Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelorthesis Informatik

Visualization of How-Provenance

Martin Lutz

25. August 2017

Gutachter

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

Betreuer

Tobias Müller
Universität Tübingen

Daniel O'Grady
Universität Tübingen

Lutz, Martin:
Visualization of How-Provenance
Bachelorthesis Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 26.04.2017 - 26.08.2017

Abstract

In dieser Arbeit wurde eine Visualisierung erstellt, die Where-, Why- und How-Provenance darstellen kann. Neben einer interaktiven SQL Queryanalyse lassen sich auch mehrere Queries verwalten. Neu gegenüber vorheriger Arbeiten ist hierbei die How-Provenance.

Inhaltsverzeichnis

Abkürzungsverzeichnis vii

1 Einleitung 1
1.1 Vorwort . . . 1
1.2 Visualisierung . . . 1
1.2.1 Core Features . . . 3
1.2.1.1 Darstellung der Data-Provenance . . . 3
1.2.1.2 Wechseln der SQL Query . . . 3
1.2.2 Zusätzliche Features . . . 4
1.2.2.1 Interaktive SQL Query-Analyse . . . 4
1.2.2.2 Debug-Modus . . . 5
1.2.2.3 KL Query . . . 5

2 Grundlagen 7
2.1 Data Provenance . . . 7
2.1.1 Where-Provenance . . . 8
2.1.2 Why-Provenance . . . 9
2.1.3 How-Provenance . . . 9
2.2 Datenbankstruktur . . . 11
2.2.1 call . . . 11
2.2.2 var . . . 11
2.2.3 shape . . . 11
2.2.4 prov . . . 12
2.2.5 mutual . . . 12
2.2.6 how . . . 12
2.2.7 Beispiel: Tabellen anhand einer einfachen Query . . . 12
2.3 Kernel Language . . . 14

3 Technische Aspekte 15
3.1 Verwendete Sprachen und Formate . . . 15
3.1.1 Clientseitig . . . 15
3.1.1.1 Hypertext Markup Language (HTML) . . . 15
3.1.1.2 Cascading Style Sheets (CSS) . . . 16
3.1.1.3 JavaScript . . . 17
3.1.2 Serverseitig . . . 17
3.1.2.1 Python . . . 17
3.1.2.2 PostgreSQL . . . 18
3.1.3 Kommunikation . . . 19
3.1.3.1 JavaScript Object Notation (JSON) . . . 19

3.2 Verwendete Frameworks, Bibliotheken und Ähnliches 20
3.2.1 Clientseitig 20
3.2.1.1 jQuery 20
3.2.1.2 jQuery UI 20
3.2.1.3 jQuery Hotkeys 20
3.2.1.4 DataTables 20
3.2.1.5 Bootstrap 21
3.2.1.6 bootstrap-select 21
3.2.1.7 mustache.js 21
3.2.1.8 Prism 21
3.2.1.9 Ace 21
3.2.1.10 Font Awesome 22
3.2.2 Serverseitig 22
3.2.2.1 Psycopg 22
3.2.2.2 Bottle 22
3.2.2.3 Weitere Module 22
3.3 Verwendete Konzepte 22
3.3.1 Model-View-Controller 22
3.3.2 Ajax 23

4 Implementierung 25
4.1 Server 25
4.1.1 Grundlagen von Bottle 25
4.1.2 Struktur 26
4.1.3 Optionen und Query-Verwaltung 27
4.1.3.1 config.json 28
4.1.3.2 query-info.json 29
4.1.4 Laden einer analysierten Query 29
4.1.4.1 Query 29
4.1.4.2 Tabellen 30
4.1.4.3 Debug-Regionen 33
4.1.5 Ausliefern der Provenance 33
4.1.5.1 Nur Tabellen-Items ausgewählt 33
4.1.5.2 Nur Regionen ausgewählt 34
4.1.5.3 Regionen und Tabellen-Items ausgewählt 35
4.1.6 Verwalten der Queries 35
4.1.6.1 Query-Auswahlliste 35
4.1.6.2 Analyse einer SQL Query 35
4.1.6.3 Löschen einer SQL Query 36
4.2 Client 37
4.2.1 `ProvServer` 38
4.2.1.1 `getProvSwitchList()` 39
4.2.1.2 `getAnnotatedSqlQuery(dbIndex)` 39
4.2.1.3 `getKLQuery(dbIndex)` 39

4.2.1.4 `getInputSqlQuery(dbIndex)` 40
4.2.1.5 `getCalls(dbIndex)` 40
4.2.1.6 `getProvenance(dbIndex, pids, rids)` 41
4.2.1.7 `getRegions(dbIndex)` 41
4.2.1.8 `getInvolvedRegions(dbIndex, pids)` 41
4.2.1.9 `getInvolvedShapeIds(dbIndex, rids)` 41
4.2.1.10 `deleteQuery(dbIndex)` 42
4.2.1.11 `analyzeSqlQuery(name, query)` 42
4.2.2 Übersicht der Visualisierung 43
4.2.3 Steuer- und Infobereich 44
4.2.3.1 Query Switch 44
4.2.3.2 Query-Eingabe 45
4.2.3.3 Debug-Modus 47
4.2.3.4 Provenance-Informationen 48
4.2.3.5 Steuerungsinformationen 48
4.2.4 SQL Query-Bereich 49
4.2.4.1 Formatierung der Query 50
4.2.4.2 Darstellung der How-Provenance 51
4.2.4.3 KL Query 52
4.2.4.4 Bearbeiten einer vorhandenen Query 54
4.2.5 Call-Bereich 55
4.2.5.1 Erzeugung eines Calls 56

5 Zusammenfassung und Ausblick 61
5.1 Zusammenfassung 61
5.2 Future Work 61
5.2.1 Nicht blockierende Aufrufe 61
5.2.2 Ersetzen von Mustache bei Tabellen 61
5.2.3 Speichern der Queries 61
5.2.4 Wählen der Quell-Datenbank 62
5.2.5 Tabellen-Abfrage vereinfachen 62
5.2.6 KL Highlighting 62

6 Anhang 63
6.1 Definition der KL Grammatik 63
6.2 Server API Call zur Analyse einer SQL Query 65
6.3 Vollständige KL Query 67
6.4 Private `parseRow(row)`-Funktion der Klasse `Table` 70
6.5 mustache.js-Template für Tabellen 72

Abbildungsverzeichnis 75

Literaturverzeichnis 76

Abkürzungsverzeichnis

JSON	JavaScript Object Notation
KL	Kernel Language
SQL	Structured Query Language
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
ES	ECMAScript
DOM	Document Object Model
Ajax	Asynchronous JavaScript and XML

1 Einleitung

1.1 Vorwort

Die Struktur dieser Thesis basiert auf der Thesis von Janek Bettinger [Bet14]. Auch einige Design-Elemente der zugehörigen Visualisierung wurden übernommen.

1.2 Visualisierung

Überall im Internet werden Daten und deren Verwaltung immer komplexer. Allein Facebook hat über 300 Millionen Nutzer und für jeden Nutzer werden Datensätze angelegt und verwaltet. Dahinter arbeiten tausende Datenbankserver. Schon anhand dieses Beispiels sieht man: Die Reproduzierbarkeit und das Verständnis der Entstehung der Daten gewinnt immer mehr an Bedeutung.

An diesem Punkt setzt Data Provenance (siehe Kapitel 2.1) an. In dieser Arbeit wurde eine Visualisierung erstellt, die verschiedene Data Provenance-Arten darstellen kann und Interaktion mit ihnen erlaubt. Neu gegenüber vorheriger Arbeiten ist hierbei die How-Provenance, die einen Zusammenhang mit der analysierten SQL Query herstellt.

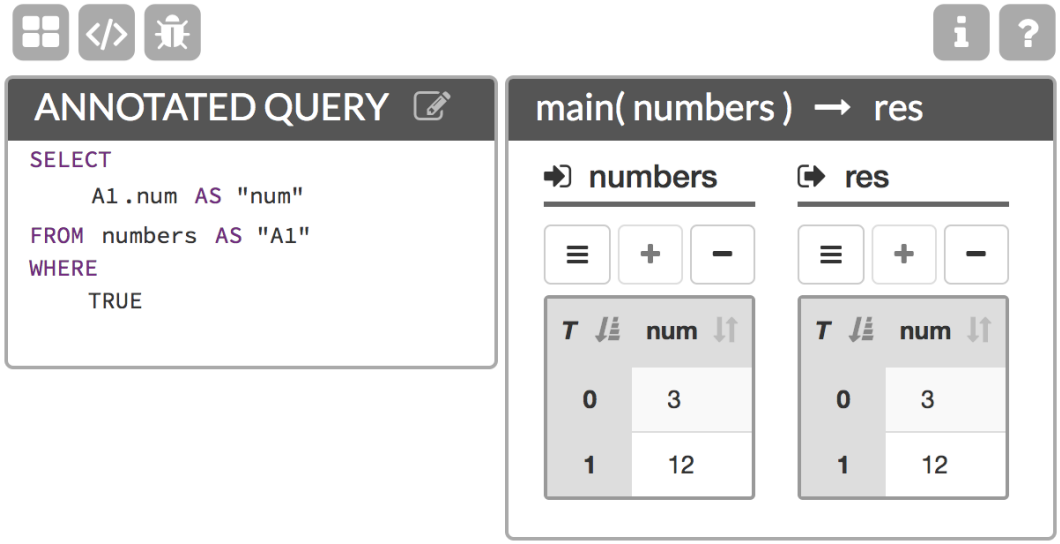

Abbildung 1.1: Screenshot der Visualisierung anhand eines Beispiels

Wir betrachten die SQL Query `Select num from numbers;`. Nach der Analyse stehen auf dem Server eine annotierte SQL Query, die tatsächliche SQL Query, die KL Query und eine Datenbank bereit.

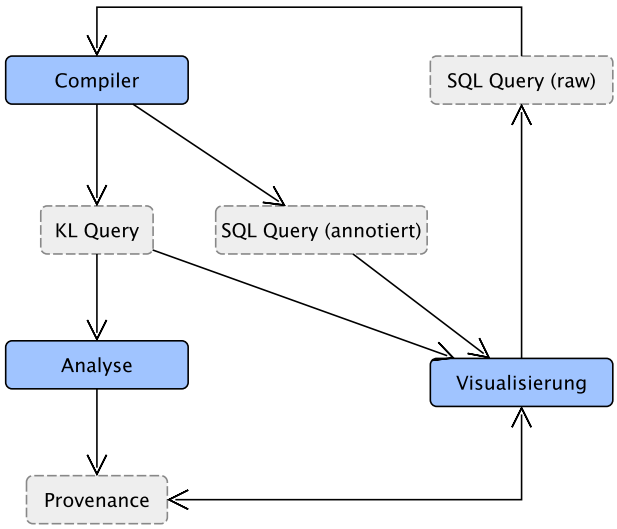

Abbildung 1.2: Position der Visualisierung in der Toolchain

Wie in Abbildung 1.2 zu sehen, übersetzt der Compiler eine SQL Query nach KL. Der Compiler wirft nebenbei auch eine annotierte SQL Query ab. Auf der KL Query läuft die Provenance-Analyse. Mit der daraus entstandenen Datenbank kommuniziert

die Visualisierung.

1.2.1 Core Features

1.2.1.1 Darstellung der Data-Provenance

Die annotierte SQL Query wird links angezeigt, die Ein- und Ausgabe-Tabellen rechts. Wenn wir nun eine Zelle in der Ausgabe-Tabelle auswählen, bekommen wir die Data Provenance-Arten (siehe Kapitel 2.1) farblich markiert.
In Abbildung 1.3 wird die Where-Provenance grün und die Why-Provenance rot in den Tabellen dargestellt. Hellblau und grau wird die How-Provenance in der annotierten SQL Query dargestellt. Dabei sieht man durch die Abgrenzungen auch die Unterteilung der Query in Regionen.

Wählen wir hingegen nur eine Region in der SQL Query, so bekommen wir die How-Provenance in der Ausgabe-Tabelle blau markiert.

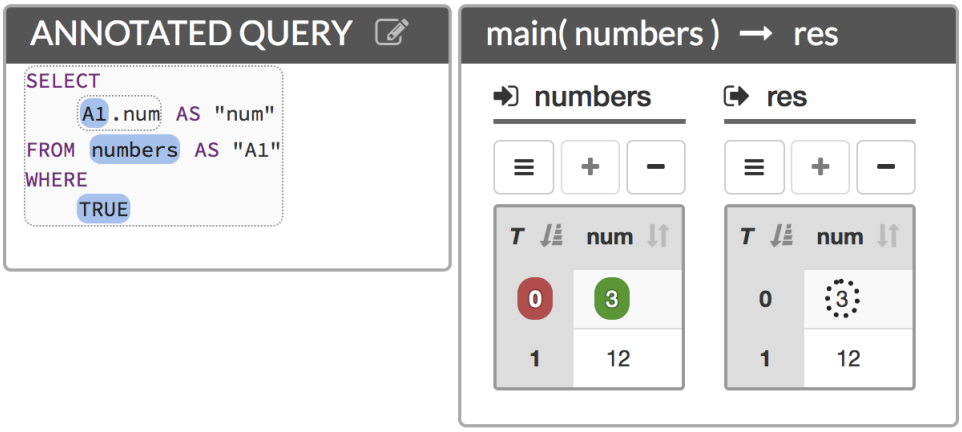

Abbildung 1.3: Zelle der Ausgabe-Tabelle ausgewählt

1.2.1.2 Wechseln der SQL Query

Auf dem Server können beliebig viele SQL Queries mit den zugehörigen Daten liegen. Zwischen ihnen kann mit einem Klick auf das Icon oben links gewechselt werden, siehe Abbildung 1.4.

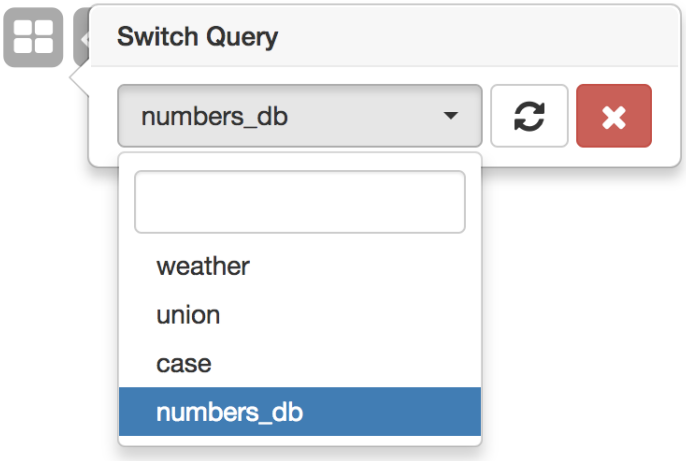

Abbildung 1.4: Wechseln der SQL Query

1.2.2 Zusätzliche Features

1.2.2.1 Interaktive SQL Query-Analyse

Eine vorhandene SQL Query kann über das Stift-Icon neben dem Text *ANNOTATED QUERY* bearbeitet werden. Darauf öffnet sich ein Editor mit der eingegebenen SQL Query und dessen Namen.

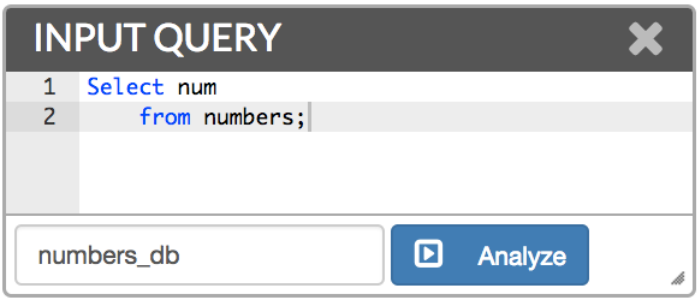

Abbildung 1.5: Eingabe einer SQL Query

Die Eingabe kann auch über das Code-Icon oben geöffnet werden. Mit einem Klick auf *Analyze* wird die SQL Query an den Server übermittelt und analysiert. Bei Fehlern gibt es Feedback, was schief gelaufen ist. Funktioniert alles, wird die SQL Query der Liste an verfügbaren Queries hinzugefügt und geladen.

1.2.2.2 Debug-Modus

Klickt man das Debug-Icon ganz oben, werden in allen Tabellen die Item-IDs und in der annotierten SQL Query die Regionen-IDs als Tooltip beim Hovern angezeigt. Zusätzlich werden die Regionen-IDs in einem eigenen Bereich aufgelistet, siehe Abbildung 1.6.

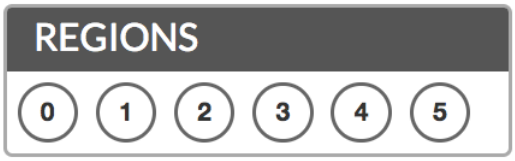

Abbildung 1.6: Regionen als Low-Level-IDs

1.2.2.3 KL Query

Vor der Analyse wird die SQL Query in KL (siehe Kapitel 2.3) übersetzt. Die KL Query kann mit einem Klick auf eine Region mit **alt** oder **c** angezeigt werden.

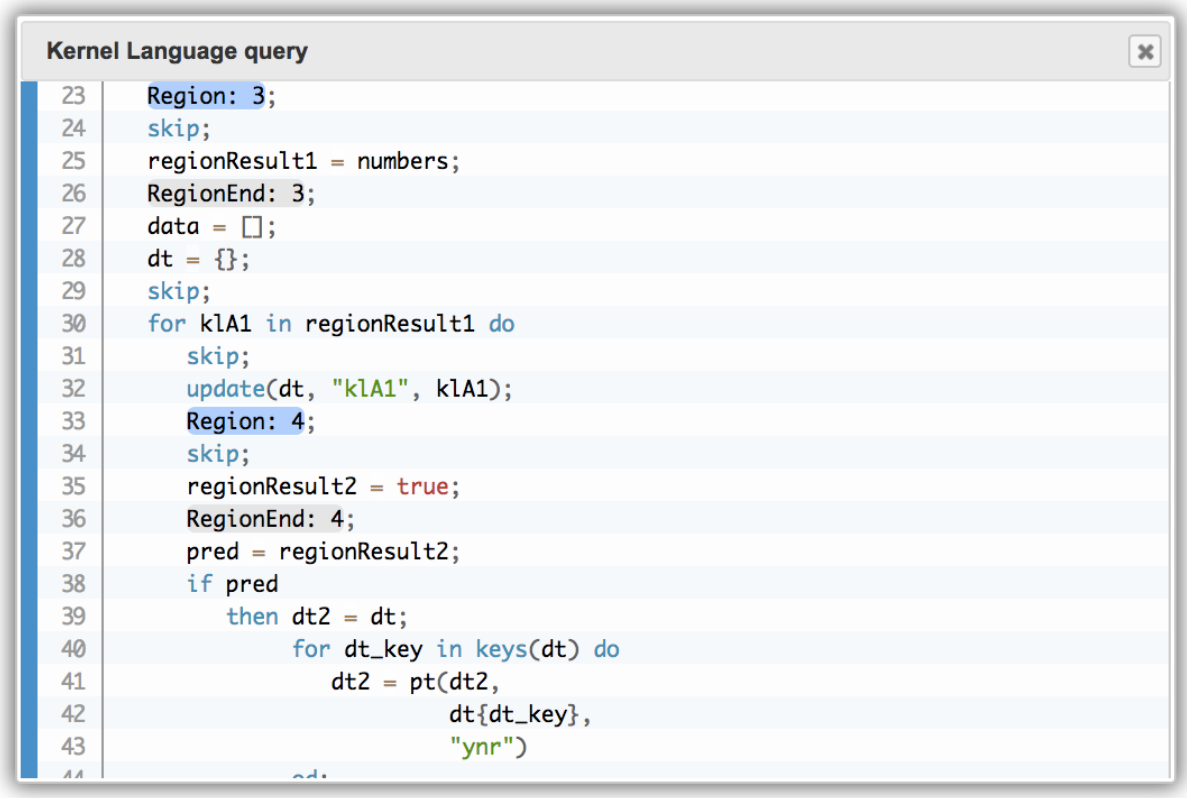

Abbildung 1.7: KL Query-Dialog

2 Grundlagen

2.1 Data Provenance

Data Provenance bezeichnet den Zusammenhang von aggregierten Tabellen zu Eingangs-Tabellen und damit die Entstehung derselben. Im Folgenden wird die Data Provenance gegliedert in Where-, Why- und How-Provenance. Ich beziehe mich auf die Definitionen aus dem Draft [MOG17, Fine-Grained How-Provenance for SQL and Query Compilers].

Bemerkung
Der Begriff *Item* wird in dieser Arbeit benutzt, um atomare Daten in den Tabellen zu beschreiben.

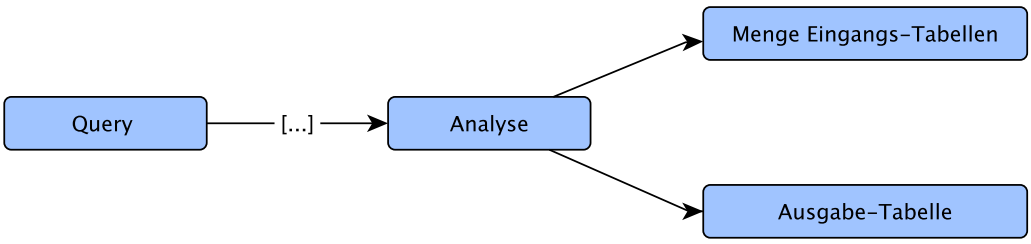

Abbildung 2.1: Von der Query zur Data Provenance

Die Analyse nutzt, abhängig von der SQL Query, beliebig viele Eingangs-Tabellen. Dabei entsteht immer genau eine Ausgabe-Tabelle.

```
select SUM(zahl)
     from zahlen
     where zahl > 5;
```

Listing 2.1: SQL Query, die eine Summe berechnet

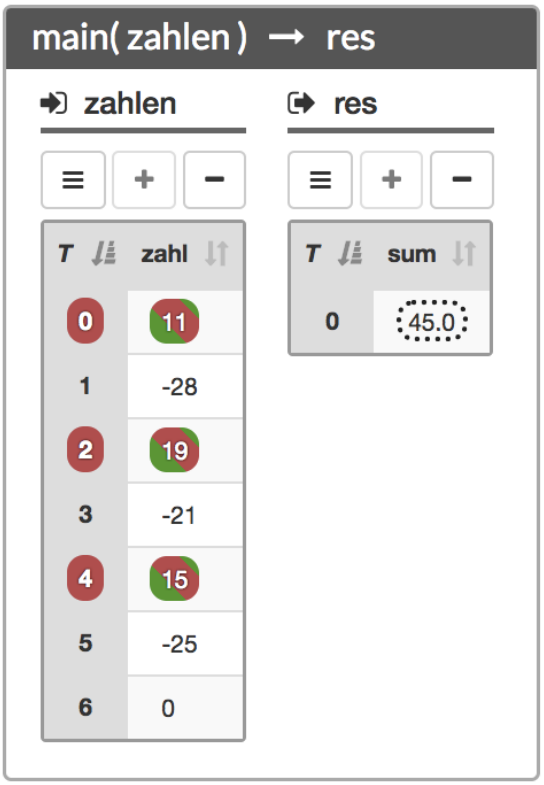

Abbildung 2.2: Tabellen zur SQL Query 2.1. Zelle der Ausgabe-Tabelle ist ausgewählt

2.1.1 Where-Provenance

Where-Provenance beschreibt die Eingangs-Items, die verwendet wurden um ein bestimmtes Ausgabe-Item zu *berechnen* bzw. zu *bestimmen*. Die Where-Provenance arbeitet also direkt auf Werte-Ebene.
In Abbildung 2.2 wird die Where-Provenance in der Eingabe-Tabelle grün dargestellt. Die Werte größer 5 der Tabelle zahl wurden verwendet, um `45.0` zu berechnen.

2.1.2 Why-Provenance

Why-Provenance stellt fest, welche Eingangs-Items *verglichen* wurden, um zu bestimmen, dass ein Item zur Ausgabe gehört. Die Why-Provenance ist also eher indirekt. In Abbildung 2.2 wird die Why-Provenance in der Eingangs-Tabelle rot dargestellt. Die Werte größer 5, als auch die zugehörigen Zeilen wurden verglichen, um zu bestimmen, dass die Werte zur Ausgabe beitragen.

2.1.3 How-Provenance

Während sich Where- und Why-Provenance auf die entstandenen Daten aus der SQL Query beziehen, geht es bei **How-Provenance** um den Zusammenhang mit dem Query-Text. Der Query-Text wird in sogenannte *Regionen* unterteilt, die Query-Kontrukte gliedern. Die How-Provenance beschreibt, welche Regionen zur Berechnung eines Items beigetragen haben.

```
select num
     from numbers
     where num > 0
union all
select num
     from numbers
     where num < 0;
```

Listing 2.2: Union SQL Query

Abbildung 2.3: Analyse der SQL Query 2.2. Item in der Ausgabe-Tabelle ist ausgewählt

In Abbildung 2.3 ist der linke SELECT-Bereich (größer 0) in der annotierten SQL Query markiert. Damit ist das linke SELECT-Statement für die Berechnung der 11 in der Ausgabe-Tabelle zuständig.

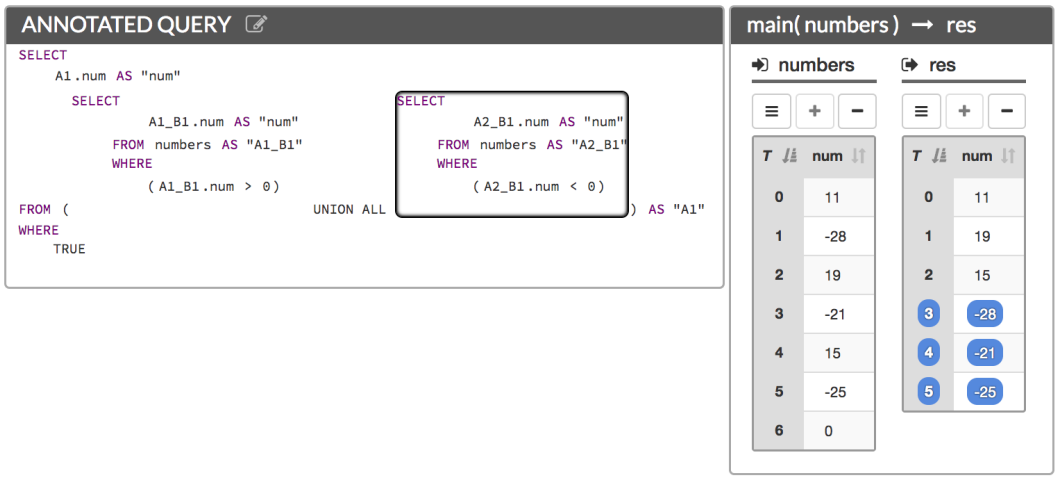

Abbildung 2.4: Analyse der SQL Query 2.2. Region im Query-Text ist ausgewählt

In Abbildung 2.4 sind alle Werte kleiner 0 markiert. Damit ist das rechte SELECT-Statement der annotierten SQL Query für die Berechnung der drei Werte in der Ausgabe-Tabelle zuständig.

2.2 Datenbankstruktur

Die Data Provenance-Analyse liefert eine Datenbank mit den Ergebnissen. Diese besteht aus folgenden Tabellen: call, var, shape, prov, how und mutual.

2.2.1 call

In der **call**-Tabelle werden die in der KL Query genutzten Funktionsaufrufe mit zugeordneten IDs gespeichert.

`id`	ID
`funcn`	Name der Funktion

Tabelle 2.1: Spalten der call-Tabelle

2.2.2 var

In der **var**-Tabelle stehen die Tabellennamen der Eingangs- und Ausgabe-Tabellen.

`id`	ID
`name`	Tabellenname
`isarg`	`true` ≡ Eingangs-Tabelle `false` ≡ Ausgabe-Tabelle
`call`	Fremdschlüssel zu call-`id`

Tabelle 2.2: Spalten der var-Tabelle

2.2.3 shape

In der **shape**-Tabelle stehen die Eingangs- und Ausgabe-Tabellen in flacher Form. Der Wert von `value` hängt von `type` ab und wird nur weiter ausgewertet, wenn `type` ein geschachtelter Wert ist.

`id`	ID
`contid`	`NULL` ≡ Tabellenbeginn Sonst: Verkettet mit `id` zusammengehörige Tabellenzellen
`idx`	Integer ≡ Spaltenindex String ≡ Spaltenname
`type`	Atomar: `null`, `bool`, `int`, `float`, `string` Geschachtelt: `table`, `list`, `dict`, `row`
`value`	`[?]` ≡ Tabelle `<?>` ≡ Record `{?}` ≡ Dictionary Sonst: Atomarer Wert
`var`	Fremdschlüssel zu call-`id`

Tabelle 2.3: Spalten der shape-Tabelle

2.2.4 prov

Die **prov**-Tabelle beinhaltet die Where- und Why-Provenances. Zwischen `output` und `input` besteht die Provenance-Beziehung. Es können bei n atomaren Werten für jeden `output`-Wert bis zu $n-1$ Provenance-Beziehungen in der prov-Tabelle stehen.

`output`	Output shape-`id`
`input`	Input shape-`id`
`iswhere`	`true` ≡ Where-Provenance `false` ≡ Why-Provenance

Tabelle 2.4: Spalten der prov-Tabelle

2.2.5 mutual

Die **mutual**-View[a] beinhaltet die Where- und Why-Provenances, genau wie die prov-Tabelle 2.2.4. Sie unterscheidet sich aber dadurch, dass die Provenances in beide Richtungen dargestellt werden. Damit kann auf die Provenances in *gleichförmiger* Weise zugegriffen werden.

[a] Eine View ist eine logische Relation in einem DBMS. Aus Anwendersicht gibt es keinen Unterschied zu einer Tabelle.

`pid`	Ausgewählte shape-`id`
`provenance`	Ergebnis shape-`id`
`iswhere`	`true` ≡ Where-Provenance `false` ≡ Why-Provenance

Tabelle 2.5: Spalten der mutual-View

2.2.6 how

Die **how**-Tabelle gibt die How-Provenance der annotierten SQL Query mit den Output-Items an.

`output`	Output shape-`id`
`region`	Region für shape-`id`

Tabelle 2.6: Spalten der how-Tabelle

2.2.7 Beispiel: Tabellen anhand einer einfachen Query

Nach dem Kompilieren der einfachen SQL Query `select num from numbers;` nach KL, werden in der Analyse der zugehörigen KL Query die eben besprochenen Tabellen folgendermaßen angelegt:

id	funcn
2	main

Tabelle 2.7: Erzeugte call-Tabelle

id	name	isarg
2	numbers	true
3	res	false

Tabelle 2.8: Erzeugte var-Tabelle

id	contid	idx	type	value	var
2	NULL	NULL	"list"	"[?]"	2
3	2	"0"	"dict"	"?"	2
4	3	"num"	ïnt"	"3"	2
6	2	"1"	"dict"	"?"	2
7	6	"num"	ïnt"	"12"	2
9	NULL	NULL	"list"	"[?]"	3
10	9	"0"	"dict"	"?"	3
11	10	"num"	ïnt"	"3"	3
13	9	"1"	"dict"	"?"	3
14	13	"num"	ïnt"	"12"	3

Tabelle 2.9: Erzeugte shape-Tabelle

output	input	iswhere
10	3	false
11	3	false
11	4	true
13	6	false
14	6	false
14	7	true

Tabelle 2.10: Erzeugte prov-Tabelle

pid	provenance	iswhere
4	11	true
7	14	true
11	3	false
3	10	false
13	6	false
6	13	false
6	14	false
3	11	false
11	4	true
14	6	false
10	3	false
14	7	true

Tabelle 2.11: Erzeugte mutual-Tabelle

output	region
9	5
10	3
10	4
10	5
11	1
11	2
11	3
11	4
11	5
13	3
13	4
13	5
14	1
14	2
14	3
14	4
14	5

Tabelle 2.12: Erzeugte how-Tabelle

2.3 Kernel Language

Kernel Language (KL) [Mü17] ist eine C-artige Sprache, die an der Eberhard Karls Universität Tübingen entwickelt wurde, um als eine Zielsprache für SQL zu dienen. KL lässt sich besser für die Provenance-Analyse nutzen als SQL direkt.

```
def main() {
    x = {"a":1,"b":2};
    return x
};
res = main()
```

Listing 2.3: KL Code-Beispiel

Einordung

Die Query kann in der Visualisierung neben SQL auch in KL angezeigt werden, um den Zusammenhang zwischen KL Query und SQL Query zu zeigen. Für einen genauen Überblick über die KL-Grammatik, siehe Anhang (Kapitel 6.1).

3 Technische Aspekte

3.1 Verwendete Sprachen und Formate

3.1.1 Clientseitig

3.1.1.1 Hypertext Markup Language (HTML)

HTML ist eine Auszeichnungssprache zur Strukturierung von Texten, Bildern und anderen Inhalten. Laut Standard [HM5] besteht ein HTML-Dokument aus dem `<html>`-Tag als Wurzel, welcher den `<head>`- und `<body>`-Tag beinhaltet. Der `<head>` definiert neben dem Titel noch einige Metadaten des Dokuments. Im `<body>` steht der Inhalt, der wiederum aus geschachtelten Tags bestehen kann.
Vor allem im Web ist HTML Standard und bildet die Grundlage des World Wide Web. Mittels CSS-Klassen und -IDs im Tag lassen sich Designvorgaben umsetzen.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width,
                                       initial-scale=1.0">
        <title>Pagetitle</title>
    </head>
    <body>
        <p>Write your content here.</p>
    </body>
</html>
```

Listing 3.1: HTML5 Minimalbeispiel [HDU, Grundgerüst]

Einordnung
Mittels JavaScript lässt sich HTML via DOM dynamisch zur Laufzeit verändern. Das DOM beschreibt die HTML-Struktur als einen Knotenbaum und definiert damit eine

Schnittstelle auf die Struktur. In der Visualisierung wird von DOM-Manipulationen rege Gebrauch gemacht, um eine dynamische und interaktive Anwendung zu kreieren.

3.1.1.2 Cascading Style Sheets (CSS)

Während HTML die Websprache zur Strukturierung ist, ist **CSS** [CS3] die Stylesheet-Sprache im Web zur Definition der Darstellung. Mit CSS lassen sich also Inhalte designen. Definitionen können via *ID*, *Klasse* oder direkter *Tag*-Zuweisung gruppiert werden.

```
p { /* Tag */
     padding-left: 5px;
     color: grey;
}

#sql-editor-container { /* ID */
     position: relative;
     height: 400px;
     min-width: 380px;
     border-bottom: 1px solid #AAAAAA;
}

.area { /* Class */
     background-color: #FFFFFF;
     border: 2px solid #AAAAAA;
     border-radius: 5px;
     margin-bottom: 5px;
}
```

Listing 3.2: CSS3 Code-Beispiel

Einordnung
CSS-Klassen werden in der Visualisierung mittels DOM-Manipulation zur Laufzeit hinzugefügt oder entfernt, um Tabellenelemente zu markieren und Sichtbarkeiten zu verändern.

3.1.1.3 JavaScript

JavaScript, eine Implementierung von ECMAScript (ES) [ES6], ist eine Skriptsprache und wird hauptsächlich in der Webentwicklung verwendet. Mit JavaScript lässt sich HTML via DOM dynamisch zur Laufzeit verändern und ermöglicht damit Nutzerinteraktion.

```
function printToBody(var text) { // Append text to body
     document.getElementsbyTagName("body")[0].appendChild(text + "<br>");
}

let countTo = 20;
for (let i = 0; i < countTo; i++) {
     if (i % 2 === 0) { // Even number
          printToBody(i + " even");
     } else { // Odd number
          printToBody(i + " odd");
     }
}
```

Listing 3.3: JavaScript (ES6) Code-Beispiel

Einordnung
Jegliche Userinteraktion in der Visualisierung wird zuerst einmal mit JavaScript ausgewertet. In der Umsetzung dieser Arbeit werden viele Konzepte aus **ES6** verwendet, was z.B. *syntaktischen Zucker* für Klassen oder auch echte lokale Variablen via `let` erlaubt.

3.1.2 Serverseitig

3.1.2.1 Python

Python [PY3] ist eine interpretierte Programmiersprache. Besonders markant ist der Verzicht auf Klammern zur Strukturierung. Stattdessen werden zusammengehörige Blöcke auf gleiche Ebene eingerückt.
Viele Dinge lassen sich besonders kompakt schreiben, z.B.:

- `range(10)` - Liste von 0-9
- `file = open(filename)` - Datei aus dem Dateisystem öffnen

```python
# Function: calculate average of a list
def avg(lst):
    return sum(lst) / len(lst)

# Simple list
list1 = [10, 99, 20, 76, 94, 2, 108]
print(avg(list1)) # 58.42857142857143

# List created in for loop
list2 = []
for i in range(10):
    list2.append(i)
print(avg(list2)) # 4.5
```

Listing 3.4: Python 3 Code-Beispiel

Einordnung
Mit **Python** wurde der Server der Visualisierung implementiert. Durch ihn erfolgt unter anderem der Zugriff auf die PostgreSQL-Datenbank. Fast alle Interaktionen im Interface lösen via *POST* eine asynchrone Anfrage an den Server aus.

3.1.2.2 PostgreSQL

PostgreSQL[1] ist ein objektrelationales Datenbankmanagementsystem (ORDBMS). Postgres hält sich weitgehend an den SQL-Standard.

```sql
select distinct h.region as region
from mutual m, how h, shape s
      where m.id=h.prov
```

Listing 3.5: PostgreSQL-Query-Beispiel

Einordnung
Die Postgres-Datenbank beinhaltet die Ergebnisse der Data Provenance-Analyse.

[1] https://www.postgresql.org

Mittels serverseitigen Queries werden Zusammenhänge und Daten der Analyse abgefragt.

3.1.3 Kommunikation

Die Server/Client-Kommunikation läuft asynchron mittels Ajax. Um den jeweiligen Kommunikationspartner verstehen zu können, wird eine Einigung im Datenformat benötigt.

3.1.3.1 JavaScript Object Notation (JSON)

JSON[2] ist ein einfaches Datenformat, das häufig zum Datenaustausch verwendet wird. In `{}`-Klammern werden *Key/Value*-Einträge genutzt, um mit dem Key auf den Wert zugreifen zu können. Ein Wert kann wiederum geschachtelt sein.
Ein Key muss ein String sein oder eine Stringrepräsentation anbieten. Ein Value kann sein: Nullwert, Boolean, Zahl, String, Array oder Objekt.

```
{
    "username": "",
    "queries": [
    {
        "database": "weather_prov",
        "codefolder": "weather"
    },
    {
        "database": "union_prov",
        "codefolder": "union"
    }
    ],
    "userpass": "",
    "dbhost": ""
}
```

Listing 3.6: JSON Code-Beispiel mit einem Array

Einordnung
Der Austausch fast aller Daten zwischen Server und Client erfolgt in der Visualisie-

[2] https://tools.ietf.org/html/rfc7159

rung im JSON-Format.

3.2 Verwendete Frameworks, Bibliotheken und Ähnliches

3.2.1 Clientseitig

Clientseitig wurden sehr viele moderne JavaScript-Bibliotheken, -Plugins und -Frameworks verwendet.

3.2.1.1 jQuery

jQuery[3] ist eine JavaScript-Bibliothek, die zur Vereinfachung der DOM-Manipulationen verwendet wird. Außerdem bringt jQuery Ajax-Funktionalität via `jQuery.ajax()`, ein erweitertes Event-System sowie einige Animationen, wie z.B. `jQuery.slideUp()`, mit sich.

3.2.1.2 jQuery UI

jQuery UI[4] ist ein JavaScript-Plugin für jQuery. Es erweitert jQuery um diverse Interface-Interaktionen, wie das Ziehen oder das Verändern der Größe von Elementen. Zudem gibt es Effekte und einige Widgets (Dialog, Tooltip etc.).

3.2.1.3 jQuery Hotkeys

jQuery Hotkeys[5] ist ein JavaScript-Plugin für jQuery, das Shortcuts auf eine Webseite oder eine Webanwendung bringt. Diverse nützliche Kürzel wurden in der Visualisierung damit umgesetzt. Die Syntax sieht folgendermaßen aus: `jQuery(document).bind('keydown', 'ctrl+a', fn);`.

3.2.1.4 DataTables

DataTables[6] ist ein JavaScript-Plugin für jQuery. Es erweitert die standard HTML-Tabellen um interaktive Funktionalität wie eine Suche, das Sortieren nach Spalten, das Begrenzen der anzuzeigenden Einträge uvm. Da das Ausblenden von Zeilen

[3] https://jquery.com
[4] https://jqueryui.com
[5] https://github.com/jeresig/jquery.hotkeys
[6] https://datatables.net

kein Feature von DataTables ist, wurde auf einige Features, wie die Suche, verzichtet, die das Ausblenden verhinderten oder verkomplizierten.

3.2.1.5 Bootstrap

Bootstrap[7] ist ein HTML-, CSS- und JavaScript-Framework. Für die DataTables-Tabellen wurde ein Bootstrap-basierendes Design verwendet.

3.2.1.6 bootstrap-select

Für die Query-Auswahl wurde das jQuery-Plugin **bootstrap-select**[8] verwendet, welches für das Design auf Bootstrap aufbaut. Es bietet unter anderem eine Suche für die Listeneinträge.

3.2.1.7 mustache.js

mustache.js[9] implementiert das Mustache-Template-System[10] in JavaScript. Es wird für das Rendern der Tabellen, der Query-Auswahl und der Debug-Regionen für die How-Provenance genutzt.

3.2.1.8 Prism

Prism[11] ist ein Regex-basierter Syntax-Highlighter, der die KL Query highlightet. Ursprünglich wurde mit highlight.js experimentiert. Die Entscheidung fiel wegen der einfachen Syntax zur Erstellung einer Sprachdefinition jedoch auf Prism.

3.2.1.9 Ace

Ace[12] ist ein eingebetteter Code-Editor, der in JavaScript geschrieben wurde. Genutzt wird er für die Eingabe einer SQL Query zur interaktiven Data Provenance-Analyse. Neben Syntax-Highlighting werden Zeilennummern angezeigt.

[7] http://getbootstrap.com
[8] https://silviomoreto.github.io/bootstrap-select/
[9] https://github.com/janl/mustache.js/
[10] http://mustache.github.io
[11] http://prismjs.com
[12] https://ace.c9.io

3.2.1.10 Font Awesome

Font Awesome[13] ist eine Schriftart, die aus Symbolen besteht. Alle Icons der Visualisierung sind damit umgesetzt, da sie stufenlos skalierbar und via CSS anpassbar sind.

3.2.2 Serverseitig

3.2.2.1 Psycopg

Psycopg[14] ist ein PostgreSQL-Adapter für Python, der Zugriff auf die PostgreSQL-Datenbanken ermöglicht.

3.2.2.2 Bottle

Bottle[15] ist eine Bibliothek für Python zum Erstellen von Web Server Gateway Interface (WSGI) Anwendungen. Per *POST* und *GET* werden JSON-Daten und unterschiedliche Dateien an den Client ausgeliefert. Dabei stellt das Server Interface das Application Programming Interface (API) dar.

3.2.2.3 Weitere Module

os wird zum Ermitteln des working directorys, **shutil** zum Löschen einer Query, **subprocess** zum Ausführen anderer Programme und **json** zum Austausch mit dem Client und Lesen der JSON-Dateien verwendet.

3.3 Verwendete Konzepte

3.3.1 Model-View-Controller

Mit Hilfe des **Model-View-Controller**-Patterns[16] wird eine Trennung der Darstellung, der Steuerung des Programms und dem Datenmodell erreicht. Klassisch findet die Kommunikation zwischen allen Komponenten statt. Wir betrachten ein spezielles Pattern, bei dem die Kommunikation des Model und der View stets über den Controller erfolgt. Siehe Abbildung 3.1.

[13] http://fontawesome.io
[14] http://initd.org/psycopg/
[15] https://bottlepy.org/docs/dev/
[16] https://de.wikipedia.org/wiki/Model_View_Controller

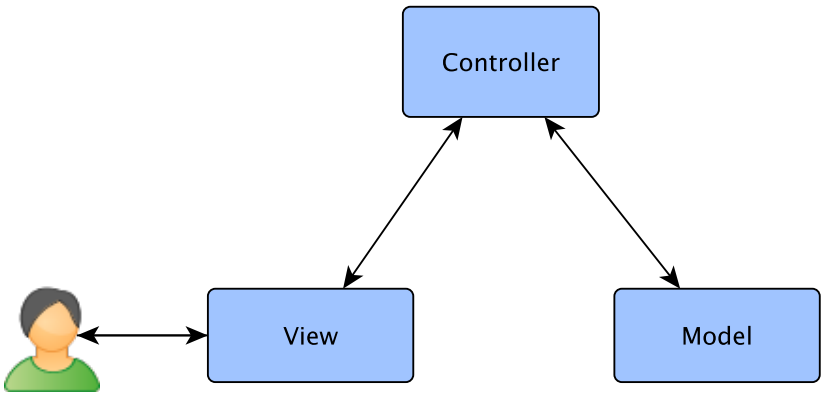

Abbildung 3.1: Model View Controller Grafik

Model
Das Model enthält die Daten und die Logik.

View
Die View übernimmt die Darstellung der Daten aus dem Model. Außerdem geschieht jede Nutzerinterkation in der View und wird dann an den Controller weitergeleitet.

Controller
Der Controller verwaltet die Kommunikation zwischen View und Model. Heißt, er verarbeitet die von der View weitergeleiteten Interaktionen. Entweder werden Daten an das Model weitergeleitet und/oder es wird die View aktualisiert.

3.3.2 Ajax

Ajax ist ein Konzept zur asynchronen Datenübertragung zwischen Client und Server. Damit wird ermöglicht, dass HTTP-Anfragen durchgeführt werden können, ohne die ganze Seite im Browser neu laden zu müssen. Es werden per JavaScript nur ausgewählte Teile der Seite aktualisiert. Außerdem bleibt die Seite ansprechbar, da die Kommunikation asynchron durchgeführt wird.

4

Implementierung

Die Visualisierung besteht aus drei Komponenten: dem Client, dem Server und der Datenbank. Im Folgenden werden aber nur die ersten Beiden behandelt. Eine Übersicht über die Datenbankstruktur findet sich in Kapitel 2.2.

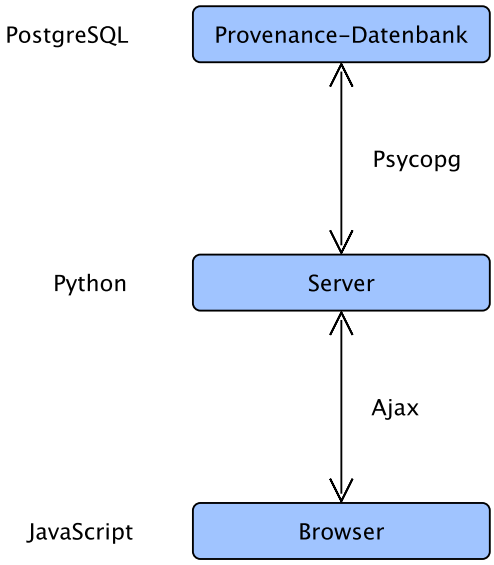

Abbildung 4.1: Server-Client-Übersicht

4.1 Server

Der Server wurde in Python mit Hilfe von Bottle (Kapitel 3.2.2.2) umgesetzt. Da die Komplexität sehr überschaubar geblieben ist, habe ich mich serverseitg für ein Imperatives Modell entschieden.

4.1.1 Grundlagen von Bottle

Das grundlegendste aber zugleich auch wichtigste Feature von Bottle ist der `route()`-Decorator. Damit wird ein URL-Pfad mit einer Funktion verknüpft. Der Rückgabe-

wert der Funktion wird zurück an den aufrufenden Browser geleitet. Ein kleines Beispiel findet sich in Listing 4.1.

```
@route('/example-path')
     def hello():
          return 'Hi!'
```

Listing 4.1: Bottle-Beispiel für den `route()`-Decorator

Für POST-Requests[1] wird `post()` genutzt, was aber nur eine Kurzschreibweise für `route('...', method='POST')` ist. Per POST überlieferte Daten werden serverseitig über das `request`-Objekt abgerufen.

4.1.2 Struktur

Der Server liefert dem Client zahlreiche Dateien aus. Diese unterliegen einer Struktur.

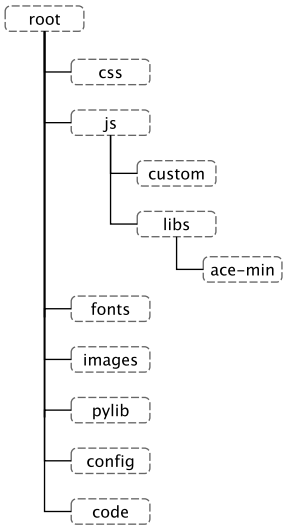

Abbildung 4.2: Verzeichnisstruktur auf dem Server

Verzeichnis	Inhalt
root	index.html - Grundstruktur des Clients
css	Alle Stylesheets
js/custom	Alle selbstgeschriebenen JavaScript-Dateien
js/libs	JavaScript-Bibliotheken und -Frameworks
fonts	Schriftarten
images	Bilder für jQuery-UI
pylib	mydb.py - Hilfklasse für Datenbankzugriffe
config	config.json - Einstellungen des Servers query-info.json - Hilfsdatei zur Query-Verwaltung
code	Jedes Verzeichnis repräsentiert eine SQL Query

Tabelle 4.1: Beschreibung der Verzeichnisse des Servers

[1] POST erlaubt die Übertragung von Daten über einen eigenen Kanal für einen Austausch zwischen Server und Client.

Das code-Verzeichnis bedarf einer genaueren Erklärung. In ihm sind beliebig viele Verzeichnisse, die alle je eine analysierte SQL Query repräsentieren. Ein solches Verzeichnis beinhaltet die tatsächliche und die annotierte SQL Query, sowie die KL Query. Die zugehörige Datenbank für die Analyseergebnisse steht in der query-info.json des config-Verzeichnisses.

4.1.3 Optionen und Query-Verwaltung

Sowohl die Optionen, als auch die Informationen zur Query-Verwaltung liegen im Verzeichnis config und sind im JSON-Format.

```
if __name__ == '__main__':
    QUERY_INFO = load_query_info()
    CONFIG = load_config()
    run(host='localhost', port=8000) # runs the server
```

Listing 4.2: main-Funktion des Servers

In der main-Funktion des Servers (Listing 4.2) werden config.json und query-info.json in die globalen Objekte `QUERY_INFO` und `CONFIG` geladen. Die `load`-Funktionen dazu sind bis auf den Pfad identisch. Die Datei wird geöffnet und mit `json.load()` als Dictionary geladen. Siehe Listing 4.3.

```
def load_config():
    "Return content of config.json in dir 'config'."
    with open('config/config.json') as file:
        return json.load(file)
```

Listing 4.3: `load_config()`-Funktion des Servers

4.1.3.1 config.json

Die config.json sieht folgendermaßen aus:

```
{
    "dbHost": "",
    "userPass": "",
    "userName": "martin",
    "sourceDbName": "demo",
    "inputQueryName": "input.sql",
    "annotatedQueryName": "query.sql",
    "klQueryName": "query.kl",
    "dbSuffix": "_prov_analysis"
}
```

Listing 4.4: Inhalt der Datei config.json

Host, Passwort und Username für den Datenbankzugriff sind in `dbHost`, `userPass` und `userName` definiert. Die Datenbank, aus der die Analyse lesen kann, ist in `sourceDbName` eingetragen.
Für jede analysierte SQL Query wird ein Ordner mit dem vom User gewählten Namen angelegt. In diesem Ordner gibt es zwei SQL-Dateien und eine KL-Datei. Die Namen dafür sind über `inputQueryName`, `annotatedQueryName` und `klQueryName` festgelegt. `dbSuffix` wird an den vom User eingegebenen Query-Namen angehängt. Diese neue Zeichenkette wird der Datenbankname der Analysedatenbank.

4.1.3.2 query-info.json

Die query-info.json sieht folgendermaßen aus:

```
{
    "queries": [
        {
            "database": "numbers_prov_analysis",
            "codefolder": "numbers"
        },
        {
            "database": "weather_prov_analysist",
            "codefolder": "weather"
        }
    ]
}
```

Listing 4.5: Inhalt der Datei query-info.json mit zwei Einträgen

Bei der Verwaltung mehrerer SQL Queries helfen die Einträge in `queries`. `database` gibt die Analysedatenbank des Eintrags an und `codefolder` den dazugehörigen Ordner mit den zwei SQL Queries und der KL Query.

4.1.4 Laden einer analysierten Query

4.1.4.1 Query

Die Query liegt serverseitig für jede analysierte SQL Query drei mal vor. In KL und zwei mal in SQL, annotiert und nicht-annotiert. In Listing 4.6 ist die Funktion zur Auslieferung der annotierten SQL Query zu sehen. Über den `db_index` wird die SQL Query ausgewählt und darüber der Pfad ermittelt. `get_code_path(index)` konkateniert den String `'code'` mit `QUERY_INFO['queries'][index]['codefolder']`. Dann wird noch der Dateiname angehängt, die Datei gelesen und als String zurück-

gegeben.

```
@post('/db/get_sql_query')
def get_sql_query():
    """Return sql query as string."""
    db_index = request.json['dbindex']
    path = get_code_path(db_index) + CONFIG['annotatedQueryName']
    with open(path) as file:
        return file.read()
```

Listing 4.6: `/db/get_sql_query Server` API Call

Die Auslieferung erfolgt für die nicht-annotierte SQL Query und die KL Query analog.

4.1.4.2 Tabellen

Die `get_calls()`-Funktion lädt zuerst alle Tabellen-Informationen. Tabellen werden serverseitig `var` genannt. Mit der Anfrage wird über `db_index` angegeben, welche Datenbank ausgewählt werden soll. Das Konzept ist bei allen Funktionen gleich, die eine Query auf einer Datenbank ausführen: Über die Hilfsfunktion `get_database_cred(db_index)` werden die notwendigen Informationen wie Host, Datenbankname, sowie User und Passwort für die Datenbank geladen. Bis auf den Datenbanknamen sind die Informationen in der config.json-Datei festgelegt. Der Datenbankname kommt aus der query-info.json und wird mit dem Index ausgewählt: `QUERY_INFO['queries'][index]['database']`;

Die Query aus Listing 4.7 liefert aus den Tabellen call und var die Tabellen-Informationen nach der Struktur in Abbildung 4.3. Da wir die Tabellen aber nach den Calls gruppiert haben wollen, verschachtelt die Funktion `nested_call()` die Tabellen unter den Calls eines Call-Arrays Schlussendlich bekommen wir die Struktur aus Abbildung 4.4, die an den Client geliefert wird.

```python
@post('/db/get_calls')
def get_calls():
    """Return array of json objects each representing one call."""
    db_index = request.json['dbindex']
    db = MyDB(get_database_cred(db_index))
    query = """
        select c.id callid, c.funcn funcname,
        v.id varid, v.name varname, v.isarg isarg
        from call c, var v
        where c.id = v.call
        order by v.isarg desc, v.name asc
        """
    res = db.select(query, ())
    response.content_type = 'application/json'
    return json.dumps(nested_call(res))
```

Listing 4.7: `/db/get_calls` Server API Call

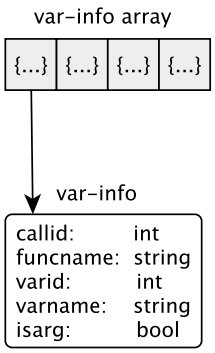

Abbildung 4.3: Von der Query in `get_calls()` gelieferte Struktur

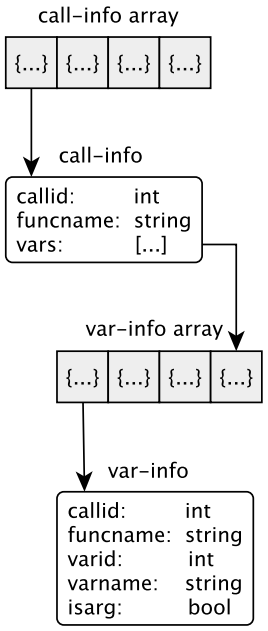

Abbildung 4.4: Von `nested_call()` gelieferte Struktur

Für jede dadurch erhaltene Tabelle wird `/db/get_var` (Listing 4.8) mit der `varid` aufgerufen.

```
@post('/db/get_var')
def get_var():
    """Return table as nested structure.
    @param varid: Id of requested table (shape->var == varid)."""
    varid = int(request.json['varid'])
    db_index = request.json['dbindex']
    db = MyDB(get_database_cred(db_index))
    res = db.selectOne("select * from shape where var=%s and idx is null",
                                                                (varid, ))
    res['elements'] = nested_var(res['id'])
    response.content_type = 'application/json'
    return json.dumps(res)
```

Listing 4.8: `/db/get_var` Server API Call

Die SQL Query `select * from shape where var=%s and idx is null` wird mit der Funktion `selectOne()` der Hilfsklasse ausgeführt, welche den ersten Eintrag mit der angegebenen `varid` aus der shape-Tabelle auswählt, bei dem `idx` null ist. Dieser Eintrag wird über die Funktion `nested_var(curid)` (Listing 4.9) rekursiv erweitert.

```
def nested_var(curid):
    """Return shape as array in nested form."""
    db_index = request.json['dbindex']
    db = MyDB(get_database_cred(db_index))
    rows = db.select("select * from shape where contid=%s", (curid, ))
    res = []
    for row in rows:
        nextid = row['id']
        nextrows = nested_var(nextid)
        row['elements'] = nextrows
        res.append(row)
    return res
```

Listing 4.9: `nested_var()`-Hilfsfunktion

Über die Verknüpfung von `contid` mit `id` (siehe Kapitel 2.2.3) wird eine geschachtelte Struktur erstellt, wie in Abbildung 4.8 zu sehen. Im `elements`-Eintrag stehen jeweils die nächsten Werte.

4.1.4.3 Debug-Regionen

Die Regionen sind schon in der annotierten SQL Query vorhanden, können aber zu Debugzwecken in der Visualisierung mit ihren IDs eingeblendet werden. Dafür wird eine Liste aller Regionen mit der SQL Query `select distinct region as region from how order by region` ausgelesen. Anschließend werden die Regionen in ein Integer-Array konvertiert und zurückgegeben.

4.1.5 Ausliefern der Provenance

Die Auslieferung der Data Provenance gliedert sich in drei Fälle. Nur Tabellen-Items ausgewählt, nur Regionen ausgewählt oder beides ausgewählt. In der annotierten SQL Query lässt sich die How-Provenance anzeigen und in den Tabellen die How-, Where- und Why-Provenance. Für die Unterscheidung der Provenance-Arten in den Tabellen werden Objekte mit den Attributen `pid` und `class` genutzt, wie in Abbildung 4.8 zu sehen.

4.1.5.1 Nur Tabellen-Items ausgewählt

In diesem Fall wird `/db/get_involved_regions` aufgerufen und die IDs der ausgewählten Tabellen-Items als Integer-Array übergeben. Die betroffenen Regionen werden mit der SQL Query `select distinct region from how where output=any(%s)` ausgewählt, wobei `%s` das übergebene Integer-Array ist. Heißt, alle Regionen, die zur Berechnung der Items beigetragen haben, werden zurückgeliefert.
Außerdem wird `/db/get_provenances` (Listing 4.10) für die Where- und Why-Provenances zwischen den Eingangs-Tabellen und der Ausgabe-Tabelle aufgerufen. Übergeben werden die ausgewählten Item-IDs und Regionen-IDs, wobei Letztere in diesem Fall irrelevant sind. Die PL/SQL-Funktion `getCssClasses(pids, rids)` bestimmt alle Data Provenances in Form von Abbildung 4.8. Die PL/SQL-Funktion stammt von Tobias Müller und ist Teil des Datenbankschemas aus dem Provenance-

Analysetool.

```
@post('/db/get_provenance')
def get_provenance():
    """Return list of dicts that assings pids to provenance css classes.
    @param pids: Selected table items.
    @param rids: Selected regions."""
    pids = request.json['pids']
    rids = request.json['rids']
    db_index = request.json['dbindex']
    params = (pids, rids)
    db = MyDB(get_database_cred(db_index))
    res = db.select("select * from getCssClasses(%s, %s)", params)
    response.content_type = 'application/json'
    return json.dumps(res)
```

Listing 4.10: `/db/get_provenance` Server API Call

4.1.5.2 Nur Regionen ausgewählt

In diesem Fall wird `/db/involved_shape_ids` aufgerufen und die IDs der ausgewählten Regionen als Integer-Array übergeben. Die How-Provenance in den Tabellen wird mit der SQL Query in Listing 4.11 ausgewählt, wobei auch hier `%s` das übergebene Integer-Array ist.

```
select distinct
    output as pid,
    'how' as class
from how
where region=any(%s)
```

Listing 4.11: SQL Query zur Auswahl der How-Provenance in `get_involved_shape_ids()`

4.1.5.3 Regionen und Tabellen-Items ausgewählt

In diesem Fall wird wie bei Fall 4.1.5.1 vorgegangen. Nur die Funktion `getCssClasses(pids, rids)` aus `get_provenance()` geht etwas anders vor: Da diesmal `pids` und `rids` übergeben wurden, filtert die Funktion die gefundenen Data Provenances nach den Regionen. Es wird also nur der Schnitt der Data Provenances aus den Regionen und den Tabellen-Items gewählt, siehe Abbildung 4.5.

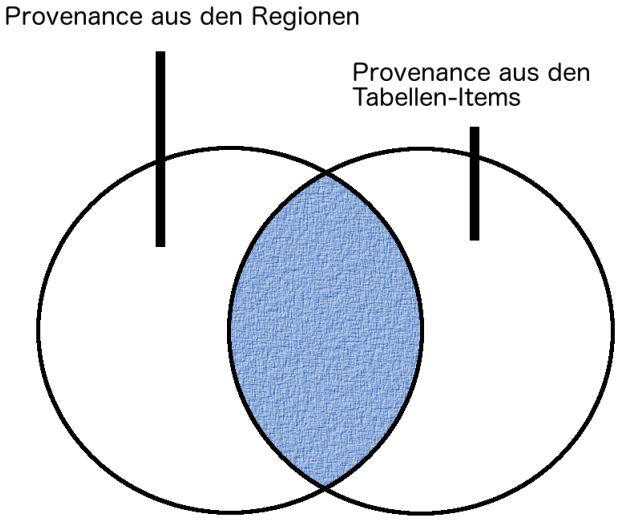

Abbildung 4.5: Schnitt der Data Provenances

4.1.6 Verwalten der Queries

4.1.6.1 Query-Auswahlliste

Das Aufrufen von `/db/get_prov_switch_list` liest alle Einträge aus der query-info.json und erstellt daraus eine Liste mit den Ordnernamen. Diese wird zurückgeliefert.

4.1.6.2 Analyse einer SQL Query

Beim Aufruf von `/db/analyze_sql_query` muss eine SQL Query und ein Name für diese übergeben werden. Die vollständige Funktion ist im Anhang in Kapitel 6.2 zu finden.
Zuerst wird ein Ordner mit dem übergebenen Namen im Code-Verzeichnis erstellt und die übergebene SQL Query in diesem Ordner als Datei gespeichert. Der

Dateiname wird aus `CONFIG['inputQueryName']` übernommen.

Als nächstes wird die SQL Query testweise ausgeführt und bei Fehlern wird der Output abgefangen, Zeilenumbrüche mit `<br>` ersetzt und die rechte Struktur aus Abbildung 4.10 zurückgegeben.

Verlief die SQL Query fehlerlos, wird sie mit dem Alias `tokl`, das den KL Compiler ausführt, nach KL übersetzt und unter `CONFIG['klQueryName']` abgespeichert.

Hat auch das funktioniert, so wird eine Datenbank erstellt. Der Datenbankname ist die Konkatenation des übergebenen Query-Namens und `CONFIG['dbSuffix']`. Auf der KL Query wird darauf die Provenance-Analyse mit dem Alias `prov` gestartet. Schlägt die Analyse fehl, wird wieder der Output abgefangen und als rechte Struktur in Abbildung 4.10 zurückgegeben.

Hat die Analyse aber erfolgreich, wird `add_entry_to_query_info(name, database)` ausgeführt, was die neue Query in `QUERY_INFO` einträgt und in query-info.json abspeichert. Jetzt wird der Query-Name als die linke Struktur in Abbildung 4.10 zurückgegeben.

4.1.6.3 Löschen einer SQL Query

Über den Aufruf `/db/delete_sql_query` wird eine SQL Query mit einem übergebenen Index gelöscht. Dabei wird das komplette Verzeichnis in code entfernt, die zugehörige Datenbank gelöscht und über `remove_entry_from_query_info(db_index)` die Query aus `QUERY_INFO` entfernt, sowie die query-info.json aktualisiert.

4.2 Client

Ein Browser dient als Client. Er wertet den JavaScript-Code aus und kommuniziert mittels Ajax (Kapitel 3.3.2) mit dem Server. Das Grund-Design-Pattern des Clients ist in Kapitel 3.3.1 beschrieben.

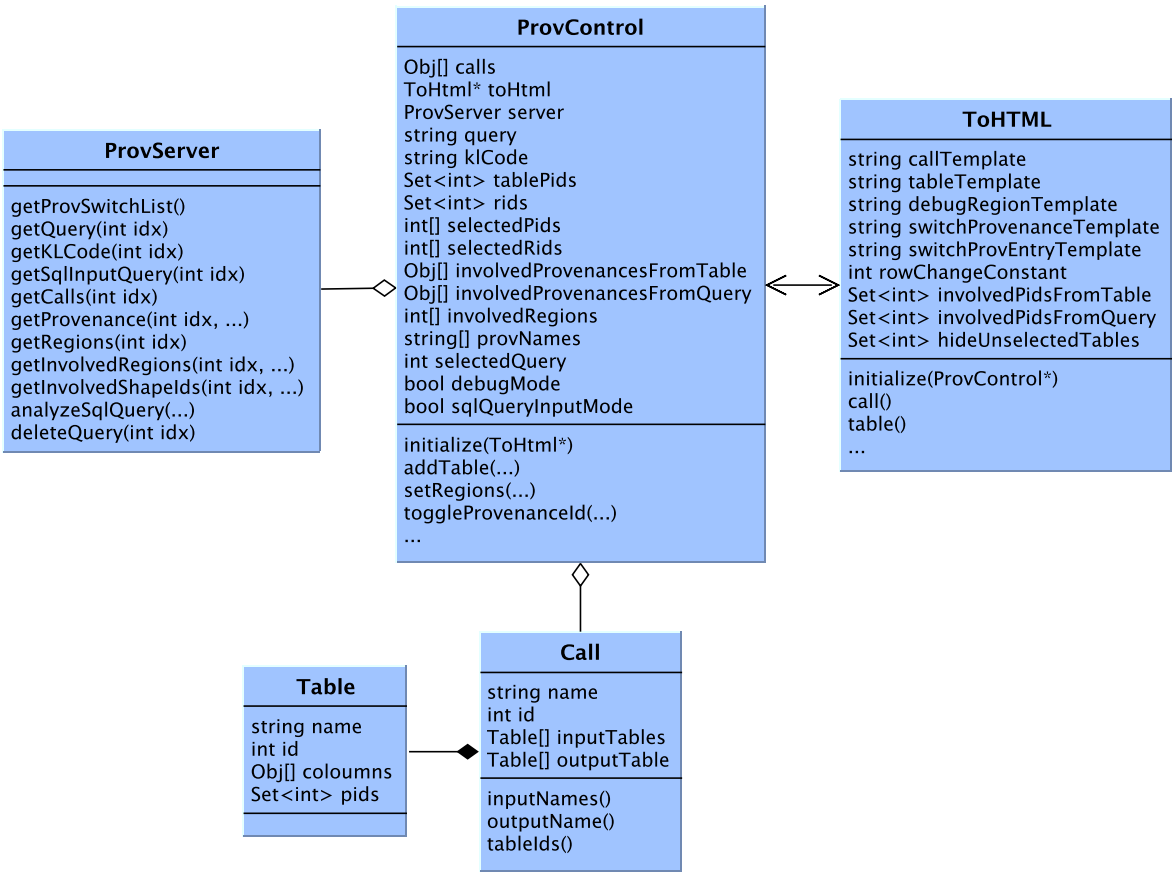

Abbildung 4.6: Unvollständiges Klassendiagramm des Clients

Es gibt die Klassen `ProvServer`, `ProvControl`, `Call`, `Table` und `ToHtml`.
`ProvControl` verwaltet beliebig viele Instanzen der Klasse `Call`, das wiederum beliebig viele Instanzen von `Table` beinhalten kann. Betrachtet man die Attribute von `ProvControl`, so sieht man, wie das Model mit dem Controller verschmilzt. Dadurch unterscheidet sich das verwendete Design-Pattern leicht von dem in Kapitel 3.3.1

beschriebenen. ToHtml ist die View und regelt die User-Interaktion.

```
$(document).ready(function() {
     var provControl = new ProvControl();
     var htmlVisualizer = new ToHtml();

     htmlVisualizer.initialize(provControl);
     provControl.initialize(htmlVisualizer);
});
```

Listing 4.12: Main-Funktion des Clients

In der Datei 4.12 main.js werden die Klassen ToHtml und ProvControl instanziiert und mit Pointern auf das jeweilige andere Objekt initialisiert.

Bevor ich die genaue Beschreibung des Clients behandle, möchte ich auf die Klasse ProvServer eingehen, die alle Ajax-Aufrufe bereitstellt und damit die Kommunikation mit dem Server regelt.

4.2.1 ProvServer

Um die Kommunikation mit dem Server möglichst transparent zu halten, habe ich die Klasse ProvServer eingeführt. Diese bündelt alle Ajax-Requests an die HTTP API und stellt sie nach außen als einfache Funktionen dar.

In Tabelle 4.2 fällt sogleich ein wiederkehrender Parameter auf: dbIndex. Da der Server zustandslos ist, brauchen wir u.a. zur Auswahl der Datenbank einen Index, der serverseitig auf einen Datenbanknamen abgebildet wird. Analog gilt das für die annotierte und eingegebene SQL Query, sowie für die KL Query. Das ermöglicht mehrere SQL Queries inklusive deren Analyseergebnisse zu verwalten.

Funktion	Parameter	Server API Call
getQuerySwitchList()	-	/db/get_prov_switch_list
getAnnotatedSqlQuery()	int dbIndex	/db/get_sql_query
getKLQuery()	int dbIndex	/db/get_kl_code
getInputSqlQuery()	int dbIndex	db/get_sql_input_query
getCalls()	int dbIndex	/db/get_calls /db/get_var
getProvenance()	int dbIndex int[] pids int[] rids	/db/get_provenance
getRegions()	int dbIndex	/db/get_regions
getInvolvedRegions()	int dbIndex int[] pids	/db/get_involved_regions
getInvolvedShapeIds()	int dbIndex int[] rids	/db/get_involved_shape_ids
deleteQuery()	int dbIndex	/db/delete_sql_query
analyzeSqlQuery()	string name string query	/db/analyze_sql_query

Tabelle 4.2: Funktionen der Klasse `ProvServer`

4.2.1.1 getProvSwitchList()

Ruft `provControl.setProvSwitchList(list)` auf und übergibt als `list` ein sortiertes String-Array, das die Namen der verschiedenen SQL Queries enthält. Zugegriffen wird auf die Queries mittels `dbIndex`, welches sich auf den Index der übergebenen Liste bezieht. Als Nächstes wird `provControl.loadSelectedQuery()` aufgerufen.

4.2.1.2 getAnnotatedSqlQuery(dbIndex)

Ruft `provControl.setAnnotatedSqlQuery(queryString)` auf und übergibt die annotierte SQL Query als String.

4.2.1.3 getKLQuery(dbIndex)

Ruft `provControl.setKlQuery(queryString)` auf und übergibt die KL Query als String.

4.2.1.4 getInputSqlQuery(dbIndex)

Ruft `provControl.setAndOpenInputSqlQuery(queryString)` auf und übergibt die eingegebene SQL Query als String.

4.2.1.5 getCalls(dbIndex)

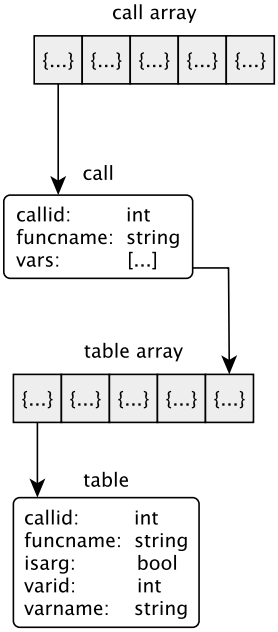

Abbildung 4.7: Struktur des Rückgabe-JSON-Objekts von `/db/get_calls`

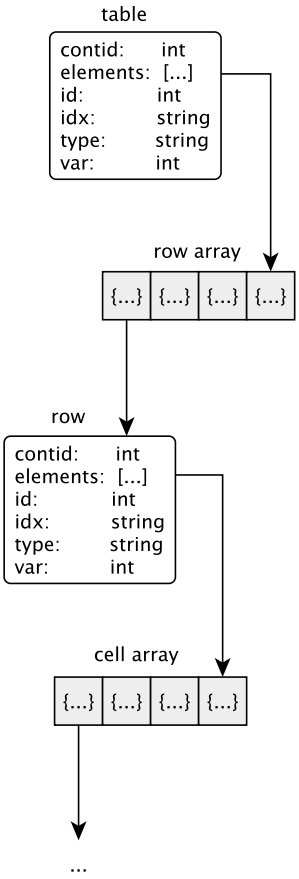

Abbildung 4.8: Struktur des Rückgabe-JSON-Objekts von `/db/get_var`

Startet für jedes erhaltene `table`-Objekt (siehe Abbildung 4.7) eine Anfrage an `/db/get_var`. Die Parameter sind `dbIndex` und die Tabellen-ID `varid`. Die erhaltene Tabelle besteht aus einer verschachtelten Struktur, wie in Abbildung 4.8 zu sehen. Ruft `provControl.addTable(tableInfo, table)` auf und übergibt das `tableInfo`-Objekt (Abbildung 4.7: table), sowie das `table`-Objekt (Abbildung 4.8).

4.2.1.6 `getProvenance(dbIndex, pids, rids)`

`pids` ist ein Integer-Array, welches die ID jedes markierten atomaren Elements aller Tabellen enthält. `rids` ist ein Array vom gleichen Typ, welches die IDs aller markierten Regionen im Query-Text enthält.

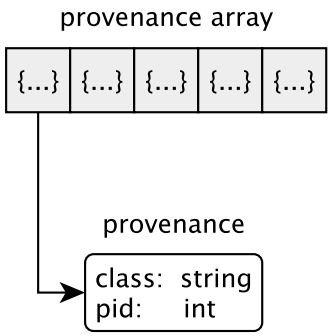

Abbildung 4.9: Struktur des Rückgabe-JSON-Objekts von `/db/get_provenance`

Als Ergebnis bekommen wir ein Array aus Objekten, wie in Abbildung 4.9 zu sehen. Jedes Objekt besteht aus der CSS-Klasse `class` (`"where"`, `"why"` oder `"mixed"`) und der `pid` des betroffenen atomaren Items.
Ruft `provControl.setProvFromTable(provenances)` mit dem erhaltenen Array auf.

4.2.1.7 `getRegions(dbIndex)`

Ruft `provControl.setRegions(regions)` mit dem Integer-Array `regions` auf, in welchem alle verfügbaren Regionen in der annotierten Query aufgelistet sind.

4.2.1.8 `getInvolvedRegions(dbIndex, pids)`

Ruft `provControl.setInvolvedRegions(regions)` mit dem Integer-Array `regions` auf, welches alle durch `pids` betroffenen Regionen beinhaltet.

4.2.1.9 `getInvolvedShapeIds(dbIndex, rids)`

Ruft `provControl.setProvFromQuery(provenances)` auf. `provenances` ist ein Array aus Objekten und hat die gleiche Struktur, die in Abbildung 4.9 zu sehen ist. Die `provenances` sind diesmal jedoch nach den `rids` gefiltert und als `class` wird immer `"how"` angegeben.

4.2.1.10 deleteQuery(dbIndex)

Nach dem serverseitigen Löschen der Query wird `provControl.removeQuerySwitchEntry(dbIndex)` aufgerufen.

4.2.1.11 analyzeSqlQuery(name, query)

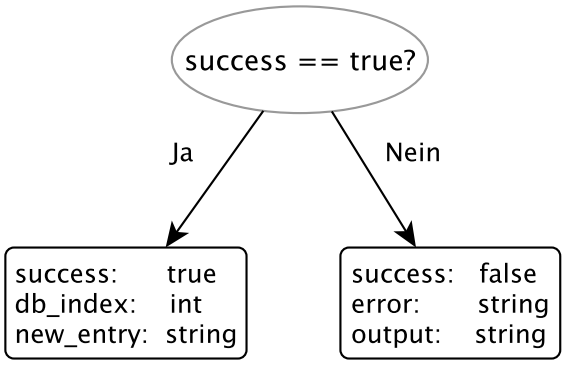

Abbildung 4.10: Struktur des Rückgabe-JSON-Objekts von `/db/analyze_sql_query`

War die Analyse erfolgreich, so wird die linke Struktur in Abbildung 4.10 zurückgegeben und `provControl.addQuerySwitchEntry(result.new_entry, result.db_index)` aufgerufen. Andernfalls die Rechte.

```
let resultText = result.success ? "Analysis successful!" : result.output;
provControl.giveAnalyzationResult(result.success, resultText);
provControl.stopWaitingforAnalyzation();
```

Listing 4.13: Code-Fragment, das unabhängig von der Analyse ausgeführt wird

4.2.2 Übersicht der Visualisierung

SQL QUERY

```
WITH
weather(day, weekday, precipitation, temperature) AS (
  SELECT d.day, d.weekday,
       SUM(d.precip) OVER
              (ORDER BY d.day
               ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS precipitation,
       MIN(d.temp) OVER
              (ORDER BY d.day
               ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS temperature
  FROM daily AS d
)
fair_cloudy(weekday, condition) AS (
  SELECT w.weekday,
       CASE WHEN w.precipitation <= 600 OR w.temperature >= 15
                    THEN '☀'
                    ELSE '☁'
            END
     AS condition
  FROM weather AS w
)
SELECT f.weekday,
          COUNT(*) FILTER (WHERE f.condition = '☀') AS fair,
          f.weekday IN ('Sat', 'Sun') AS weekend
FROM fair_cloudy AS f
GROUP BY f.weekday
```

1

2

3

main(daily) → res

daily

T	day	precip	temp	weekday
0	1	800.0	14.0	Fri
1	2	300.0	16.0	Sat
2	3	100.0	16.0	Sun
3	4	200.0	20.0	Mon
4	5	300.0	20.0	Tue
5	6	120.0	18.0	Wed
6	7	0.0	14.0	Thu
7	8	0.0	10.0	Fri
8	9	500.0	12.0	Sat
9	10	300.0	14.0	Sun
10	11	80.0	15.0	Mon
11	12	0.0	17.0	Tue

res

T	fine	weekday	weekend
0	1	Fri	False
1	2	Mon	False
2	0	Sat	True
3	1	Sun	True
4	1	Thu	False
5	2	Tue	False
6	1	Wed	False

Abbildung 4.11: Interface der Visualisierung

Die Visualisierung besteht grundsätzlich aus:

1. Steuer- und Infobereich
2. SQL Query-Bereich
3. Call-Bereich

Zusätzlich gibt es noch einen optionalen SQL Query-Eingabebereich und einen Debug-Regionenbereich.

4.2.3 Steuer- und Infobereich

Der Steuerbereich befindet sich links und der Infobereich rechts oben.

4.2.3.1 Query Switch

Abbildung 4.12: Query Switch-Icon

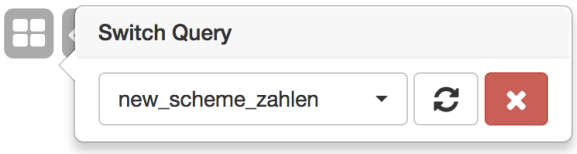

Abbildung 4.13: Query Switch-Popover

Das **Query Switch**-Icon 4.12 öffnet mit einem Klick das Popover 4.13. Da das Standard-Klickenvent eines Bootstrap-Popovers keine Kombination aus Öffnen und Schließen beim Klick auf das Icon, sowie Schließen beim Klicken in einen anderen Bereich anbietet, wurden zwei eigene Klickevents definiert. Das Eine toggelt bei Klick auf das Query Switch-Icon das Popover. Das Andere prüft global folgende Bedingung bei jedem Klick in der Visualisierung:

```
$(event.target).data('toggle') !== 'popover'  && // Not on popover icon
$(event.target).parents('.popover.in').length === 0 // Outside popover
```

Listing 4.14: Bedingung bei globalem Klickevent

Ist der Klick nicht auf dem Popover oder dem entsprechenden Icon, so wird `addClass('hidden')` ausgeführt. Die Funktion zeigt eine weitere Besonderheit beim Umgang mit diesem Popover: Es wird nie *zerstört*. Mit `addClass('hidden')` wird das Popover versteckt und mit `removeClass('hidden')` wieder angezeigt.

Für den üblichen Gebrauch bietet Popover dafür `popover('show')` und `popover('hide')`. Dabei wird das Popover jedesmal neu kontruiert und wieder zerstört. Für den Zweck der Auswahl ist das aber kontraproduktiv, da sie kein statisches Element ist. Die Auswahl soll nicht verworfen werden und ein anschließendes Hinzufügen von `selected` zum entsprechenden `<option>`-Tag der Liste brachte nicht nur zusätzlichen Overhead durch das eigenständige Verwalten der Auswahl, die Anzeige zeigte auch jedesmal kurz die erste Auswahlmöglichkeit, bevor es zum tatsächlich ausgewählten Element wechselte. Also habe ich mich entschieden, die Sichtbarkeit des Popovers mittels CSS-Eigenschaft `display` zu managen.

Für die Dropdown-Liste wurde bootstrap-select verwendet, welches unter anderem für die Elemente eine Suche ermöglicht. Dafür wurde `selectpicker()` auf das Select-Element im Popover angewandt.

4.2.3.2 Query-Eingabe

Abbildung 4.14: Query-Eingabe-Icon

Mit einem Klick auf das **Query-Eingabe**-Icon 4.14 oder mit dem Shortcut **ctrl+e** wird die Funktion `provControl.toggleSqlQueryInput()` aufgerufen, welche die interaktive Eingabe einer SQL Query toggelt.

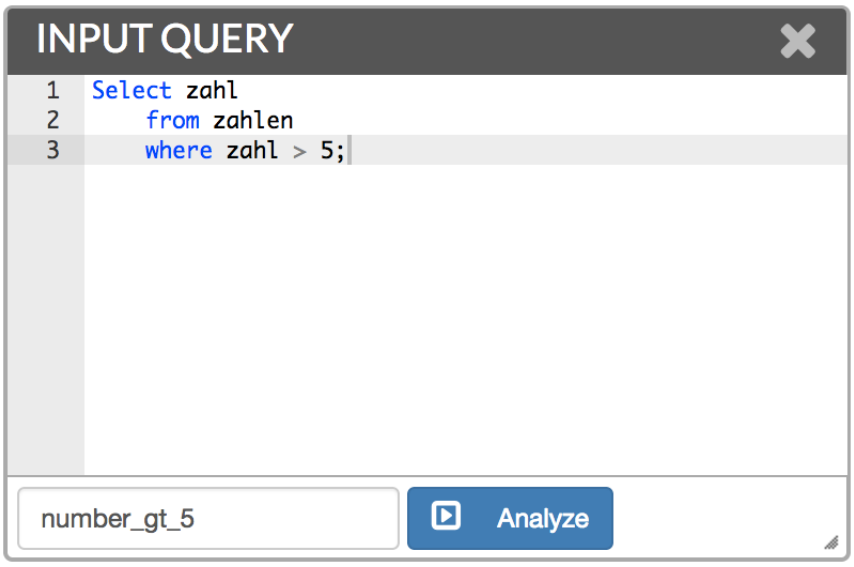

Abbildung 4.15: Query-Eingabebereich

Der Query-Eingabebereich besteht aus dem SQL-Editor, einem Feld zur Namenseingabe und einem Analysebutton. Der Editor ist Ace 3.2.1.9.

```
let editor = ace.edit('sql-editor');
editor.setTheme('ace/theme/sqlserver');
editor.getSession().setMode('ace/mode/sql');
editor.setAutoScrollEditorIntoView(true); // Inline scrolling
editor.$blockScrolling = Infinity;        // Avoid warning
```

Listing 4.15: Initialisierungs-Code von Ace

Mit Zeilenangaben, Syntax-Highlighting und Scrolling direkt im eingebetteten Element bietet Ace eine angenehme Möglichkeit zur Code-Eingabe.
Das Textfeld dient der Namesvergabe der zu analysierenden SQL Query. Der Analysebutton leitet die Query per `server.analyzeSqlQuery(queryName, query)` an den Server weiter. Nach der Analyse wird visuelles Feedback an den Nutzer gegeben. War die Query korrekt und die Analyse hat funktioniert, dann gibt es die Rückmeldung in Abbildung 4.16.

Successful! Analysis successful! ×

Abbildung 4.16: Positives Feedback der Query-Analyse

Wenn ein Syntaxfehler in der eingegeben SQL Query vorliegt oder die angesprochenen Tabellen / Spalten nicht existieren, so gibt es das Feedback in Abbildung 4.17. Der gefundene Fehler in der Query wird von der PSQL-Ausgabe an den User weitergeleitet.

Failure! psql:input.sql:3: ERROR: column "zahl_" does not exist ×
LINE 1: Select zahl_
^
HINT: Perhaps you meant to reference the column "zahlen.zahl".

Abbildung 4.17: Negatives Feedback der Query-Analyse

4.2.3.3 Debug-Modus

Während der Entwicklung der Visualisierung war es von Nöten, auch Low-Level-Informationen leicht zugänglich zu halten. Dazu zählen *Regionen-IDs*, *Shape-IDs* (bzw. *Item-IDs*) und die *Call-IDs*.

Abbildung 4.18: Debug-Icon

Ein Klick auf das Debug-Icon 4.18 oder das Shortcut **ctrl+d** togglet den **Debug-Modus**. Beim Aktivieren werden folgende Aktionen ausgeführt:
Die Regionen-IDs werden beim Hovern über die Query in einem Tooltip angezeigt. Zusätzlich wird ein eigener Bereich für die Regionen-IDs 4.19 eingeblendet.

REGIONS
0 1 2 3 4 5 6 7 8

Abbildung 4.19: Regionenbereich

Die Shape-IDs werden auch beim Hovern über das entsprechende Tabellen-Element angezeigt. Die Call-IDs werden in der oberen rechten Ecke eines jeden Call-Bereichs dargestellt.

4.2.3.4 Provenance-Informationen

Abbildung 4.20: Provenance-Info-Icon

Wird das **Provenance-Info**-Icon gehovert, so öffnet sich ein Popover (Abbildung 4.21), das die Farben in den Tabellen und der Query den Provenance-Arten zuordnet.

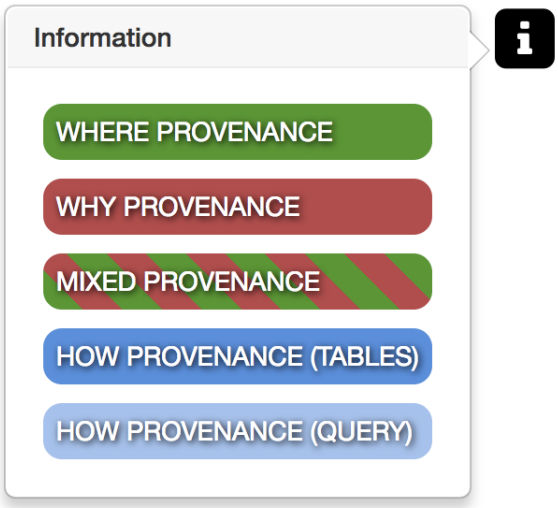

Abbildung 4.21: Provenance-Info-Popover

4.2.3.5 Steuerungsinformationen

Abbildung 4.22: Steuerungsinfo-Icon

Wird das **Steuerungsinfo**-Icon gehovert, so wird ein Popover (Abbildung 4.23) angezeigt, dass die verfügbaren Shortcuts zeigt und die Tabellenfunktionen erklärt.

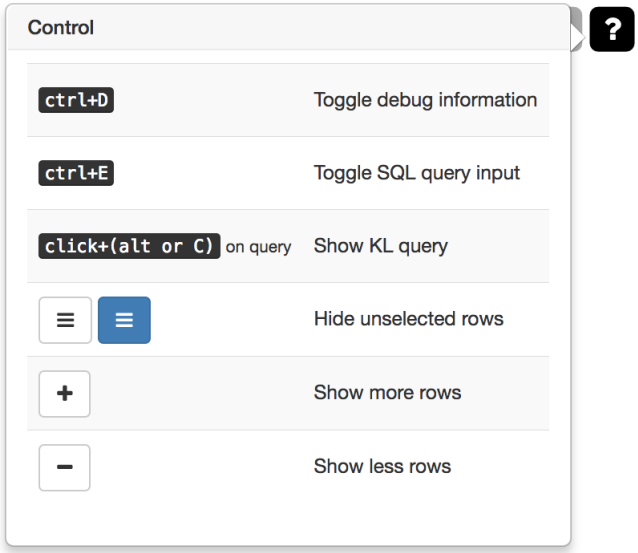

Abbildung 4.23: Steurungsinfo-Popover

4.2.4 SQL Query-Bereich

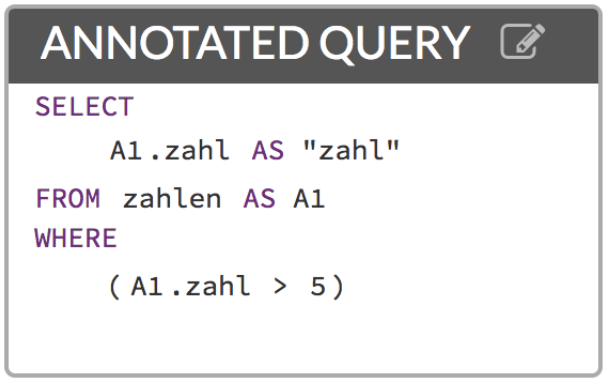

Abbildung 4.24: SQL Query-Bereich

Der SQL Query-Bereich zeigt die annotierte SQL Query, die mittels `server.getAnnotatedSqlQuery()` (Kapitel 4.2.1.2) per POST-Request geladen wird. Neben der Annotierung sind Regionen in der SQL Query hinterlegt. Außerdem gibt es ein sehr simples Syntax-Highlighting der Keywords in SQL. In der oberen rechten Ecke kann man die ursprünglich eingegebene SQL Query inklusive Namen

der Query in den Editor laden.

4.2.4.1 Formatierung der Query

Vom Server wird eine annotierte SQL Query geliefert, wie in Listing 4.16 zu sehen.

```
[R8|main:SELECT
     [R1|main:[R2|main:A1].zahl] AS "zahl"
FROM [R3|main:zahlen] AS A1
WHERE
     [R4|main:([R5|main:[R6|main:A1].zahl] > [R7|main:5])]
]
```

Listing 4.16: Annotierte SQL Query

Eine Region wird mit `[Rx|funcName: sqlText]` gekennzeichnet.

- `x` ist ein Integer und steht für die Region.
- `funcName` ist ein String und steht für den Funktionsnamen, in der die Region existiert. Für diese Arbeit hat `funcName` aber keine Bedeutung.
- `sqlCode` ist ein String und stellt den SQL-Text dar, den die Region umfasst.

Regionen können beliebig tief verschachtelt werden.

```
let formatRegions = function(queryString) {
     let regex = /\[R(\d+)\|([a-zA-Z][a-zA-Z\d+]*):([^\]]*)\]/g;
     let replace = '<span id="sql-rid-$1" class="sql-region sql-region-border"
                    data-rid=$1 data-function-name=$2 title="Region: $1">$3</span>';
     while(queryString.search(regex) > -1) {
          queryString = queryString.replace(regex, replace);
     }
     return queryString;
}
```

Listing 4.17: Ersetzen der Annotation durch Span-Elemente

Die Funktion `formatRegions(queryString)` in Listing 4.17 sucht alle annotierten Regionen und ersetzt sie durch ein Span-Element, das `<data>`-Tags für die Region

und den Funktionsnamen nutzt. Für den Tooltip im Debug-Modus wird der `<title>`-Tag gesetzt. Zusätzlich bekommt jedes Element eine eindeutige `<id>` und die Klasse `sql-region` und `sql-region-border` zugewiesen.

Ähnlich funktioniert das Syntax-Highlighting. Per Regex wird jedes SQL Keyword mit einem Span-Element ersetzt. Dabei wird die Klasse `sql-syntax` gesetzt, welche die Keywords einfärbt.

4.2.4.2 Darstellung der How-Provenance

Die How-Provenance kann in zwei Richtungen gezeigt werden. Einmal, indem in der annotierten SQL Query eine oder mehrere Regionen ausgewählt werden. Dann werden alle Where- und Why-Provenances in den Tabellen markiert, die von der/den Region(en) betroffen sind. Dafür wird die Funktion 4.2.1.9 `server.getInvolvedShapeIds(dbIndex, rids)` genutzt.
Außerdem, indem Tabellen-Items markiert werden. Dadurch werden alle Regionen markiert, die diese Items betreffen. Siehe Abbildung 4.25. Dafür wird die Funktion 4.2.1.8 `server.getInvolvedRegions(dbIndex, pids)` genutzt.

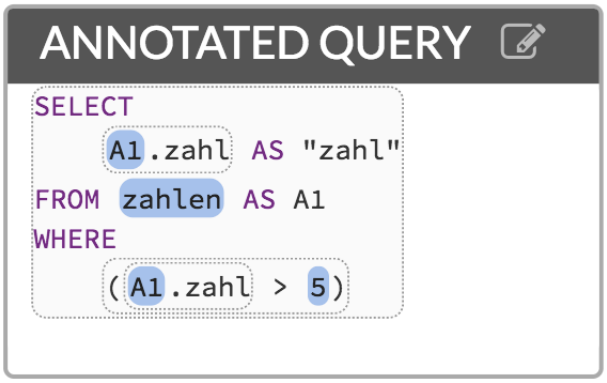

Abbildung 4.25: SQL Query-Bereich mit markierten Regionen

Blau eingefärbt werden nur die Blatt-Regionen, also die innersten. Der Rest bekommt eine gestrichtelte Outline und einen grauen Hintergrund. Damit wirken die markierten Regionen aufgeräumter, als wenn alles blau markiert ist.

Sind sowohl Regionen, als auch Tabellen-Items markiert, so werden nur Where- und Why-Provenances in den Tabellen markiert, die sowohl im Zusammenhang mit den ausgewählten Items stehen, als auch von den Regionen betroffen sind.

Einfach gesagt wird also nach den Regionen gefiltert. Dafür wird die Funktion 4.2.1.6 `server.getProvenance(dbIndex, pids, rids)` genutzt.

4.2.4.3 KL Query

Ein weiteres Feature ist das Einblenden der Query in Kernel Language, die mittels 4.2.1.3 `server.getKLQuery()` per POST-Request geladen wird. Auch diese Query besitzt Regionen. Wird mit **alt** oder **c** auf eine Region in der annotierten SQL Query geklickt, so wird die KL Query eingeblendet und zur gleichen Region gesprungen.

Kernel Language query

```
24   Region: 3;
25   skip;
26   regionResult2 = zahlen;
27   RegionEnd: 3;
28   data = [];
29   dt = {};
30   skip;
31   for klA1 in regionResult2 do
32      skip;
33      update(dt, "klA1", klA1);
34      Region: 4;
35      Region: 5;
36      Region: 6;
37      regionResult5 = dt{"klA1"};
38      RegionEnd: 6;
39      regionResult4 = regionResult5{"zahl"};
40      RegionEnd: 5;
41      Region: 7;
42      skip;
43      regionResult6 = 5;
44      RegionEnd: 7;
45      regionResult3 = (regionResult4
46                       >
47                       regionResult6);
```

Abbildung 4.26: KL Query-Dialog

Die vollständige KL Query befindet sich im Anhang in Kapitel 6.3.

```
let regionAnchors = function(klQuery) {
     let regexRegion = /Region<span class="token punctuation">:<\/span>
                       <span class="token number">(\d+)<\/span>/g;
     let regexRegionEnd = /RegionEnd<span class="token punctuation">:<\/span>
                          <span class="token number">(\d+)<\/span>/g;
     let replaceRegion = '<span id="kl-query-region-$1"
                          class="kl-query-region">Region: $1</span>';
     let replaceRegionEnd = '<span class="kl-query-region-end">
                             RegionEnd: $1</span>';

     klQuery = klQuery.replace(regexRegion, replaceRegion);
     return klQuery.replace(regexRegionEnd, replaceRegionEnd);
}
```

Listing 4.18: Highlighten der Regions in der KL Query

Die Regulären Ausdrücke in Listing 4.18 `regexRegion` und `regexRegionEnd` sind interessant. Um Doppelpunkte wird ein Span-Element erwartet. Ebenso um die Dezimalzahl. Das liegt daran, dass die KL Query vorher schon mit Prism gerendert wurde und deshalb schon dessen Annotation in der Query vorhanden ist.
In keinem Syntax-Highlighter gab es eine Sprachdefinition für KL. Eine Zeit lang habe ich deshalb mit dem Gedanken gespielt, einen eigenen Parser dafür zu schreiben. Da der Fokus dieser Arbeit aber nicht auf perfektem Syntax-Highlighting liegt und ein Parser zudem recht zeitaufwendig ist, habe ich nach Alternativen Ausschau gehalten. highlight.js war die erste Option und schien auf den ersten Blick perfekt. Als ich jedoch eine Sprachdefinition für KL schreiben sollte, wurde es komplizierter. Die Dokumentation war gut, aber die Syntax recht komplex. Ohne weiter einzutauchen suchte ich weiter und fand Prism. Hier war die Syntax zur Sprachdefinition sehr simpel. Und damit fiel auch die Entscheidung.

Als Grundlage habe ich die Python-Sprachdefinition genommen, da KL der Syntax von Python sehr ähnlich ist. Daraus erarbeitete ich die Sprachdefinition für

KL. Fertig sieht diese nun folgendermaßen aus:

```
Prism.languages.kl = {
     comment: {
          pattern: /(^|[^\\])#.*/,
          lookbehind: !0
     },
     string: {
          pattern: /("|')(?:\\\\|\\?[^\\\r\n])*?\1/,
          greedy: !0
     },
     'function': {
          pattern: /((?:^|\s)def[ \t]+)[a-zA-Z_][a-zA-Z0-9_]*(?=\()/g,
          lookbehind: !0
     },
     keyword: /\b(var|type|int|float|string|if|then|else|fi|while|do|od|
                for|in|append|update|skip|length|keys|contains|def|return|end)\b/,
     'boolean': /\b(true|false)\b/,
     'number': /\b-?(?:0x[\da-f]+|\d*\.?\d+(?:e[+-]?\d+)?)\b/i,
     'operator': /\+|\-|\*|\\|\||<=?|>=?|==?|\b(and|or|not)\b/,
     punctuation: /[{}[\];(),.:]/
};
```

Listing 4.19: KL-Sprachdefinition für Prism

4.2.4.4 Bearbeiten einer vorhandenen Query

Ein Klick auf das Bearbeiten-Icon in der oberen rechten Ecke führt `provControl.editSqlQuery()` aus. `ProvControl` fordert vom Server mittels POST-Request 4.2.1.4 `server.getInputSqlQuery(this.selectedQuery)` die eingegebene SQL Query an. Sobald diese da ist, wird `provControl.setAndOpenInputSqlQuery(queryString)` aufgerufen, womit die SQL Query im Editor geladen, sowie der Name der Query aus dem `queryNames`-String-Array gelesen und gesetzt wird. Letzendlich wird die SQL Query-Eingabe geöffnet.

4.2.5 Call-Bereich

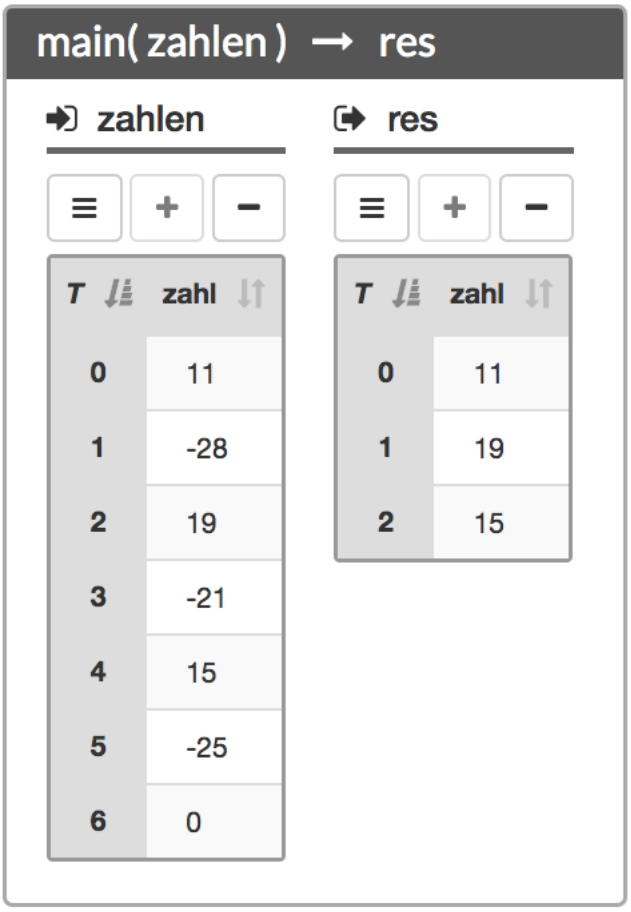

Abbildung 4.27: Tabellen innerhalb eines Calls

Der Call-Bereich kann eine beliebige Anzahl an Calls umfassen. In Abbildung 4.27 ist ein Call zu sehen. Die Tabellen sind in Ein- und Ausgabe-Tabellen gegliedert. Jeder Call hat eine beliebige Menge an Eingangs-Tabellen, aber immer nur eine Ausgabe-Tabelle. Jede Tabelle hat als erste Spalte sein Tupel. Das Tupel ordnet jeder Zeile einer Tabelle einen einzigartigen Wert $n \in \mathbb{N}_0$ zu und macht sie dadurch eindeutig. Außerdem gibt es über jeder Tabelle drei Buttons. Der Erste zeigt nur vom User markierte oder die mit Provenance versehenen Zeilen in einer Tabelle an. Der Zweite und Dritte zeigen mehr bzw. weniger Gesamtzeilen der Tabelle an.

4.2.5.1 Erzeugung eines Calls

Zu Beginn existiert nur ein Span-Element mit der `<id> call-area`. Dann wird die Funktion `provControl.loadSelectedQuery()` aufgerufen. Existiert mindestens eine Query, so wird `provControl.initializeQuery(this.selectedQuery)` ausgeführt, welche erst die Oberfläche zurücksetzt und dann asynchron die Calls, die Regionen, die annotierte SQL Query und die KL Query vom Server anfordert. Im Folgenden behandle ich die Calls.
Mittels 4.2.1.5 `server.getCalls(dbIndex)` werden die Tabellen aller Calls angefordert. Diese werden per 4.20 `provControl.addTable(info, table)` der Visualisierung hinzugefügt.

```
addTable(info, table) {
     let callId = info.callid;
     let callName = info.funcname;
     let isInputTable = info.isarg; // False = output table
     let tableId = info.varid;
     let tableName = info.varname;

     // Add call for callId if not existing
     if (this.calls[callId] === undefined) {
          this.calls[callId] = new Call(callId, callName);
          this.toHtml.call(callId, callName, this.debugMode);
     }
     let newTable = this.calls[callId].addTable(tableId, tableName,
                                                table, isInputTable);
     let inputNames = this.calls[callId].inputNames();
     let outputName = this.calls[callId].outputName();
     this.tablePids = this.tablePids.union(newTable.pids);

     this.toHtml.callName(callId, callName, inputNames, outputName);
     this.toHtml.table(tableId, tableName, newTable, callId, isInputTable);
}
```

Listing 4.20: Funktion `addTable(info, table)` der Klasse ProvControl

Im `info`-Objekt stehen unter anderem die ID des Calls und dessen Name. Diese werden genutzt, um ein `call`-Objekt im `calls`-Array zu erzeugen und auszugeben, wenn noch nicht vorhanden. Die Ausgabe in der Klasse `ToHtml` erfolgt mit Hilfe

eines Templates 4.21, das mit mustache.js gerendert wird.

```
callTemplate =
     '<div id="call-id-{{callId}}" class="call area"> ' +
     '<div class="call-heading dark-heading draggable"> ' +
     '<span class="call-heading-name">{{callName}}</span> ' +
     '<span class="call-heading-debug-id debug-info hidden"> ' +
     '{{callId}}</span></div> ' +
     '<div class="table-area"></div> ' +
     '</div> ';
```

Listing 4.21: mustache.js Template für einen Call

Darüber hinaus brauchen wir Daten in der Form `{callId: int, callName: string}`, mit denen das Template gerendert werden kann. Die Daten werden in doppelt geschweiften Klammern im Template `{{}}` eingefügt. Der fertig gerenderte Call wird dann in den Call-Bereich eingefügt.

Über die ID wird darauf der Call ausgewählt, dem eine Tabelle mit `addTable(id, name, table, isInput)` hinzugefügt werden soll. Die Funktion der Klasse `Call` erzeugt ein `table`-Objekt. `Call` speichert die Tabelle in seinem `inputTables`- oder `outputTables`-Array und gibt das erzeugte `table`-Objekt auch gleich an ProvControl zurück. Doch bevor ich die Ausgabe bespreche, möchte ich genauer auf die Erzeugung des `table`-Objekts eingehen.

Die Klasse `Table` hat keine Funktionen, sondern nur die Attribute `name`, `id`, `coloumns` und `pids`. Die Attribute werden direkt bei der Erzeugung im Konstruktor berechnet. Dazu wird zuerst jede `row` 4.8 geparst und dann ausgewertet. Die private Funktion im Konstruktor `parseRow(row)` (im Anhang, Kapitel 6.4) nimmt ein 4.8 `table`-Objekt und parst es rekursiv.

- Bei atomaren Werten wird die Rekursion abgebrochen und ein Objekt mit der Struktur `{tupleVal: int, tupleId: int, colName: string, id: int, value: atomValue}` zurückgegeben.
- Bei Listen, Dictionaries und Rows (Dictionary mit festen Feldern) werden die Werte in einer Schleife weiter geparst.
- Liegt eine Tabelle vor, so wird ein neues `table`-Objekt erstellt. Dieser Fall

funktioniert aber nur in der Theorie, dargestellt werden kann das geschachtelte `table`-Objekt wegen Einschränkungen von mustache.js nicht.

Während des Parsens wird bei atomaren Werten die `shapeId` zum Set `pids` hinzugefügt. Darüber können stehts alle vorhandenen Provenance-IDs der Tabelle eingesehen werden.
Die nächste private Funktion - `evaluateRowList(row)` - fügt ein Objekt mit der Struktur `{id: int, value: atomValue, tupleVal: int, tupleId: int}` dem `coloumn`-Dictionary hinzu. Der Key dazu ist `colName` aus `parseRow(row)`. Das wird für jeden atomaren Wert ausgeführt, wodurch sich in dem `coloumns`-Dictionary am Ende alle Spalten der Tabelle befinden. Jetzt haben wir eine fertige Tabelle.

Der Name im Header des Calls wird generiert und dargestellt. Erst dann kommt die Ausgabe der Tabelle. Dafür wurde, wie auch für die Calls, ein Template genutzt. Im Tabellen-Template 6.8 werden häufig Ausdrücke der Form `{{#name}}` ... `{{/name}}` genutzt. Das bedeut, dass `name` ein Array ist, über dessen Objekte iteriert wird. Aus den Objekten werden die Daten, wie schon beim Call, über ihren Schlüssel angesprochen. Damit sind auch beliebig tiefe, aber feste Verschachtelungen möglich. Ich hatte jedoch zuvor geschrieben, dass keine tiefer geschachtelten Strukturen möglich sind, wie beispielsweise eine verschachtelte Tabelle. Das stimmt aber nur zum Teil. Denn wenn in jeder Struktur eine veschachtelte Tabelle vorkommen würde, wäre das mit mustache.js machbar. Da die Strukturen einer Tabelle aber variieren und das Template vorher festgelegt werden muss, ist dies nicht ohne weiteres möglich. Das liegt auch daran, dass Mustache Schleifen erlaubt, aber keine Rekursion. Für das Rendern einer Tabelle werden neben dem Template auch Daten in einer Datenstruktur nach dem Schema in Abbildung 4.28 benötigt.

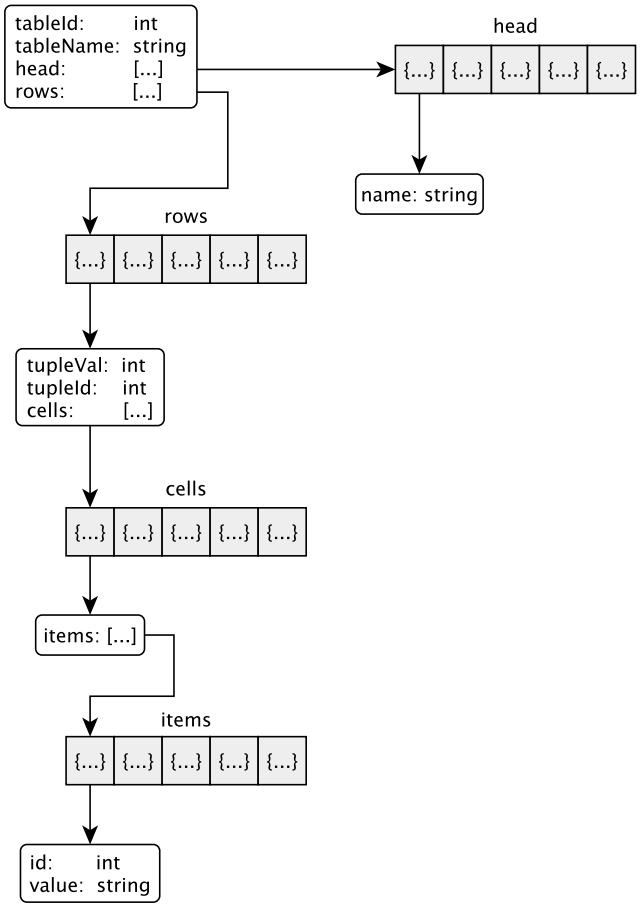

Abbildung 4.28: Datenstruktur für das Tabellen-Template

Diese Datenstruktur wird aus dem `table`-Objekt in der Funktion `toHtml.table(..)` berechnet. Nach dem Rendern wird die Tabelle dem `table-area`-Bereich im zugehörigen Call hinzugefügt und mit `DataTable()` zu einer DataTable transformiert.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Es wurde erfolgreich eine High-Level-Visualisierung für die Where-, Why-, und How-Provenance erstellt. Neben den Eingangs- und Ausgabe-Tabellen wird die annotierte SQL Query, generiert vom KL Compiler, angezeigt. Das Einblenden der schlussendlich analysierten KL Query ist ebenso möglich, wie interaktiv eine SQL Query einzugeben und analysieren zu lassen. Die Verwaltung mehrerer Queries ist nicht nur ein nettes Zusatzfeature, sondern auch überaus sinnvoll für den Multi-User-Betrieb.

5.2 Future Work

5.2.1 Nicht blockierende Aufrufe

Bottle blockiert, sobald es einen Aufruf bearbeitet, da es singlehreaded ist. Mit mehreren Threads könnte man dieses Problem lösen und die Responsiveness deutlich erhöhen.

5.2.2 Ersetzen von Mustache bei Tabellen

Mustache ist zum Rendern wohldefinierter Objekte gut geeignet. Bei Tabellen mit undefinierter Rekursionstiefe stößt ein Mutache-Template aber an seine Grenzen. Es wäre sinnvoll, Mustache hier durch ein anderes Templating-System zu ersetzen oder die Tabellen von *Hand* mit einer rekursiven Konkatenation zu erstellen.

5.2.3 Speichern der Queries

Auf dem Server wird in der config.json ein `inputQueryName`, ein `annotatedQueryName` und ein `klQueryName` definiert. Beim Analysieren einer SQL Query werden so die Namen der drei Queries gesetzt. Wenn nun einer der drei Namen in der config.json

geändert wird, funktionieren bisher analysierte Queries nicht mehr, da jetzt ein anderer Name erwartet wird. Ein besserer Ansatz wäre, die zugehörigen Dateinamen in der query-info.json zu speichern.

5.2.4 Wählen der Quell-Datenbank

Bisher ist die Datenbank, aus der für die Analyse gelesen werden kann, in der config.json unter `sourceDbName` festgelegt. Bei großen Datenmengen würde es Sinn ergeben, die Quell-Datenbank wählen zu können.

5.2.5 Tabellen-Abfrage vereinfachen

Um an alle Tabellen zu gelangen, werden etliche Anfragen an `/db/get_var` geschickt. Auch serverseitig werden rekursiv etliche Datenbankabfragen gestartet. Besser wäre es, nur eine Anfrage an den Server zu schicken, die call- und shape-Tabelle mit je einer SQL Query auszulesen und auf dem Server zu parsen. Damit könnten die Tabellen deutlich schneller geladen werden.

5.2.6 KL Highlighting

Prism unterliegt gewissen Einschränkungen. Da es mit Regulären Ausdrücken arbeitet, wird es zwangsläufig an bestimmten Fällen scheitern. Ein eigener Parser für KL würde Abhilfe schaffen.

6

Anhang

6.1 Definition der KL Grammatik

Von Tobias Müller, Stand 23.03.2017 [Mü17].

$$
\begin{array}{lcl}
pp & ::= & \texttt{def}\ fname\texttt{(}v_1\texttt{,}\ \ldots\texttt{,}\ v_n\texttt{)\{}ss\texttt{;}\ \texttt{return}\ e\texttt{\}}\ \texttt{;}\ pp \\
 & | & ss
\end{array}
$$

$$
\begin{array}{lcl}
ss & ::= & s\ \texttt{;}\ ss \\
 & | & s
\end{array}
$$

$$
\begin{array}{lcl}
s & ::= & v = e \\
 & | & \texttt{if}\ e\ \texttt{then}\ ss_1\ \texttt{else}\ ss_2\ \texttt{fi} \\
 & | & \texttt{for}\ v\ \texttt{in}\ e^{list}\ \texttt{do}\ ss\ \texttt{od} \\
 & | & \texttt{while}\ e\ \texttt{do}\ ss\ \texttt{od} \\
 & | & \texttt{append(}v^{list}\texttt{,}e_{elt}\texttt{)} \\
 & | & \texttt{update(}v^{dict}\texttt{,}e_{idx}\texttt{,}e_{elt}\texttt{)} \\
 & | & \texttt{skip}
\end{array}
$$

$$
v \ ::= \ \langle \textit{variable name} \rangle
$$

$$
\begin{array}{lcl}
e & ::= & v \\
 & | & e^{list}\texttt{[}e\texttt{]} \\
 & | & e^{dict}\texttt{\{}e\texttt{\}} \\
 & | & \texttt{length(}e^{list}\texttt{)} \\
 & | & \texttt{keys(}e^{dict}\texttt{)} \\
 & | & e^{list}\ \texttt{contains}\ e_{elt} \\
 & | & \circledast\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n \\
 & | & fname\texttt{(}e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)} \\
 & | & c
\end{array}
$$

$$
\begin{array}{lcl}
c & ::= & \texttt{None} \\
 & | & \langle integer \rangle \\
 & | & \langle float \rangle \\
 & | & \langle string \rangle \\
 & | & \texttt{[}e_0\texttt{,}\ldots\texttt{,}e_n\texttt{]} \\
 & | & \texttt{\{}c_0\texttt{:}e_0\texttt{,}\ldots\texttt{,}c_n\texttt{:}e_n\texttt{\}} \\
 & | & b
\end{array}
$$

$$
\begin{array}{lcl}
b & ::= & \texttt{True} \\
 & | & \texttt{False}
\end{array}
$$

$$
\begin{array}{lcl}
\circledast & ::= & + \\
 & | & - \\
 & | & * \\
 & | & / \\
 & | & \ldots
\end{array}
$$

scalar operators (fragmentary)

6.2 Server API Call zur Analyse einer SQL Query

```python
@post('/db/analyze_sql_query')
def analyze_sql_query():
    """Analayze sql query via toolchain."""
    name = request.json['name']
    query = request.json['query']
    database = name + CONFIG['dbSuffix']
    directory = os.getcwd() + '/code/' + name + '/'

    # Create new provenance directory to work in
    if not os.path.exists(directory):
        os.makedirs(directory)
    # Write input to input.sql
    with open(directory + '/' + CONFIG['inputQueryName'], 'w+') as input_file:
        input_file.write(query)

    # Run query to check correctness
    try:
        subprocess.check_output(['psql', '-f',
                    CONFIG['inputQueryName'], '-d',
                    CONFIG['sourceDbName'], '-v',
                    'ON_ERROR_STOP=1'],
                    stderr=subprocess.STDOUT,
                    cwd=directory)
    except subprocess.CalledProcessError as e:
        output_text = e.output.decode('ascii')
        output_text = output_text.replace('\n', '<br>') # html newline
        res = {'success': False, 'error': 'query_test', 'output': output_text}
        return json.dumps(res)
```

Listing 6.1: `/db/analyze_sql_query` Server API Call, Teil 1/2

```
try: # Create query.kl
     subprocess.check_output(['tokl', CONFIG['inputQueryName']],
     stderr=subprocess.STDOUT, cwd=directory)
except subprocess.CalledProcessError as e:
     print('Error while executing "tokl" in dir ', directory)
     raise

try: # Create database if not existing
     subprocess.check_output(['createdb', database],
                    stderr=subprocess.STDOUT,
                    cwd=directory)
except subprocess.CalledProcessError as e:
     print('Warning: Database "' + database + '" already exists')

try: # Run provenance analysis
     subprocess.check_output(['prov', database,
                    CONFIG['klQueryName'], 'vis'],
                    stderr=subprocess.STDOUT,
                    cwd=directory)
except subprocess.CalledProcessError as e:
     print('Provenance Analysis failed')
     res = {'success': False, 'error': 'prov_analysis', 'output': e.output}
     return json.dumps(res)
position = add_entry_to_query_info(name, database)
res = {'success': True, 'new_entry': name, 'db_index': position}
return json.dumps(res)
```

Listing 6.2: `/db/analyze_sql_query` Server API Call, , Teil 2/2

6.3 Vollständige KL Query

```
import "!std-dyn.kl";
# ********************************** imprint **********************************;
# ;
# SQL -> KL Compiler;
# ;
# query translated: 2017-08-13 23:53:32.063433 CEST;
# ;
# ******************************** input query ********************************;
# Select zahl;
#     from zahlen;
#     where zahl > 5;;
# ;
# ***************************** annotated query ******************************;
# [R8|main:SELECT;
#     [R1|main:[R2|main:A1].zahl] AS "zahl";
# FROM [R3|main:zahlen] AS A1;
# WHERE;
#     [R4|main:([R5|main:[R6|main:A1].zahl] > [R7|main:5])];
# ];
# ************************ compiler-generated functions ************************;
def sfw9(zahlen)
{ # Sfw [(Region 1 (Col (Region 2 (Var "A1")) "zahl"),"zahl")]
  # [(Region 3 (TableRef "zahlen"),False,"A1")]
  # (Region 4 (Call (PgFun "pg_operator.>")
  # [Region 5 (Col (Region 6 (Var "A1")) "zahl"),
  # Region 7 (LitInt 5)])) [] (LitBool True) [] [] (Null,Null) False;
  # From [(Region 3 (TableRef "zahlen"),False,"A1")]
  # (Region 4 (Call (PgFun "pg_operator.>")
  [Region 5 (Col (Region 6 (Var "A1")) "zahl"),Region 7 (LitInt 5)]));
     Region: 3;
     skip;
     regionResult2 = zahlen;
     RegionEnd: 3;
     data = [];
     dt = {};
     skip;
```

Listing 6.3: Vollständige KL-Query zum `zahl > 5` Beispiel, Teil 1/3

```
for klA1 in regionResult2 do
     skip;
     update(dt, "klA1", klA1);
     Region: 4;
     Region: 5;
     Region: 6;
     regionResult5 = dt{"klA1"};
     RegionEnd: 6;
     regionResult4 = regionResult5{"zahl"};
     RegionEnd: 5;
     Region: 7;
     skip;
     regionResult6 = 5;
     RegionEnd: 7;
     regionResult3 = (regionResult4
                     >
                     regionResult6);
     RegionEnd: 4;
     pred = regionResult3;
     if pred
          then dt2 = dt;
               for dt_key in keys(dt) do
                    dt2 = pt(dt2,
                          dt{dt_key},
                          "ynr")
               od;
               append(data,
                       pt(dt2, pred, "ynr"))
          else skip
     fi
od;
skip;
skip;
skip;
skip;
# Select [(Region 1 (Col (Region 2 (Var "A1")) "zahl"),"zahl")];
ship = [];
```

Listing 6.4: Vollständige KL-Query zum `zahl > 5` Beispiel, Teil 2/3

```
    for dt in data do
        Region: 1;
        Region: 2;
        regionResult8 = dt{"klA1"};
        RegionEnd: 2;
        regionResult7 = regionResult8{"zahl"};
        RegionEnd: 1;
        tuple = {"zahl":regionResult7};
        tuple = pt(tuple, dt, "ynn");
        append(ship, tuple)
    od;
    return ship
};
# ************************** user-defined functions **************************;
# ********************************* table data *********************************;
zahlen = [{"zahl":11}
    ,{"zahl":-28}
    ,{"zahl":19}
    ,{"zahl":-21}
    ,{"zahl":15}
    ,{"zahl":-25}
    ,{"zahl":0}];
# ***************************** query execution ******************************;
def main(zahlen)
{ Region: 8;
    skip;
    regionResult0 = sfw9(zahlen);
    RegionEnd: 8;
    res = regionResult0;
    print(res);
    return res
};
res = main(zahlen);
tupleCount = 0;
for dummy in res do
    tupleCount = (tupleCount + 1)
od;
print(tupleCount)
```

Listing 6.5: Vollständige KL-Query zum `zahl > 5` Beispiel, Teil 3/3

6.4 Private parseRow(row)-Funktion der Klasse Table

```
let parseRow = function(row) {
     let resultRow = [];
     switch(row.type) {
          // Atom value
          case 'null':
          case 'bool':
          case 'int':
          case 'float':
          case 'string':
               // Reset tuple
               tupleHelper.taken = true;
               // Add pids
               that.pids.add(tupleHelper.shapeId);
               that.pids.add(row.id);

               return {tupleVal: tupleHelper.tupleVal,
                       tupleId: tupleHelper.shapeId,
                       colName: row.idx,
                       id: row.id,
                       value: row.value};
}
```

Listing 6.6: Private Funktion `parseRow(row)` im Konstruktor der Klasse `Table`, Teil 1/2

```
            // Nested structure
            case 'list':
            case 'dict':
            case 'row':
                // Overwrite when a new tuple begins
                if (tupleHelper.taken) {
                    tupleHelper.tupleVal = row.idx;
                    tupleHelper.shapeId = row.id;
                    tupleHelper.taken = false;
                }
                row.elements.forEach(function(row) {
                    resultRow.push(parseRow(row));
                });
                break;
            // Does not work in rendering because of mustache.js
            case 'table':
                let tableId = row.varid;
                let tableName = row.varname;
                return Table(tableId, tableName, row);
            default:
                throw new Error("Unknown type" + row.type);
        }
        return resultRow;
    }
```

Listing 6.7: Private Funktion `parseRow(row)` im Konstruktor der Klasse `Table`, Teil 2/2

6.5 mustache.js-Template für Tabellen

```
tableTemplate =
     '<span id="{{tableName}}{{tableId}}" class="provenance-table"> ' +
     '<h4 class="table-heading"> ' +
     '<i class="fa {{tableIcon}}" aria-hidden="true"></i>  ' +
     '{{tableName}}</h4> ' +
     '<button id="hide-rows-button-{{tableId}}" data-value="{{tableId}}"' +
     '        class="btn btn-default hide-unselected-rows-button"> ' +
     '   <i class="fa fa-bars" aria-hidden="true"></i> ' +
     '</button> ' +
     '<button id="plus-button-{{tableId}}" data-value="{{tableId}}" ' +
     '        class="btn btn-default plus-rows-button"> ' +
     '   <i class="fa fa-plus" aria-hidden="true"></i> ' +
     '</button> ' +
     '<button id="minus-button-{{tableId}}" data-value="{{tableId}}" ' +
     '        class="btn btn-default minus-rows-button"> ' +
     '   <i class="fa fa-minus" aria-hidden="true"></i> ' +
     '</button> ' +
```

Listing 6.8: mustache.js Template für eine Tabelle Teil 1/2

```
'<table id="table-{{tableId}}" ' +
' class="table table-striped table-hover" ' +
' cellspacing="0" data-id={{tableId}}> ' +
'   <thead><tr> ' +
'      <th class="tuple-table-head">T</th> ' +
'      {{#head}} ' +
'       <th>{{name}}</th> ' +
'      {{/head}} ' +
'   </thead></tr> ' +
'   <tbody> ' +
'   {{#rows}} ' +
'   <tr> ' +
'      <td class="tuple-table-cell"> ' +
'       <span id="shape-id-{{tupleId}}" class="shape-id" ' +
'        title="Shape id: {{tupleId}}" data-id={{tupleId}}> ' +
'          {{tupleVal}} ' +
'        </span> ' +
'      </td> ' +
'      {{#cells}} ' +
'      <td> ' +
'      {{#items}} ' +
'      <span id="shape-id-{{id}}" class="shape-id" ' +
'       title="Shape id: {{id}}" data-id={{id}}>{{value}}</span> ' +
'      {{/items}} ' +
'       </td> ' +
'      {{/cells}} ' +
'   </tr> ' +
'   {{/rows}} ' +
'   </tbody> ' +
'</table> ' +
'</span> ';
```

Listing 6.9: mustache.js Template für eine Tabelle Teil 2/2

Abbildungsverzeichnis

1.1 Screenshot der Visualisierung anhand eines Beispiels 2
1.2 Position der Visualisierung in der Toolchain 2
1.3 Zelle der Ausgabe-Tabelle ausgewählt 3
1.4 Wechseln der SQL Query 4
1.5 Eingabe einer SQL Query 4
1.6 Regionen als Low-Level-IDs 5
1.7 KL Query-Dialog 5

2.1 Von der Query zur Data Provenance 7
2.2 Tabellen zur SQL Query 2.1. Zelle der Ausgabe-Tabelle ist ausgewählt 8
2.3 Analyse der SQL Query 2.2. Item in der Ausgabe-Tabelle ist ausgewählt 10
2.4 Analyse der SQL Query 2.2. Region im Query-Text ist ausgewählt . . 10

3.1 Model View Controller Grafik 23

4.1 Server-Client-Übersicht 25
4.2 Verzeichnisstruktur auf dem Server 26
4.3 Von der Query in `get_calls()` gelieferte Struktur 31
4.4 Von `nested_call()` gelieferte Struktur 31
4.5 Schnitt der Data Provenances 35
4.6 Unvollständiges Klassendiagramm des Clients 37
4.7 Struktur des Rückgabe-JSON-Objekts von `/db/get_calls` 40
4.8 Struktur des Rückgabe-JSON-Objekts von `/db/get_var` 40
4.9 Struktur des Rückgabe-JSON-Objekts von `/db/get_provenance` 41
4.10 Struktur des Rückgabe-JSON-Objekts von `/db/analyze_sql_query` . . 42
4.11 Interface der Visualisierung 43
4.12 Query Switch-Icon 44
4.13 Query Switch-Popover 44
4.14 Query-Eingabe-Icon 45
4.15 Query-Eingabebereich 46
4.16 Positives Feedback der Query-Analyse 47
4.17 Negatives Feedback der Query-Analyse 47
4.18 Debug-Icon 47
4.19 Regionenbereich 47
4.20 Provenance-Info-Icon 48
4.21 Provenance-Info-Popover 48
4.22 Steuerungsinfo-Icon 48
4.23 Steurungsinfo-Popover 49
4.24 SQL Query-Bereich 49
4.25 SQL Query-Bereich mit markierten Regionen 51

4.26 KL Query-Dialog 52
4.27 Tabellen innerhalb eines Calls 55
4.28 Datenstruktur für das Tabellen-Template 59

Literaturverzeichnis

[Bet14] J. Bettinger. Bachelorthesis: Implementation of a browser-based interface for provenance analysis. Eberhard Karls Universität Tübingen, 2014.

[CS3] Css3 working draft. https://www.w3.org/TR/2001/WD-css3-roadmap-20010523/. Eingesehen am 12.07.2017, 15:30 Uhr.

[ES6] Ecmascript® 2015 language specification. https://www.ecma-international.org/ecma-262/6.0/. Eingesehen am 12.07.2017, 15:30 Uhr.

[HDU] Html/dokumentstruktur und aufbau. https://wiki.selfhtml.org/wiki/HTML/Dokumentstruktur_und_Aufbau. Eingesehen am 07.07.2017, 16:30 Uhr.

[HM5] W3c html5 spezifikation. https://www.w3.org/TR/2011/WD-html5-20110405/. Eingesehen am 11.07.2017, 13:30 Uhr.

[MG15] T. Müller and T. Grust. Provenance for SQL Based on Abstract Interpretation: Value-less, but Worthwhile. In *Proc. VLDB*, Kohala Coast (Hawaii), USA, 2015.

[MOG17] T. Müller, D. O'Grady, and T. Grust. Draft: Fine-grained how-provenance for sql and query compilers. Eberhard Karls Universität Tübingen, 2017.

[Mü17] T. Müller. Draft: Kernel language and provenance derivation, stand 23.03.2017. Eberhard Karls Universität Tübingen, 2017.

[PY3] The python language reference. https://docs.python.org/3.2/reference/. Eingesehen am 12.07.2017, 15:40 Uhr.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift