWin}(\lambda,\ t.\varrho,\ \texttt{FIRST\_VALUE}(t.\varrho)\ \texttt{OVER}\ (w), \\ \qquad\qquad\qquad \texttt{RANK() OVER}\ (w))\ \texttt{AS}\ \varrho, \\ \qquad e_1^{1}\ \texttt{AS}\ c_1, \dots, e_n^{1}\ \texttt{AS}\ c_n \\ \texttt{FROM}\quad q^{1}\ \texttt{AS}\ t \\ \texttt{WINDOW}\ w\ \texttt{AS}\ (\texttt{PARTITION BY}\ g_1^{1}, \dots, g_m^{1} \\ \qquad\qquad \texttt{ORDER BY}\quad o_1^{1}\ dir_1, \dots, o_k^{1}\ dir_k) \end{array}$$

$$i^{2} = \begin{array}{l} \texttt{SELECT win}.\varrho\ \texttt{AS}\ \varrho,\ e_1^{2}\ \texttt{AS}\ c_1, \dots, e_n^{2}\ \texttt{AS}\ c_n \\ \texttt{FROM}\quad q^{2}\ \texttt{AS}\ t, \\ \qquad \texttt{readWin}(\lambda,\ t.\varrho)\ \texttt{AS win}(\varrho), \\ \qquad \texttt{toY}(g_1^{2} \cup \dots \cup g_m^{2} \cup o_1^{2} \cup \dots \cup o_k^{2})\ \texttt{AS y}(\upsilon) \\ \texttt{WINDOW}\ w\ \texttt{AS}\ (\texttt{PARTITION BY win.part} \\ \qquad\qquad \texttt{ORDER BY}\quad \texttt{win.rank}) \end{array}$$

$$\overline{\begin{array}{l} \texttt{SELECT}\ e_1\ \texttt{AS}\ c_1, \dots, e_n\ \texttt{AS}\ c_n \\ \texttt{FROM}\quad q\ \texttt{AS}\ t \\ \texttt{WINDOW}\ w\ \texttt{AS}\ (\texttt{PARTITION BY}\ g_1, \dots, g_m \\ \qquad\qquad \texttt{ORDER BY}\quad o_1\ dir_1, \dots, o_k\ dir_k) \end{array} \Rightarrow \langle i^{1}, i^{2}\rangle}$$

3.4.3.7 With

$$\left| q_i \Longmapsto \langle q_i^{\mathbb{1}}, q_i^{\mathbb{2}} \rangle \right|_{i=0,\ldots,n}$$

$$i^{\mathbb{1}} = \begin{array}{l} \textbf{WITH [RECURSIVE]}\ t_1\ (\varrho, c_{11}, \ldots, c_{1k_1})\ \textbf{AS}\ (q_1^{\mathbb{1}}),\ \ldots, \\ \qquad\qquad t_n\ (\varrho, c_{n1}, \ldots, c_{nk_n})\ \textbf{AS}\ (q_n^{\mathbb{1}}) \\ \qquad\qquad q_0^{\mathbb{1}} \end{array}$$

$$i^{\mathbb{2}} = \begin{array}{l} \textbf{WITH [RECURSIVE]}\ t_1\ (\varrho, c_{11}, \ldots, c_{1k_1})\ \textbf{AS}\ (q_1^{\mathbb{2}}),\ \ldots, \\ \qquad\qquad t_n\ (\varrho, c_{n1}, \ldots, c_{nk_n})\ \textbf{AS}\ (q_n^{\mathbb{2}}) \\ \qquad\qquad q_0^{\mathbb{2}} \end{array}$$

$$\begin{array}{l} \textbf{WITH [RECURSIVE]}\ t_1\ (c_{11}, \ldots, c_{1k_1})\ \textbf{AS}\ (q_1),\ \ldots, \\ \qquad\qquad t_n\ (c_{n1}, \ldots, c_{nk_n})\ \textbf{AS}\ (q_n) \qquad\qquad \Longmapsto \langle i^{\mathbb{1}}, i^{\mathbb{2}} \rangle \\ \qquad\qquad q_0 \end{array}$$

4 Performance Analyse

In diesem Kapitel werden die aus dem *SQLProv* resultierenden Queries einer Performance Analyse unterzogen. Dazu wird TPC-H verwendet, welches ein beliebter Benchmark zum Vergleich von Datenbanksystemen ist.

4.1 Transaction Processing Performance Council

Das Transaction Processing Performance Council (TPC) ist ein *Non-Profit-Konsortium* mit dem Ziel Benchmarks für verschiedene Datenbankoperationen zur Verfügung zu stellen. Beispiele sind TPC-C (*Transaction Processing*), TPC-H (*Decision Support*) oder TPCx-BB (*Big Data*). Die hier durchgeführte Performance Analyse wurde mit Hilfe des TPC-H Benchmarks durchgeführt, welcher 8 Tabellen und 22 Queries umfasst. Die Datensätze für die verwendeten Tabellen existieren in unterschiedlichen Größenordnungen (10MB, 100MB, 1GB, . . .). Ziel der 22 Queries ist die Simulation von Berechnungen, welche unternehmensrelevante Entscheidungen unterstützen. Sie lesen oft große Datenmengen und besitzen eine hohe Komplexität, was eine lange Ausführungszeit zur Folge hat.

4.2 Durchführung

Für die hier durchgeführte Performance Analyse wurde ein Datensatz der Größe 100MB verwendet. Einige TPC-H Queries besitzen SQL Konstrukte, welche von *SQLProv* nicht unterstützt werden, weshalb nur 12 der 22 Queries gemessen werden können.

Im ersten Schritt der Durchführung werden die 12 TPC-H Queries von Hand normalisiert. Die Normalisierung geschieht dabei wie in Abschnitt 2.2.1 beschrieben. Im nächsten Schritt wird die normalisierte Query von *SQLProv* in Queries der Phasen $\mathbb{1}$, $\mathbb{2}$ ohne *Why-Provenance* und $\mathbb{2}$ mit *Why-Provenance* übersetzt. Die Ausführungszeit jeder Query wird 5 Mal gemessen und anschließend vom Minimum und Maximum bereinigt. Aus den restlichen Ausführungzeiten wird der Mittelwert gebildet. In Abbildung 4.1 wird die Ausführungszeit in Relation zur nicht-normalisierten TPC-H Query dargestellt.

Die Messung wurde auf einem Heimrechner unter Linux 4.15 mit einem Intel Core i7 6700K Prozessor und einer SSD durchgeführt. PostgreSQL wurde in der Version 10 verwendet mit den Einstellungen `shared_buffers = 4GB` und `work_mem = 2GB`.

Die Normalisierung einer Query hat kaum eine Veränderung der Ausführungszeit zur Folge, obwohl der Quelltext einer SQL Query dadurch um ein vielfaches wachsen kann. Dieses Ergebnis war zu erwarten, da die Semantik einer Query durch die Normalisierung nicht verändert wird.

In Phase $\mathbb{1}$ wird die normalisierte Query mit UDFs annotiert, welche das Verhalten der Query protokollieren. Durch Schreiben dieser Daten in externe Tabellen entsteht ein natürlicher *Overhead*, welcher klar im Graph erkennbar ist. Dieser hängt von der Größe der verarbeiteten Tabellen und der Anzahl der geschriebenen Logs ab.

In Phase $\mathbb{2}$ werden die in Phase $\mathbb{1}$ protokollierten Daten gelesen und *Provenance-Sets* aggregiert. Die Ausführungszeit wird maßgeblich durch die Implementierung der *Provenance-Sets* beeinflusst. Durch die Verwendung einer optimierten Implementierung können hier bessere Ausführungszeiten erreicht werden. Auch die zusätzliche Berechnung der *Why-Provenance* hat einen negativen Einfluss auf die Ausführungszeit, da hier zusätzliche Daten aggregiert werden und *Provenance-Sets* mittels der `toY`-UDF konvertiert werden müssen.

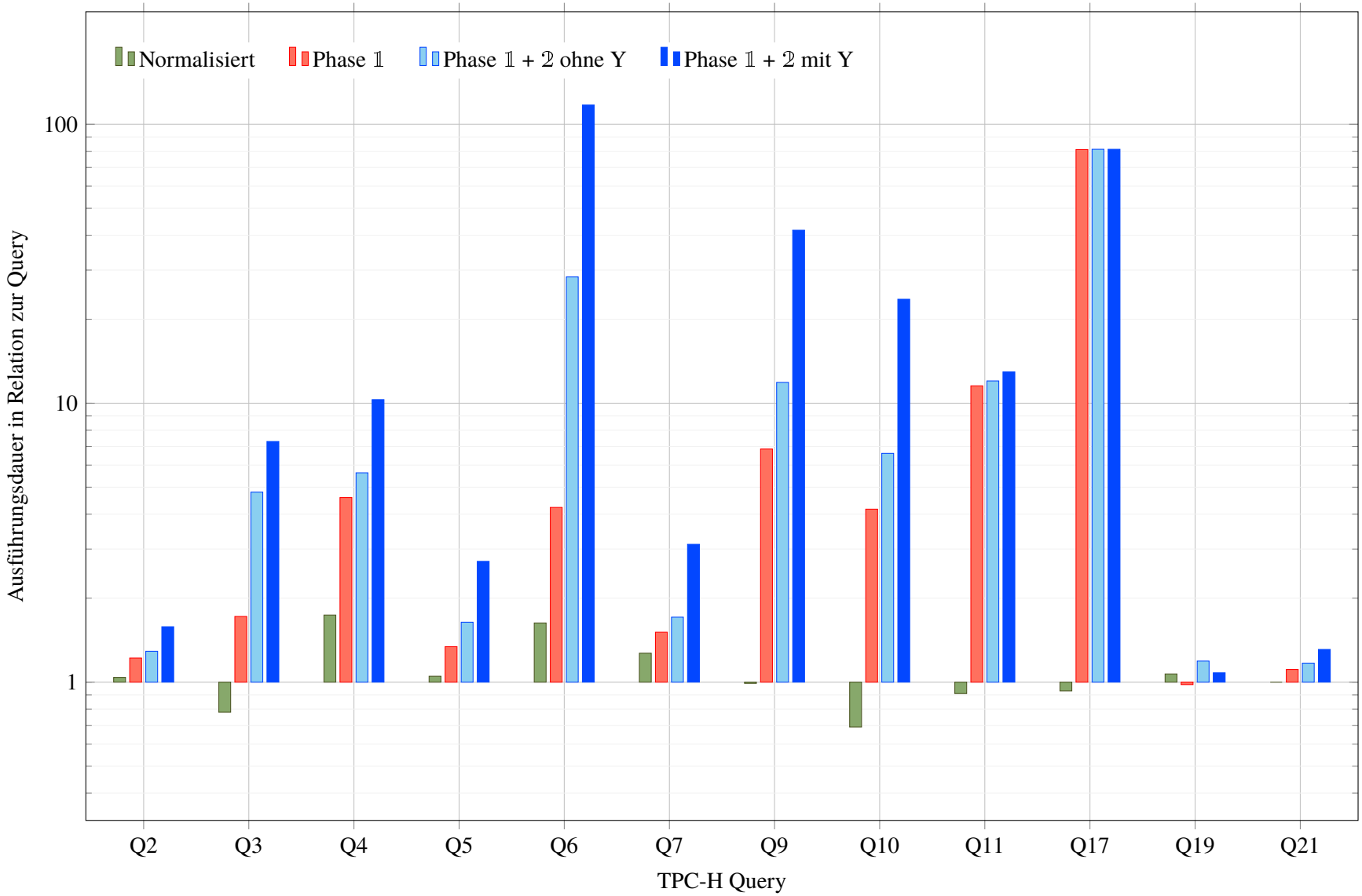

Abbildung 4.1: Performance Analyse der übersetzten TPC-H Queries

5 Ausblick

5.1 Fazit

Es wurde erfolgreich ein Programm geschrieben, welches die im Rahmen der *Language-Level Provenance Analysis* notwendigen Transformationen durchführt. Dabei werden nahezu alle gängigen SQL Konstrukte unterstützt oder lassen sich auf solche Weise normalisieren, dass diese von *SQLProv* übersetzt werden können. Auch die Implementierung der *Provenance-Sets* ist abstrakt gestaltet und lässt sich durch die Änderung des *PSet* Moduls an eine neue Darstellung anpassen. Zudem besitzt *SQLProv* verschiedene Programmargumente, um das Verhalten des Programms zu konfigurieren. Ein Beispiel ist die (De-)Aktivierung der *Why-Provenance* oder die dateibasierte Ein- und Ausgabe.

5.2 Future Work

Diese Arbeit bietet viel Raum für Erweiterungen und anknüpfende Projekte, welche in diesem Abschnitt vorgestellt werden.

5.2.1 Darstellung der Provenance-Sets

In der aktuellen Implementierung wird der in Postgres eingebaute *Array*-Datentyp für die Darstellung von *Provenance-Sets* verwendet. Diese verhalten sich aus mathematischer Sicht nicht duplikatfrei, was bei der Berechnung in Phase 2 zu sehr großen Zwischenergebnissen führen kann und eine Verschlechterung der Performance zur Folge hat.

An einer Lösung zu diesem Problem wird aktuell am Lehrstuhl gearbeitet. Dazu wird eine Erweiterung des Postgres entworfen, welche einen neuen Datentyp mit den gewünschten Eigenschaften hinzufügt[1]. Dieser ist duplikatfrei und ist in der Lage *Where-* und *Why-Provenance* gleichzeitig in einer Menge darzustellen.

[1] Im Repository existiert im Branch `pset` bereits eine angepasste Version von *SQLProv*, welche diesen Datentyp verwendet.

5.2.2 Unterstützung weiterer Queries

Bei der Übersetzung der TPC-H Queries wurden einige SQL Konstrukte verwendet, welche aktuell nicht von *SQLProv* unterstützt werden (`DISTINCT`, `LEFT OUTER JOIN`, `CASE ... WHEN`, . . .). Durch das Verwenden von Pattern Matching bei den Übersetzungsfunktionen ist die Implementierung neuer Regeln kein großer Aufwand.

5.2.3 Visualisierung

Die Queries der Phasen $\mathbb{1}$ und $\mathbb{2}$ berechnen Tabellen, welche die *Data Provenance* in Form von *Provenance-Sets* enthalten. Diese *Provenance-Sets* sind für kleine Tabellen und simple Queries noch übersichtlich, werden allerdings mit zunehmender Größe der Tabelle oder Komplexität der Query unlesbar für Menschen. Eine Visualisierung, welche die *Where-* und *Why-Provenance* farblich darstellt, würde diesem Problem Abhilfe schaffen.

Ein simples Programm namens *ProvVis*[2], welches die *Provenance-Sets* der Ergebnistabellen in eine Darstellung im Browser umwandelt, ist aus dieser Not heraus entstanden. Es erlaubt dem Nutzer mit der Maus über Zellen der Ausgabetabelle zu fahren, um eine farbliche Hervorhebung der Eingabezellen zu erhalten. Diese sind je nach Abhängigkeit über die *Where-* oder *Why-Provenance* unterschiedlich eingefärbt. Da *ProvVis* nicht Teil der Zielsetzung dieser Arbeit ist, existiert hier viel Raum zur Weiterentwicklung.

[2] *ProvVis* ist im Repository unter `PgLlProvenance/ProvVis/` zu finden.

Abbildungsverzeichnis

1.1 Data Provenance von Q mit *ProvVis* visualisiert 2

2.1 SELECT: Reihenfolge der Ausführung 8
2.2 Übersicht über SQL-basierte Berechnung der Provenance 12
2.3 Einordnung von *SQLProv* in die SQL Provenance Toolchain 17

3.1 SQLProv Modulübersicht 19

4.1 Performance Analyse der übersetzten TPC-H Queries 39

Literaturverzeichnis

[Don74] R. F. B. Donald D. Chamberlin. *SEQUEL: A Structured English Query Language*. 1974. URL: http://www.almaden.ibm.com/cs/people/chamberlin/sequel-1974.pdf (zitiert auf S. 6).

[Hir17] D. Hirn. *Compilation of SQL into KL*. 2017. URL: http://db.inf.uni-tuebingen.de/attachments/thesis-hirn-2017.pdf (zitiert auf S. 4).

[Jam07] W.-C. T. James Cheney Laura Chiticariu. *Provenance in Databases: Why, How, and Where*. 2007. URL: http://homepages.inf.ed.ac.uk/jcheney/publications/provdbsurvey.pdf.

[Mül16] T. Müller. *Have Your Cake and Eat it, Too: Data Provenance for Turing-Complete SQL Queries*. 2016. URL: http://db.inf.uni-tuebingen.de/staticfiles/publications/cake-turing-2016.pdf (zitiert auf S. 10).

Alle URLs wurden zuletzt am 24.03.2018 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift