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

Bachelorarbeit Informatik

Design und Implementierung einer browserbasierten Benutzeroberfläche für Habitat

Lena Weinmann

28.02.2017

Gutachter

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

Betreuer

Benjamin Dietrich
Lehrstuhl für Datenbanksysteme
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Weinmann, Lena:
Design und Implementierung einer browserbasierten Benutzeroberfläche für Habitat
Bachelorarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 01.11.2016 - 28.02.2017

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

Zusammenfassung

Die Arbeit auf großen Datenbanken erfordert korrekt formulierte SQL-Queries. Liefert eine der Datenbankabfragen unerwartete Ergebnisse, ist die Fehlersuche oft sehr aufwändig und unübersichtlich. Zur Vereinfachung dieses Prozesses hat der Lehrstuhl für Datenbanksysteme der Universität Tübingen den Debugger Habitat entwickelt. Dieser liefert Ergebnisse von Teilanfragen der inkorrekten Query, welche bei der Lokalisierung des Fehlers helfen.
Diese Arbeit beschreibt die Entwicklung einer browserbasierten Benutzeroberfläche für die interaktive Nutzung des Debuggers Habitat.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Zur Extrahierung von Daten aus einer Datenbank sind oft umfangreiche und komplizierte Abfragen notwendig. Dabei kommt es schnell zu Fehlern, die schlecht zu finden sind. Einen Ausweg bietet meist nur das Zerkleinern der Abfrage in Teilstücke, deren Ergebnisse einzeln berechnet und überprüft werden. Das ist vor allem bei korrelierten Unterabfragen und rekursiven Queries mit großem Aufwand verbunden. In so einem Fall ist es hilfreich, einen Debugger zu verwenden, um den Fehler zu lokalisieren.
Der Lehrstuhl für Datenbanksysteme der Universität Tübingen hat den Debugger Habitat entwickelt. Er berechnet die Ergebnisse markierter Teilausdrücke einer Query und stellt diese in Zusammenhang. Die Nutzung des Debuggers ist ein interaktiver Prozess, in dem Nutzer, abhängig vom erhaltenen Ergebnis, neue Markierungen anlegen, durch die sie den Fehler lokalisieren.
Der Debugger liefert Daten, die für Menschen nur schwer zu verstehen sind. Diese Arbeit beschreibt die Entwicklung einer Benutzeroberfläche für Habitat zur Darstellung der Ergebnisse des Debuggers. Ziel ist es, die Interaktive Nutzung des Debuggers durch die browserbasierte Oberfläche zu unterstützen und die Ergebnisse korrekt und für Menschen lesbar darzustellen. Außerdem soll die Benutzeroberfläche am Ende der Entwicklung einige Zusatzfunktionen, wie die Verkleinerung des Ergebnisses durch Setzen eines Filters oder die Änderung von Tabellenzellen einer Beobachtung, ermöglichen.

Die Dateien, der im Rahmen dieser Arbeit entwickelten Benutzeroberfläche für Habitat, sind im Git-Repository „HabitatUI“ gespeichert, welches sich hinter der URL `git@dbworld.informatik.uni-tuebingen.de:HabitatUI` verbirgt. Die Version, welche die Grundlage dieser Arbeit bildet, lässt sich über das Git-Tag „bachelorthesis“ finden.

In Kapitel 2 dieser Arbeit wird der Debugger Habitat, welcher das Backend für die Benutzeroberfläche bildet, sowie dessen Schnittstelle beschrieben. Anschließend folgt in Kapitel 3 die Demonstration des Debuggingprozesses anhand einiger Beispiele. Kapitel 4 geht erst auf die zur Implementierung genutzten Technologien ein, bevor die Implementierung der Benutzeroberfläche beschrieben wird. Am Ende folgt das Fazit in Kapitel 5, welches mögliche Probleme sowie mögliche Weiterentwicklungen des Programms aufzeigt.

Kapitel 2

Grundlagen

In diesem Kapitel werden die dem entwickelten Programm zu Grunde liegenden Techniken genauer beschrieben. Darunter fällt zum einen der Debugger Habitat, für den die Benutzeroberfläche entwickelt wurde und zum anderen die Schnittstelle, über die die Oberfläche mit dem Server kommuniziert.

2.1 Habitat

Bei Habitat handelt es sich um einen Debugger für SQL, welcher auf Sprachebene agiert. Durch das Markieren beliebig langer Teilausdrücke kann der Benutzer Teilergebnisse seiner Query auswerten. Dadurch kann er erkennen, ob die Teilausdrücke die erwarteten Ergebnisse liefern bzw. bis zu welcher Stelle das Ergebnis korrekt ist. Zur Auswertung dieser Ausdrücke benötigt Habitat ein Datenbankmanagementsystem (DBMS), welches die SQL Queries berechnet [GR13]. Im Rahmen dieser Arbeit verwendet Habitat PostgreSQL, grundsätzlich sind aber auch andere DBMS möglich [DG15].

> PostgreSQL[1] ist ein objekt-relationales Datenbankmangementsystem (ORDBMS), das frei verfügbar ist. Es läuft auf den gängigen Betriebssystemen. Das DBMS basiert auf dem relationalen Datenmodell und speichert seine Daten deshalb in Relationen als Menge von Tupeln aus Daten [Pos17a]. Der Debugger, der im Rahmen dieser Arbeit verwendet wird, benötigt mindestens PostgreSQL Version 9.5.

Im Folgenden wird die Idee der Implementierung des Debuggers nach [GR13] und [DG15] beschrieben. Ziel des Debuggers ist es, dem Nutzer das Erkennen von Fehlern in der Datenbankabfrage zu erleichtern. Dabei geht es nur um die Erkennung logischer Fehler und nicht um Lauf-

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

zeitanalysen. Habitat liefert die Ergebnisse der Teilausdrücke in Tabellenform. Der Nutzer erhält keine Information über die interne Ausführung der Query im RDBMS, sondern lediglich Ergebnisse für die markierten Ausdrücke. Habitat stellt eine Markierung intern als Funktion f dar. Diese ist abhängig von ihren „freien Row-Variablen“. Eine freie Variable liegt in diesem Kontext dann vor, wenn die Bindung des Variablennamen t_0 an ihren Wert (in der `FROM` Klausel) nicht innerhalb der Markierung stattfindet. Zum besseren Verständnis betrachten wir die Query aus Abb. 2.1. In Zeile 5 wird die Row-Variable $t1$ gebunden. Markierung ① ist abhängig von der freien Variable $t1$, da diese nicht innerhalb der Markierung gebunden wird. In dieser Form ist der markierte Unterausdruck nicht unabhängig vom Rest der Query ausführbar, da er eine Unbekannte enthält. Nach [GR13] liefert Habitat für jede Markierung eine tabellarische Beobachtung zurück, die für jede Bindung der freien Variablen den berechneten Wert enthält. Enthalten mehrere Markierungen dieselben freien Variablen oder ist die Menge der freien Variablen eine Teilmenge derer einer anderen Markierung, fasst der Debugger deren Ergebnisse in einer Tabelle zusammen. Jedes Tupel einer Datenbanktabelle ist über ihren Kontextidentifikator (tid) eindeutig zu identifizieren. Beim Erstellen der Ergebnistabellen nutzt der Debugger zur Berechnung des Kontexts die tids der Tupel, die in das Ergebnistupel einfließen. Als Beispiel betrachten wir einen Join zwischen den zwei Datenbanktabellen a und b, deren erstes Tupel jeweils den Kontext {`tid_a:1`} bzw. {`tid_b:1`} besitzen. Das Tupel, das aus dem Join der beiden ersten Tupel entsteht, erhält den zusammengesetzten Kontext {`tid_a:1, tid_b:1`}. Der Debugger liefert Beobachtungen in einer Tabelle, falls die Identifikatoren dieser Beobachtungen zusammenhängen. Andernfalls liefert er sie in separaten Tabellen.

```
1 SELECT t1.a,
2   ( SELECT t2.c ①
3     FROM tbl2 AS t2
4     WHERE t1.a = t2.c )
5 FROM tbl1 AS t1
```

Abbildung 2.1: Beispielmarkierung mit freier Variable $t1$; ähnlich Fig.2 aus [GR13]

Der Debugger nutzt seit [DG15] die SQL-Funktionen des DBMS zur Berechnung der Ergebnisse. Auf die Verwendung eines externen Codegenerators, wie er in früheren Implementierungsansätzen verwendet wurde, wird verzichtet. Er ermöglicht Markierungen, die sowohl atomare Werte als auch arithmetische und boolsche Ausdrücke enthalten. Außerdem ist es möglich, rekursive Queries und korrelierte wie auch nicht korrelierte Unterabfragen, sowie Queries mit Gruppierungen zu debuggen. Die Auswertung der Teilausdrücke erfolgt während der Berechnung der Query. Habitat ermöglicht das Filtern nach den wichtigsten Tupeln und „What If“-Debugging. Letzteres ermöglicht Änderungen der Zwischenergebnisse, um zu testen, an welcher Stelle der Fehler liegt.[DG15]

Habitat ist in der Programmiersprache Haskell entwickelt. Zur Verwendung des Debuggers ist das Haskell Tool Stack [2] notwendig. Es sorgt dafür, dass das Haskellprogramm auf dem Rechner übersetzt und ausgeführt werden kann, indem es u. a. GHC (Glasgow Haskell Compiler) installiert [Sta].

2.2 Schnittstelle

Die Benutzeroberfläche kommuniziert mit Habitat über eine vorgegebene Schnittstelle, welche vom Habitat-Server bereitgestellt wird. Sie basiert auf HTTP (Hypertext Transfer Protocol). HTTP stellt seit Version HTTP/1.1 die Methoden GET, POST, PUT, DELETE, HEAD, TRACE, OPTIONS und CONNECT bereit [Shi03, S. 26]. Das Protokoll ermöglicht die Kommunikation zwischen dem Server (hier Habitat) und dem Client (hier die Benutzeroberfläche Habitat UI).
Die GET-Methode dient dazu, Informationen zu erhalten. Mit einer GET-Anfrage wird eine URL (Uniform Resource Locator) aufgerufen, um den dort bereitgehaltenen Inhalt anzuzeigen oder zu verarbeiten. Daten können in der URL übertragen werden. Sie werden als Schlüssel=Wert-Paare abgebildet [Shi03, S. 26], z. B. `http://localhost?var1=wert1&var2=wert2`. Im Rumpf der Anfrage können keine Daten mitgesendet werden. Diese Funktionalität wird von der POST-Methode unterstützt. Eine POST-Anfrage sendet Daten in ihrem Rumpf. Dadurch ist die Datengröße und der Datentyp unbeschränkt. Außerdem werden Daten, anders als bei einer GET-Anfrage, nicht sichtbar in der URL versendet. Die PUT-Anfrage wird bereitgestellt, um Daten an den Server zu senden. Sie dient meist nur der Speicherung auf dem Server und nicht zum Empfang von Inhalten. Das Gegenstück zur PUT-Methode ist die DELETE-Methode. Durch eine DELETE-Anfrage werden Daten auf dem Server gelöscht. Die restlichen Methoden werden in der Schnittstelle nicht verwendet, deshalb werden sie hier nicht weiter erläutert.

[2] https://docs.haskellstack.org/

2.2.1 Funktionen

Im Folgenden werden die Funktionen, die die Schnittstelle bereitstellt, aufgelistet und beschrieben. Dazu gehören das Anlegen einer Debugging Sitzung, das Bereitstellen, Ändern und Abrufen von Queries, Markierungen anlegen, ändern, abrufen und deaktivieren sowie das Abrufen von gefilterten und ungefilterten Beobachtungen. [Die] legt die URLs und HTTP-Methoden sowie Parameter der Anfragen und die Antworten des Servers fest. Außerdem sind dort die Statuscodes und -meldungen definiert. Anhand dieser und eigener Erkenntnisse, die bei der Entwicklung der Benutzeroberfläche entstanden sind, werden die Funktionen der Schnittstelle erklärt.

Anlegen einer neuen Debugging Sitzung

Habitat verlangt eine Verbindung zur Datenbank. Dazu muss sich ein Benutzer im Server mit seiner loaklen PostgreSQL Datenbank verbinden. Der Server benötigt den lokalen Benutzernamen (**user**) und den Namen der Datenbank (**dbname**), in der der Nutzer arbeiten will. Optional können noch einige weitere Parameter übergeben werden: Die Adresse des Datenbankservers (**host**), die standardmäßig als `localhost` gesetzt ist; der Port, auf dem der Datenbankserver lauscht (**port**), der standardmäßig auf `5432` gesetzt ist und das Passwort (**password**), das angegeben werden muss, falls der Benutzer ein Passwort zur Anmeldung auf der Datenbank festgelegt hat. In den folgenden URLs sind die dick hervorgehobenen Wörter als Variablen zu verstehen.
Die Schnittstelle erwartet eine POST-Anfrage über die URL `http://`**`host`**`:`**`port`**`/connect?user=`**`user`**`&dbname=`**`dbname`**`&password=`**`password`**.
Der Server versucht, eine Verbindung zur angegebenen Datenbank herzustellen. Bei einer erfolgreichen Verbindung liefert er eine Antwort mit dem Status `201 Created` zurück. Der Rumpf enthält die Kennung der erstellten Sitzung (**sid**), welche als Zahl dargestellt ist. Die Kennung wird während der kompletten Arbeit mit Habitat benötigt, um die aktuelle Sitzung zu identifizieren. In manchen Fällen schlägt die Verbindung zur Datenbank fehl. Das ist beispielsweise der Fall, wenn der Benutzer einen falschen Benutzernamen, den Namen einer Datenbank, die nicht existiert oder ein falsches Passwort angibt. Kommt es zu so einem Fehler, enthält die Antwort des Servers eine Fehlermeldung. Der Status einer solchen Antwort ist `403 Forbidden`. [Die]

Bereitstellen bzw. Ändern einer Query

Bevor eine Query (Datenbankabfrage) getestet werden kann, muss sie dem Server bereitgestellt werden. Das geschieht sowohl am Anfang, beim Eingeben der Query, als auch bei jeder Änderung. Beim Ändern der Query deaktiviert Habitat alle bereits gesetzten Markierungen, um solche, die nach der Änderung ungültig sind, zu vermeiden. Markierungen, die auch nach der Änderung erhalten bleiben sollen, müssen im Nachhinein reaktiviert werden.
Durch eine PUT-Anfrage übermittelt der Client dem Server die aktuelle Query. Dies geschieht über die URL `http://host:port/sid/query`. Der Anfragerumpf enthält die Query, welche als String oder als JSON-Objekt vorliegt. Kann der Server die Anfrage verarbeiten, liefert er eine leere Antwort mit dem Status `204 No Content`. Bei der Verarbeitung der Query können Fehler auftreten, wenn die Query nicht der korrekten SQL-Syntax entspricht. In diesem Fall antwortet der Server mit dem Status `900 Bad Input`. Im Rumpf der Antwort teilt der Server dem Client mit, an welcher Stelle in der Query ein Fehler vorliegt. Ein weiterer Fehler entsteht beim Angeben einer nicht spezifizierten Kennung. Das kann passieren, wenn der Benutzer an einer Query arbeitet und der Server zwischenzeitlich neu gestartet wird. Dann greift der Client auf eine Sitzung zu, deren Kennung dem Server unbekannt ist. Um das dem Client mitzuteilen, antwortet der Server mit dem Status `404 Not Found`, wobei der Client erfährt, dass die angeforderte Sitzung nicht existiert. [Die]

Query abrufen

Der Client kann eine GET-Anfrage stellen, um die Query abzurufen, die auf dem Server unter der URL `http://host:port/sid/query` vorgehalten wird. Die Query, die der Server im Rumpf der Anfrage zurückgibt, kann der Client prüfen und gegebenenfalls, wie oben beschrieben, ändern. Bei erfolgreicher Anfrage ist der Status der Antwort `200 OK`. Beim Auftreten eines Fehlers enthält der Rumpf statt der Query eine Fehlermeldung, die Auskunft über die Art des Fehlers gibt. Mögliche Ursache kann eine fehlerhafte Kennung der Sitzung sein. Auch für den Fall, dass bisher keine Query bereitgestellt wurde, liefert der Server einen Fehler mit dem Status `404 Not Found`. [Die]

Markierung anlegen

Habitat arbeitet, wie in Kapitel 2.1 beschrieben, mit Markierungen. Um eine Markierung anzulegen, erwartet der Server eine POST-Anfrage, welche die Start- (**from**) und Endposition (**to**) der Markierung als Zahlenwerte enthalten muss. Die erste mögliche Position ist 0, die letzte beträgt die Länge der Query. Der Client sendet die POST-Anfrage an die URL `http://host:port/sid/query/markings?from=from&to=to`. Die Ant-

wort des Servers hat bei erfolgreichem Anlegen der Markierung den Status `200 OK`. Die Markierung erhält der Client als JSON-Objekt. Darin sind die Daten der Markierung gespeichert:

- `mid` - Die eindeutige Identifikation der Markierung.
- `active` - Ein bool'scher Wert, welcher anzeigt, ob die Markierung aktiv ist (`true`) oder nicht (`false`).
- `from` - Die Startposition der Markierung als Zahlenwert. Sie kann sich von der Startposition unterscheiden, die der Client in der POST-Anfrage übermittelt hat. Der Server soll aus fehlerhaften Markierungen korrekte erstellen. Daher ändert sich die Position möglicherweise. Bisher ist diese Funktion jedoch noch nicht implementiert.
- `to` - Die Endposition der Markierung als Zahlenwert. Diese kann sich aus dem gleichen Grund wie die Startposition von der ursprünglichen Endposition unterscheiden.
- `type` - Der PostgreSQL Datentyp des Ergebnisses des markierten Ausdrucks. Dieser kann ein atomarer Typ, wie z. B. Integer oder Boolean, sein. Dann enthält `type` den Namen des Datentyps als String. Ist das Ergebnis ein tabellarischer Wert, wird der Typ als Array angegeben. Dabei enthält er für jede Tabellenspalte einen Eintrag. Die Einträge sind Objekte von folgendem Typ: {`Spaltenname: Datentyp der Spalte`}.
- `context` - Der Kontext der Markierung. Beim Auswerten des Teilausdrucks bearbeitet der Server möglicherweise mehrere Tabellen. Jedes Tupel dieser Tabellen hat eine eindeutige Identifikation. Für jede Tabelle gibt es einen Identifikator, welcher erkennen lässt, um welche Tabelle es sich handelt (z. B. jedes Tupel der Tabelle R hat einen eindeutigen Tupelidentifikator `tid_r`). Werden für die Markierung Tupel aus Tabelle R und Tabelle S benötigt, ist ihr Kontext `[tid_r, tid_s]`. Für andere Fälle, wie Gruppierungen, gibt es spezielle Kontextidentifikatoren, die vom Habitatserver erstellt werden. Mithilfe dieses Kontexts kann der Client erkennen, in welchem Zusammenhang das Ergebnis steht. Das ist vor allem dann wichtig, wenn mehrere Markierungen in einem Gesamtergebnis dargestellt werden, um geschachtelte Ergebnisse von anderen zu unterscheiden.

Diese Daten ermöglichen es, clientseitig mit der Markierung zu arbeiten. Auch beim Anlegen einer Markierung sind Fehlerfälle möglich. Bei der Angabe einer ungültigen Sitzungskennung liefert der Server wie oben einen Fehler mit dem Status `404 Not Found`. Denselben Fehler erhält der Client, wenn er dem Server zuvor keine Query bereitgestellt hat. Die Fehlermeldung hat den Status `900 Bad Input`, falls ein Teil der Query markiert wird, der kein gültiger Unterausdruck in PostgreSQL-Syntax ist. [Die]

Markierung ändern bzw. aktivieren

Nachdem eine Query geändert wurde, muss der Client dem Server mitteilen, welche Markierungen aktiv bleiben sollen. Zusätzlich muss er die angepassten Start- und Endpositionen übergeben. Das macht er über eine PUT-Anfrage an die URL `http://`**`host`**`:`**`port`**`/`**`sid`**`/query/markings/`**`mid`**`?from=`**`from`**`&to=`**`to`**. Nachdem der Server die Markierung geändert hat, antwortet er mit dem Status `200 OK`. Die Antwort enthält ein JSON-Objekt mit den Daten der geänderten Markierung. Das JSON-Objekt hier und das im vorherigen Abschnitt haben dieselbe Struktur. Beim Ändern oder Reaktivieren einer Markierung können dieselben Fehler auftreten, wie beim Anlegen einer Markierung. [Die]

Markierung abrufen

Um dem Client Informationen über eine angelegte Markierung zu liefern, stellt der Habitatserver über die Schnittstelle eine Funktion bereit, durch die der Client Informationen zu einer Markierung abrufen kann. Diese erhält er durch eine GET-Anfrage über die URL `http://`**`host`**`:`**`port`**`/`**`sid`**`/query/markings/`**`mid`**. In Folge einer korrekten Anfrage liefert der Server die Informationen zur Markierung als JSON-Objekt, wie oben mit dem Status `200 OK`, zurück. Neben den Fehlern von oben, wie der fehlerhaften Angabe der **sid** oder das Fehlen der Query, führt auch die Angabe einer inkorrekten **mid** zu einem Fehler mit dem Status `404 Not Found`. [Die]

Markierung deaktivieren

Der Client teilt dem Server mit, falls eine Markierung für den Benutzer uninteressant ist. Er stellt eine DELETE-Anfrage über die URL `http://`**`host`**`:`**`port`**`/`**`sid`**`/query/markings/`**`mid`**. Der Server setzt `active` der Markierung auf `false` und liefert eine leere Antwort mit dem Status `204 No Content` zurück an den Client. Beim Deaktivieren von Markierungen können dieselben Fehler auftreten, wie auch beim Abrufen der Markierungen. [Die]

Markierungen auflisten

Eine weitere Funktion, die der Server bereitstellt, ist das Auflisten aller bisher gesetzten Markierungen. Als Antwort einer GET-Anfrage über die URL `http://host:port/sid/query/markings` erhält der Client ein JSON-Objekt. Dieses wird im Rumpf der Serverantwort übermittelt, welchen den Status `200 OK` hat. Das JSON-Objekt enthält eine Liste aller Markierungen. Die Markierungen werden wie im Abschnitt „Markierung anlegen“ dargestellt. Mögliche Fehler sind wiederum die Angabe einer inkorrekten **sid**. Diesen übermittelt der Server mit dem Status `404 Not Found`. [Die]

Beobachtungen abrufen

Den Benutzer interessiert vor allem das Ergebnis der markierten Ausdrücke. Um diese abzurufen, sendet der Client eine GET-Anfrage über die URL `http://host:port/sid/observations`. Bei erfolgreicher Bearbeitung der Anfrage antwortet der Server mit dem Status `200 OK`. Die Antwort enthält einen Array von Objekten, welche die Beobachtungen der markierten Ausdrücke darstellen. Das JSON-Objekt einer Beobachtung hat die folgende Struktur:

```
[
  {
    "context" :     {tid: string, ...},
    "observations" :  {mid: obs-value, ...},
    "nested" :      [observation]
  }
]
```

Abbildung 2.2: aus [Die]; Struktur einer Beobachtung

Eine Beobachtung besteht aus einer Liste von Zeilen. Jede Zeile enthält die Schlüsselwerte `"context", "observations", "nested"`.

- Der `context` gibt an, in welchem Kontext die Beobachtung(en) dieser Zeile stehen. Er baut sich aus den Identifikatoren der Tupel zusammen, die für die Beobachtungen dieser Zeile benötigt wurden. Ein möglicher Kontext sieht wie folgt aus: `{"tid_r" : "(0,1)" }`. Er wird zur Zuordnung der Tupel in die entsprechende Tabellenzeile bei der Ausgabe der Beobachtung benötigt. Zu jeder `tid`, die im Objekt enthalten ist, gibt es eine aktive Markierung, die in ihrem Kontext dieselbe `tid` enthält.

- Hinter dem Schlüsselwert `observations` verbergen sich die tatsächlichen Beobachtungen. Je nachdem, welchen Typ das Ergebnis der entsprechenden Markierung hat, kann eine Beobachtung aus einem atomaren Wert oder aus einem Tupel bestehen. Das nachfolgende Beispiel zeigt eine atomare (a) und eine tabellarische Beobachtung (b).

  (a) `{"m1" : [1000]}`

  (b) `{"m2" : ["population" : [1000]]}`

  Eine leere Beobachtung wird im Objekt mit dem Wert `null` dargestellt. Die `mids` entsprechen einem Identifikator für eine aktive Markierung.

- In `nested` kann das Objekt weitere Beobachtungen enthalten. Es gibt keine obere Grenze für die Tiefe der Verschachtelung durch `nested`-Objekte. Hauptsächlich verwendet Habitat diesen Teil des Objekts zur Darstellung von Gruppierungen oder Ergebnissen von korrelierten Subqueries. Beobachtungen im `nested`-Teil des Objekts stehen meist im Bezug zu den im `observations`-Teil gespeicherten Beobachtungen.

Der Server benötigt eine Query, um die Anfrage auszuwerten. Ist diese nicht vorhanden, antwortet er dem Client mit dem Status `404 Not Found`. Außerdem können Fehler bei der Auswertung entstehen. Diese übermittelt er mit dem Status `500 Internal Server Error`. [Die]

Gefilterte Beobachtungen anzeigen

Realistische Datenbanktabellen sind meist sehr groß. Sie enthalten sowohl viele Tupel als auch viele Spalten. Um die Wartezeit für den Benutzer zu verkürzen, bietet Habitat die Möglichkeit, nur bestimmte Tupel zu übertragen. Der Benutzer kann die für ihn interessanten Tupel auswählen. Diese Information übergibt der Client beim Abruf der gefilterten Beobachtung dem Server, welcher nur die interessanten Tupel zurückliefert. Außerdem ist der Benutzer durch diese Funktion in der Lage, einen Wert in der Beobachtung zu verändern. Dadurch kann er herausfinden, ob die Query das erwartete Ergebnis liefert, wenn die Beobachtung an einer Stelle abweicht. Auch diese Information sendet der Client an den Server. Dieser verwendet zur Berechnung der Ergebnisse den geänderten anstelle des eigentlichen Werts. Der Client sendet eine POST-Anfrage über die URL `http://host:port/sid/observations/based-on`. Im Anfragerumpf übergibt er die Einschränkungen. Diese bestehen aus dem Filter für bestimmte Tupel und der Ersetzung eines oder mehrerer Werte. Das Objekt der Filter wird im Folgenden genauer beschrieben. Abb. 2.3 zeigt die Struktur des Filters.

```
{
  "filter" :  {"tid" : [string], ...},
  "what-if" : [
          {
            "mid" :     mid,
            "column" :    string,
            "context" :   [string],
            "substitute": string
          },
          ...
        ]
}
```

Abbildung 2.3: Struktur eines Filterobjekts nach [Die]

Im `"filter"` gibt der Client an, welche Tupel er in der Antwort erhalten will. Der Client bildet diesen aus den Tupelidentifikatoren der interessanten Tupel. Der Filter ist eine Liste der Identifikatoren. Ist z. B. das Tupel mit dem Kontext {`"tid_r": "(0,1)", "tid_s": "(0,2)"`} interessant für den Benutzer, so sieht der Filter wie folgt aus: {`"tid_r": ["(0,1)"], "tid_s": ["(0,2)"]`}. Ist zusätzlich das Tupel mit dem Kontext {`"tid_r": "(0,3)", "tid_s": "(0,4)"`} interessant, so ensteht der folgende Filter: {`"tid_r": ["(0,1)", "(0,3)"], "tid_s": ["(0,2)", "(0,4)"]`}. Der Server lässt nur Tupel in die Beobachtung einfließen, deren Kontext dem Filter entspricht. Der Client identifiziert im `"what-if"` Teil die Markierung, deren Beobachtung der Benutzer an einer Stelle ändern will, durch den Identifikator der Markierung (`mid`). Falls das Ergebnis dieser Markierung ein tabellarischer Wert ist, muss zusätzlich eine Identifizierung der Spalte stattfinden. Der Client gibt dafür den entsprechenden Spaltennamen als Wert für den Schlüssel `"column"` an. Ist das Ergebnis der Markierung atomar, ist der Wert `null`. Um die korrekte Zeile zu identifizieren, ist ihr Kontext notwendig. Dieser wird in der Spalte `"context"` angegeben. Schließlich gibt man den neuen Wert in der Spalte `"substitute"` an. Es können gleichzeitig mehrere dieser Ersetzungen durchgeführt werden.
Kann der Server die Anfrage erfolgreich bearbeiten, liefert er die Beobachtungen mit dem Status `200 OK` zurück. Die Beobachtungen übermittelt der Server als JSON-Objekt, wie im Abschnitt „Beobachtungen Abrufen“, als Antwort. Einen Fehler mit dem Status `400 Bad Request` liefert der Server, falls der Kontext im `"what-if"` Teil des Filters nicht mit dem Kontext der angegebenen Markierung übereinstimmt oder auch wenn keine Spalte für ein tabellarisches Ergebnis angegeben wird. Ist die `mid` dem Server unbekannt, antwortet er mit dem Status `404 Not Found`. Weitere Fehler, wie das Fehlen einer Query bzw. die Angabe einer falschen Sitzungskennung, werden wie oben behandelt. [Die]

Kapitel 3

Funktionsweise

In diesem Kapitel wird die Funktion der Benutzeroberfläche anhand von Beispielen erklärt. Dazu werden die Tabellen 3.1 bis 3.4 verwendet. Die Tabellen 3.1 und 3.2 stammen aus [DG15]. Das Szenario beschreibt eine Miniwelt, in der die vier Städte Arvin, Biggs, Chico und Dixon liegen. Die Tabelle **cities** (3.2) gibt Informationen über die Bevölkerungszahl jeder Stadt und über die Anwesenheit einer Tankstelle im Ort. Die Besonderheit ist, dass sich das Straßennetzwerk als azyklischer Graph zeichnen lässt. Die Spalte `dist` in **roads** (3.1) gibt an, wie viele Liter Kraftstoff man benötigt, um die Straße von `here` nach `there` zu befahren. Zur Veranschaulichung des Straßennetzwerks dient Abb. 3.1.

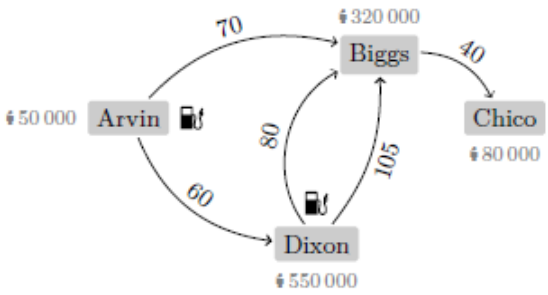


Abbildung 3.1: aus [DG15]; Straßennetzwerk der Miniwelt mit den Werten der Tabellen **cities** und **roads**

here	dist	there
Arvin	70	Biggs
Arvin	60	Dixon
Biggs	40	Chico
Dixon	80	Biggs
Dixon	105	Biggs

Tabelle 3.1: aus [DG15]; roads

city	population	fuel
Arvin	50.000	1
Biggs	320.000	0
Chico	80.000	0
Dixon	550.000	1

Tabelle 3.2: aus [DG15]; cities

job	city
Carpenter	Dixon
Chemist	Biggs
Scientist	Arvin
Hairdresser	Chico
Baker	Arvin

Tabelle 3.3: jobs

firstName	lastName	residence	licence	job	car
John	Doe	Arvin	0	Carpenter	limousine
Bob	Brown	Biggs	1	Chemist	cabriolet
Nadja	Smith	Biggs	1	`null`	`null`
David	Miller	Chico	1	Scientist	suv
Jennifer	Williams	Dixon	0	Hairdresser	`null`
Nina	Johnson	Dixon	1	Scientist	cabriolet

Tabelle 3.4: people

In der Tabelle **people** (3.4) sind Informationen über exemplarische Bewohner dieser Städte enthalten. Die Tabelle **jobs** (3.3) hält Informationen über Berufe bereit, die in der Miniwelt ausgeführt werden können.

3.1 Gruppierung

Das `GROUP BY` Konstrukt in SQL erlaubt es, Tupel, die in einem oder mehreren Argumenten übereinstimmen, zusammenzufassen [Pos17f]. Die Aggregatfunktionen, wie z. B. `sum()`, `count()`, `max()`, `min()` oder `avg()`, können Werte für die Gruppe berechnen [Pos17d].

```
1 SELECT r.there AS city, SUM(c.population) AS belt
2 FROM cities AS c, roads AS r
3 WHERE c.city = r.here
4 GROUP BY r.there;
```

(② bei `c.population`, ③ bei `r`, ① bei `r.there`)

Abbildung 3.2: Query (aus [DG15]) zur Berechnung der Einzugsbereichsgrößen (fehlerhaft)

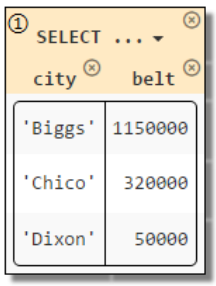


Abbildung 3.3: Ergebnis der gruppierten Query (fehlerhaft)

Die Beispielabfrage aus [DG15] in Abb. 3.2 berechnet für jede Stadt die Summe der Bevölkerungszahlen der Nachbarstädte, deren Straßen in diese Stadt führen. Das Ergebnis gibt eine ungefähre Auskunft über den Pendlerverkehr in eine Stadt. Die Abfrage führt einen Join der Tabellen **roads** und **cities** durch. Städte mit eingehenden Straßen lassen sich in der Spalte `there` in der Tabelle **roads** finden. Jede dieser Städte findet ihre direkten Nachbarstädte aus **cities** mit Hilfe des Prädikats in Zeile 3. Für jede Stadt fasst die Query die Nachbarstädte in einer Gruppe zusammen und berechnet für jede dieser Gruppen die Summe der Nachbarbevölkerungen (`belt`). Ein Blick auf das Ergebnis dieser Query (siehe Abb. 3.3) zeigt, dass die Summe der Bewohner in der Nachbarschaft von Biggs 1.150.000 beträgt, während die Gesamtbevölkerung der Miniwelt nur bei 1.000.000 liegt. Demnach ist die Query fehlerhaft. [DG15, S. 4]

Habitat vereinfacht die Fehlersuche. Der nachfolgende Debuggingprozess folgt den Strukturen von [DG15], jedoch nutzen wir hier zusätzlich die Filterfunktion und „What If“-Debugging. Zuerst lassen wir uns durch Markierung der gesamten Query das Endergebnis anzeigen. Wir wissen, dass der Fehler in der ersten Zeile des Ergebnisses liegt. Deshalb filtern wir es nach dieser Zeile.

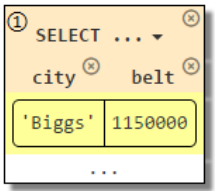


Abbildung 3.4: Abb. 3.3 gefiltert nach Biggs

Abb. 3.4 zeigt das Resultat. Es werden nur noch die Tupel angezeigt, die direkt mit der Biggs-Gruppe zu tun haben. Interessant ist, welche Bevölkerungszahlen zu der großen Zahl des Einzugsbereichs von Biggs führen. Daher lassen wir uns das Ergebnis des Unterausdrucks `c.population` anzeigen. Im Normalfall listet es für jede der drei Gruppen die Bevölkerungen auf, die in das Ergebnis der jeweiligen Gruppe einfließen. Da wir das Ergebnis aber einen Schritt zuvor gefiltert haben, zeigt uns das Programm nur die Beobachtungen der Biggs-Gruppe an. Es fällt auf, dass die Zahl 550.000 doppelt in der Biggs-Gruppe auftaucht, obwohl es nur eine Stadt mit dieser Bevölkerungszahl gibt. Um herauszufinden, was der Grund dafür ist, benötigen wir die Auskunft über die Tupel in **roads**, welche in das Ergebnis einfließen. Das Ergebnis (Beobachtung ②) zeigt, dass es zwei Straßen gibt, die von Dixon nach Biggs führen (siehe Abb. 3.5). Daher zählt die Einwohnerzahl von Dixon doppelt in die Summe des Einzugsbereichs von Biggs.

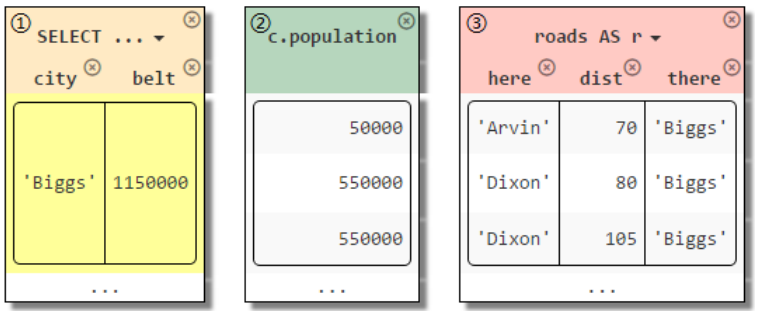
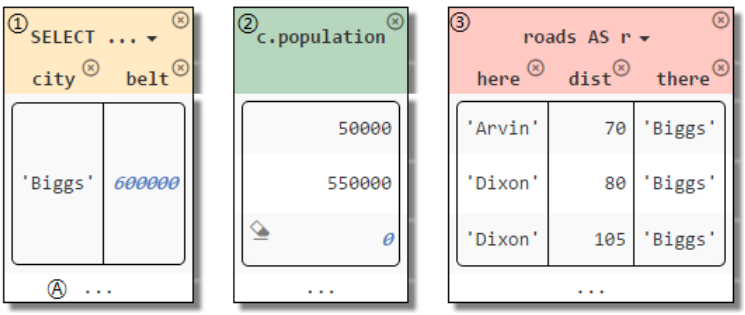
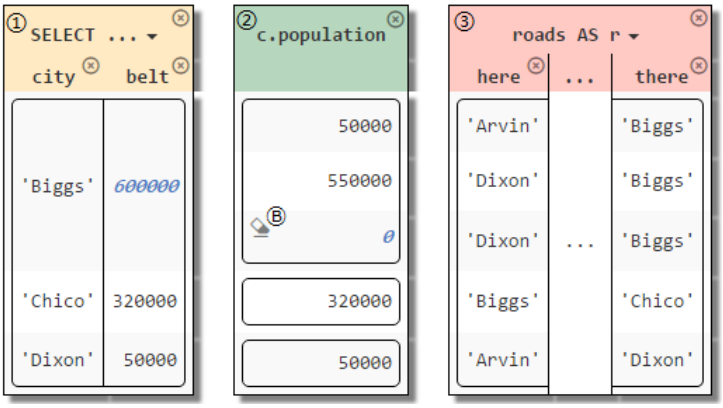


Abbildung 3.5: Anzeige der gefilterten Beobachtungen der Query aus Abb. 3.2

Da wir nun wissen, an welcher Stelle der Fehler liegt, können wir dank des „What If“-Debuggings testen, ob das Ergebnis der Erwartung entspricht, wenn die Einwohnerzahl von Dixon nicht doppelt in das Ergebnis einfließt. Dazu ändern wir eines der beiden Auftreten von 550.000 in Beobachtung ② zu 0. Abb. 3.6a zeigt das hierdurch veränderte Ergebnis. Um sicherzustellen, dass wir keine Änderungen in anderen Gruppen übersehen, entfernen wir den Filter und lassen uns wieder die gesamte Beobachtung anzeigen (siehe Abb. 3.6b). Rechnet man von Hand nach, erkennt man, dass dies nun das korrekte Ergebnis ist.



(a) gefiltert



(b) ungefiltert

Abbildung 3.6: Ergebnisse der gruppierten Query mit Änderung durch „What If“-Debugging

Der letzte Schritt besteht nun darin, die Query zu korrigieren. Wir wissen, dass Orte, die durch mehr als eine Straße miteinander verbunden sind, mehrfach in das Ergebnis eingehen. Daher reicht es nicht aus, wie bisher angenommen, einen Join auf den beiden Tabellen **roads** und **cities** durchzuführen. Stattdessen muss die Query Duplikate von Straßen, bei denen sowohl **here** als auch **there** übereinstimmen, eliminieren [DG15]. Die Abfrage in Abb. 3.7 erfüllt dies und liefert das korrigierte Ergebnis aus Abb. 3.6b.

```
1 SELECT r.there AS city, SUM(c.population) AS belt
2 FROM cities AS c, (SELECT DISTINCT here, there FROM roads) AS r
3 WHERE c.city = r.here
4 GROUP BY r.there;
```

Abbildung 3.7: aus [DG15]; Korrekte Berechnung der Einzugsbereichsgrößen

Im Normalfall debuggen wir die Query durch Teilabfragen. Wir stellen eine Anfrage an das Datenbanksystem, die uns alle Städte liefert, die in die kritische Gruppe hineinfallen. Die Query in Abb. 3.8 liefert dieses Ergebnis und gibt zusätzlich die Einwohnerzahlen dieser Städte aus. Auch hier stellen wir

fest, dass das Tupel (Dixon, 550.000) doppelt vorkommt. Zur Überprüfung der Herkunft listen wir die Tabelle **roads** auf, welche zwei Straßen von Dixon nach Biggs enthält. Auch hier kennen wir nun das Problem und können die korrekte Query aus Abb. 3.7 ermitteln. Bei Tabellen der Größenordnung von **roads** und **cities** ist diese Art des Debuggings nicht problematisch. Schon bei 100 Einträgen in **roads** wird es aber unübersichtlich, da wir im Gegensatz zum Debugging mit Habitat die Tupel der Tabelle nicht im Bezug zum Endergebnis sehen.

```
1 SELECT c.city, c.population
2 FROM cities AS c, roads AS r
3 WHERE r.there = 'Biggs'
4 AND c.city = r.here;
```

Abbildung 3.8: Query zur Fehlersuche ohne Habitat (zu Abb. 3.2)

3.2 Korrelierte Unterabfragen

Eine Query mit korrelierter Unterabfrage enthält eine Unterabfrage, die Informationen der äußeren Abfrage verwendet. D. h., dass das Tupel, das die äußere Query in einem Durchlauf bindet, in diesem Durchlauf auch für die Unterabfrage verwendet wird. Die korrelierte Abfrage kann nicht separat ausgewertet werden, da sie für jeden Durchlauf der Query, die sie umgibt, Informationen des dort verwendeten Tupels benötigt. Die Beispielquery in Abb. 3.9 enthält eine korrelierte Unterabfrage (Zeile 3-6). Dort wird die Spalte `c.city` referenziert, welche aus dem `FROM` Teil (Zeile 2) der umgebenden Query stammt. Die innere Abfrage muss also für jeden Durchlauf der Äußeren, d. h. für jedes Tupel aus **cities**, ausgeführt werden.

```
1 SELECT c.city ①
2 FROM cities AS c
3 WHERE NOT EXISTS (  SELECT 1
4           FROM people AS p②
5           WHERE c.city = p.residence③
6           AND (NOT p.licence::Bool OR p.car = NULL④));
```

Abbildung 3.9: Ermittlung aller Städte, deren Bewohner sowohl Führerschein als auch ein Auto besitzen (fehlerhaft)

Wir suchen alle Städte, deren Bewohner alle sowohl einen Führerschein als auch ein Auto besitzen. Wir können die Aufgabenstellung umformen, indem wir alle Städte suchen, in denen es Niemanden gibt, der keinen Führerschein oder kein Auto besitzt. Zur Beantwortung dieser Fragestellung verwen-

den wir die Query in Abb. 3.9. Zuerst nehmen wir uns alle Tupel aus **cities** vor (Zeile 2). Für jedes dieser Tupel testen wir alle Personen aus Tabelle **people**, deren Wohnort mit der Stadt des Tupels aus **cities** übereinstimmt (Zeile 4 und 5). Nur wenn es unter diesen Personen jemanden gibt, der kein Auto oder keinen Führerschein besitzt, liefert die Unterabfrage ein Ergebnis (Zeile 6). Zum Schluss sammelt die äußere Query alle Städte auf, für die die Unterabfrage kein Ergebnis geliefert hat (`NOT EXISTS` in Zeile 3). Laut dem Ergebnis dieser Query (siehe Abb. 3.10) sind Biggs und Chico die gesuchten Städte. Betrachten wir die Tabelle 3.4 **people** genauer, sehen wir aber, dass in Biggs Nadja Smith lebt, die kein Auto besitzt. Also sollte Biggs nicht im Ergebnis der Query enthalten sein.

Abbildung 3.10: Ergebnis der korrelierten Query (fehlerhaft)

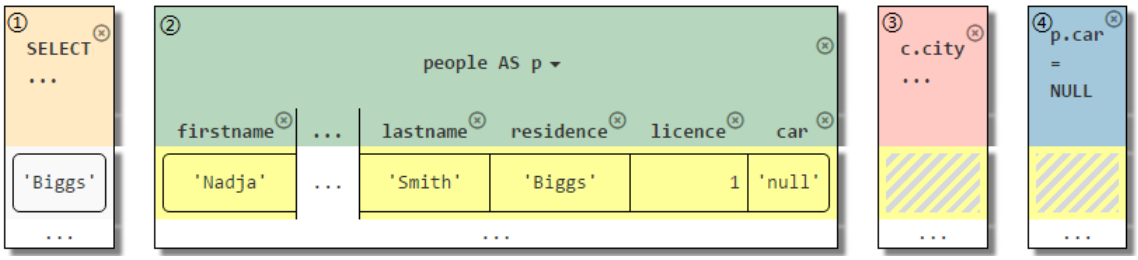


Abbildung 3.11: Anzeige der gefilterten Beobachtungen zu Abb. 3.9

Wir nutzen wieder HabitatUI zur Fehlersuche. Die Markierung der gesamten Query liefert uns das Ergebnis aus Abb. 3.10. Nun lassen wir uns die Tabelle **people** anzeigen und sehen, dass die Oberfläche diese für jede Stadt separat auflistet. Uns interessiert nur die Person Nadja Smith in der Ergebniszeile von Biggs. Also wenden wir einen Filter in dieser Zeile an (Abb. 3.11, Markierung ②). Als nächstes überprüfen wir das Prädikat (Markierung ③). Entgegen der Erwartung, die aus der Betrachtung der Tabelle **people** hervorgeht, ergibt das Prädikat an dieser Stelle `null`. Wir zerlegen das Prädikat zur genaueren Untersuchung. Der Fehler muss im letzten Teil des Prädikats liegen, da wir dort testen, ob ein Auto vorhanden ist (Markierung ④). Dieser ergibt `null` und ist somit für das falsche Ergebnis verantwortlich. Dieser Wert beeinflusst die gesamte logische Verknüpfung und sorgt für das unerwartete Ergebnis. Wir testen `p.car = NULL`. Werte können in SQL nicht mit `null` durch mathematische Operatoren verglichen werden. Denn es gilt: `null` $\neq$ `null` [Pos17c]. Wir müssen also den Test auf `null` verändern. Laut [Pos17c] funktioniert dieser in PostgreSQL durch `IS NULL`. Nach dieser Verbesserung der Query (siehe Abb. 3.12a) erhalten wir das erwartete Ergebnis, welches in Abb. 3.12b zu sehen ist.

```
1 SELECT c.city
2 FROM cities AS c
3 WHERE NOT EXISTS (
4     SELECT 1
5     FROM people AS p
6     WHERE c.city = p.residence
7     AND (NOT p.licence::Bool OR p.car IS NULL));
```

(a) korrigierte Query

(b) Ergebnis der korrigierten Query

Abbildung 3.12: Korrekte Ermittlung aller Städte, deren Bewohner sowohl Führerschein als auch ein Auto besitzen

3.3 Rekursion

Rekursive SQL Queries erweitern die Möglichkeiten der Datenbankabfragen. Eine rekursive Query terminiert erst, wenn sie keine Tupel mehr generieren kann [Pos17b]. So kann sie beispielsweise in einem Graphen bis zum letzten Knoten absteigen, um die gefragten Informationen zu berechnen. Anhand des folgenden Beispiels wird die Verarbeitung von rekursiven Queries durch Habitat und deren Darstellung auf der Oberfläche gezeigt.

```
1  WITH RECURSIVE hops(city, gauge) AS (
2      VALUES ('Arvin', 0)
3    UNION ALL
4      SELECT  r.there AS city,
5          h.gauge + c.fuel * 100 - r.dist AS gauge
6      FROM  cities AS c, roads AS r, hops AS h
7      WHERE   h.city = c.city
8      AND   h.city = r.here
9      AND   h.gauge + c.fuel * 100 >= r.dist
10 )
11 SELECT *
12 FROM hops;
```

Abbildung 3.13: Rekursive Query (aus [DG15]) zur Berechnung aller Städte, die von Arvin erreichbar sind (fehlerhaft)

Mit Hilfe der rekursiven Query in Abb. 3.13 berechnen wir alle Städte, die man von Arvin erreichen kann. Die einzige Einschränkung dabei ist der 100 Liter große Tank unseres Autos. Dank der azyklischen Eigenschaft des Straßennetzwerks unserer Miniwelt besteht bei einer rekursiven Query nicht die Gefahr, dass sie nicht terminiert. Das Beispiel, wie auch die fehlerhafte und die korrigierte Query, stammen aus [DG15]. Die Query berechnet die Tabelle `hops(city, gauge)`. In dieser speichert sie alle Städte, die wir von Arvin aus erreichen können und den Resttank, den wir noch übrig haben. Der Basisfall ist unser Startpunkt Arvin, wo wir mit einem leeren Tank starten (Zeile 2).

①SELECT * FROM hops

city	gauge
Arvin	0
Biggs	30
Dixon	40
Biggs	60
Biggs	35
Chico	20

Abbildung 3.14: Ergebnis der rekursiven Query (fehlerhaft)

In jedem Durchlauf führen wir einen Join der Tabellen **cities, roads** und den im letzten Schritt errreichten Städten **h** durch. Stadt `c` ist von Stadt `h` erreichbar, falls es eine Straße `r` gibt, über die `c` mit der aktuellen Tankfüllung erreichbar ist. Die aktuelle Tankfüllung entspricht dem Resttank aus `h`, falls es dort keine Tankstelle gibt. Andernfalls wird der Tank aufgefüllt (Zeile 9). Gibt es eine solche Straße, nimmt die Query die Stadt `c` wie auch den neuen Resttank (Zeile 4 und 5) in die Tabelle **hops** auf. Abb. 3.14 zeigt das Ergebnis. Dort sehen wir auch Chico, welche mit 20 Liter Resttank erreicht werden kann. Die einzige Straße, die nach Chico führt, startet in Biggs und benötigt 40 Liter Treibstoff. Mit einem Starttank von maximal 100 Litern bleiben uns noch maximal 40 Liter für die Fahrt nach Biggs, da es dort keine Tankstelle gibt. Es gibt aber keine Straße in unserer Miniwelt, die höchstens 40 Liter verbraucht und nach Biggs führt. Auch die 4. `(Biggs, 35)` und 5. Zeile `(Biggs, 60)` des Ergebnisses sind fragwürdig. [DG15, S. 2]

①SELECT * FROM hops

city	gauge
Arvin	0
Biggs	30
Dixon	40
Biggs	60
Biggs	35
Chico	20

②hops AS h

city	gauge
Arvin	0
Arvin	0
Dixon	40
Dixon	40
Biggs	30
Biggs	60
Biggs	35
Chico	20

③roads AS r

here	dist	there
'Arvin'	70	'Biggs'
'Arvin'	60	'Dixon'
'Dixon'	80	'Biggs'
'Dixon'	105	'Biggs'
'Biggs'	40	'Chico'

④h.gauge + c.fuel * 100

100
100
140
140
60

Abbildung 3.15: Anzeige der Beobachtungen zur rekursiven Query

Auch hier dient [DG15] als Vorlage des Debuggingprozesses. Einige Schritte (Markierungen ① und ③) weichen jedoch ab. Zuerst lassen wir uns das Endergebnis auf der Benutzeroberfläche anzeigen (Markierung ①). Abb. 3.15 zeigt das Ergebnis der markierten Unterausdrücke. Dann markieren wir das Zwischenergebnis `hops` (siehe ②), welches uns für jedes Tupel aus dem Ergebnis die vorherige Stadt anzeigt. Dort sehen wir, wie oben schon herausgefunden, dass wir nur von Biggs nach Chico kommen. Das entsprechende Tupel finden wir in Beobachtung ① im 2. Rekursionsschritt (Zeile 4). Dort erkennen wir auch, dass uns der Weg von Dixon nach Biggs geführt hat. Die entsprechenden Straßen zeigt Beobachtung ③. Hier sehen wir den Fehler. In Dixon gibt es eine Tankstelle, also sind 100 Liter Treibstoff zur Weiterfahrt verfügbar. Für die Fahrt von Dixon nach Biggs verbrauchen wir entweder 80 oder 105 Liter. Die 2. Straße können wir nicht befahren, da der Tank 5 Liter zu klein ist, also nehmen wir die Erste. Jetzt haben wir nur noch 20 Liter im Tank. Die Beobachtung zeigt aber eine Resttankfüllung von 60 Litern. Woran liegt das? Markierung ④ zeigt, dass der gesamte Tank nach dem Auffüllen 140 Liter fasst, obwohl wir 100 Liter als Maximum festgelegt haben. Wir müssen also bei der Berechnung berücksichtigen, dass beim Auftanken - auch wenn noch ein Rest im Tank ist - maximal 100 Liter im Tank sind. Abb. 3.16a zeigt die korrigierte Query, bei der wir mit `LEAST` sicherstellen, den maximalen Tank von 100 Litern nicht zu überschreiten. Sie liefert nun das korrekte Ergebnis, welches in Abb. 3.16b zu sehen ist.

```
1  WITH RECURSIVE hops(city, gauge) AS (
2          VALUES ('Arvin', 0)
3      UNION ALL
4          SELECT  r.there AS city,
5                  LEAST(100, h.gauge + c.fuel * 100)
6                  - r.dist AS gauge
7          FROM    cities AS c, roads AS r, hops AS h
8          WHERE   h.city = c.city
9          AND     h.city = r.here
10         AND     LEAST(100, h.gauge + c.fuel * 100)
11                 >= r.dist
12 )
13 SELECT *
14 FROM hops;
```

(a) aus [DG15]; korrigierte Query

SELECT * FROM hops

city	gauge
Arvin	0
Biggs	30
Dixon	40
Biggs	20

(b) Ergebnis der korrigierten Query

Abbildung 3.16: Korrekte Ermittlung aller Städte, die von Arvin erreichbar sind

3.4 NOT IN und NOT EXISTS

Nun suchen wir alle Berufe, die es in der Miniwelt gibt, die aber keine der Personen aus Tabelle **people** ausführt. Dazu testen wir mit Hilfe der Query in Abb. 3.17 für jeden Beruf aus der Tabelle **jobs** (Zeile 2), ob es eine Person in **people** gibt, die diesen Beruf ausführt (Zeile 3 und 4). Wenn dies nicht der Fall ist, nimmt die Query diesen Beruf in ihr Ergebnis auf.

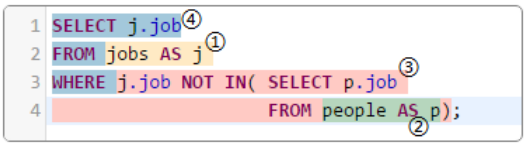


Abbildung 3.17: Ermittlung der nicht ausgeführten Berufe (fehlerhaft)

Die Ergebnismenge dieser Query ist leer. Dabei wissen wir, dass keine der Personen aus **people** den Beruf Bäcker ausübt, dieser aber in **jobs** existiert. Also müssen wir auch diese Query debuggen. Das Markieren der gesamten Query nützt uns ins diesem Fall zuerst nichts, da das Ergebnis leer ist. Wir markieren zuerst die Tabellen **jobs** und **people**. In **jobs** filtern wir nach der Zeile `Baker` (Bäcker), da dieses Tupel dasjenige ist, das fälschlicherweise nicht im Ergebnis auftaucht. Zu Vergleichszwecken nehmen wir auch die Zeile `Carpenter` (Schreiner) mit in den Filter auf. Abb. 3.18 zeigt die gefilterte Beobachtung dieser Markierungen. Auf die Frage, ob der Beruf `Baker` von jemandem aus Tabelle

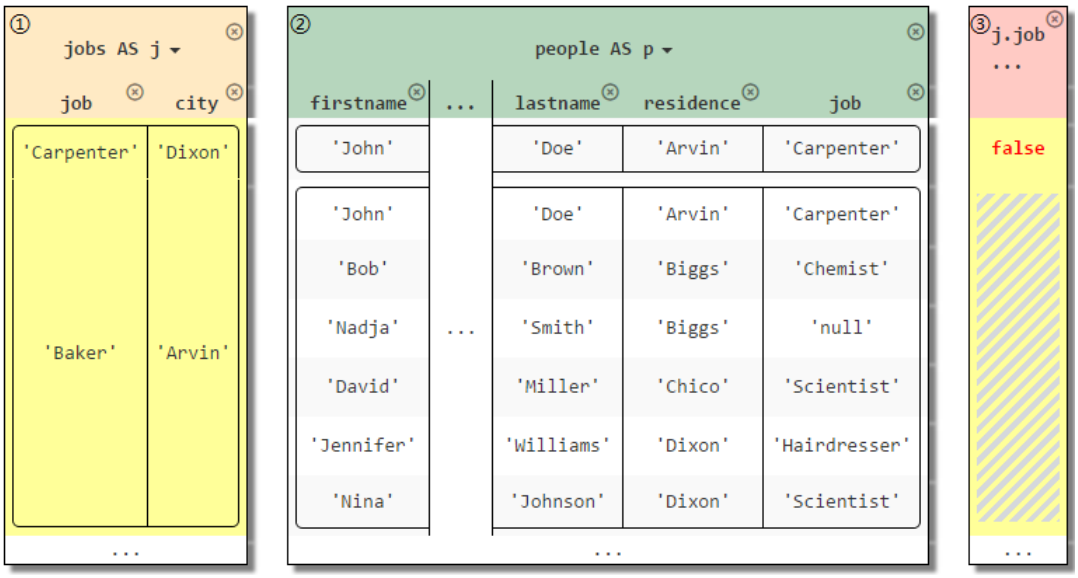


Abbildung 3.18: Anzeige der Beobachtungen zur Query aus Abb. 3.17

people ausgeführt wird, erhalten wir die Antwort `null` (Markierung ③). Beim `Carpenter` liefert sie die Antwort `false`. PostgreSQL wertet die innere Abfrage nur solange aus, bis es ein Tupel findet, das mit dem Äußeren übereinstimmt [Pos17e]. Im Fall des Bäckers gibt es kein solches Tupel. Aber es gibt ein Tupel, das den Wert `null` in der Spalte `job` besitzt. In Bsp. 3.2 haben wir gesehen, dass der Vergleich von `null`-Werten leicht zu Fehlern führen kann. Ob auch hier der Fehler daher rührt, können wir herausfinden, indem wir in der Spalte `job` der Tabelle **people** den Wert `null` verändern. Wir geben in Textform an, dass die Person Nadja Smith arbeitslos ist. Die veränderte Beobachtung zeigt Abb. 3.19. Nun ergibt die `NOT IN` Abfrage

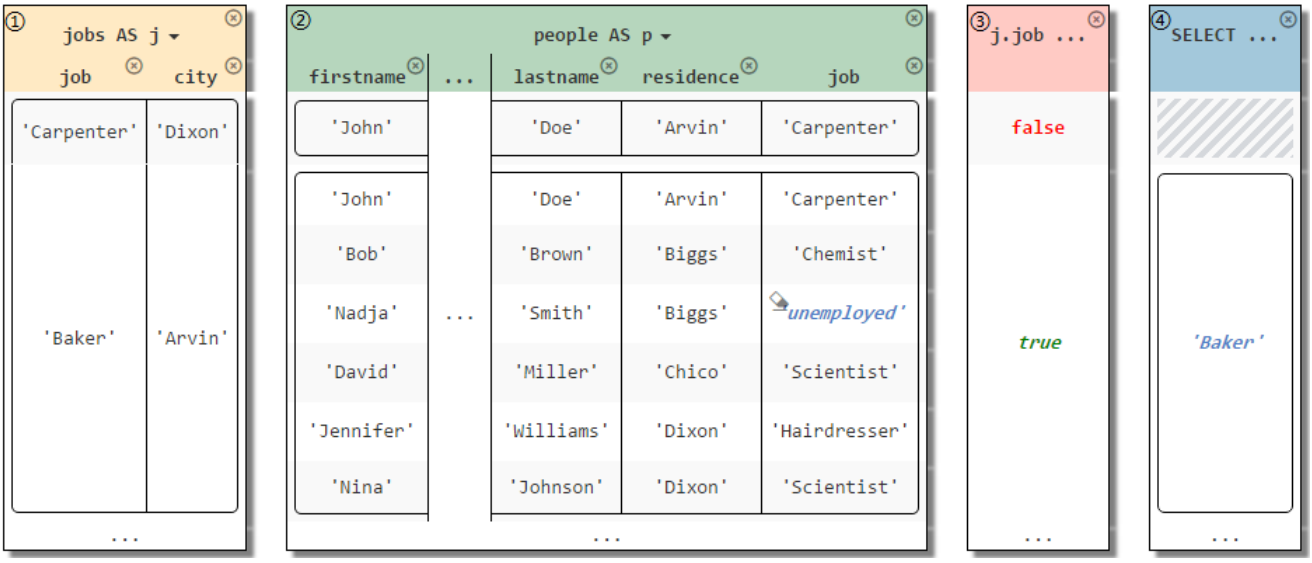


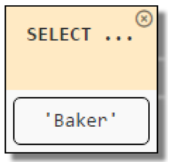
Abbildung 3.19: Anzeige der Beobachtungen zur Query aus Abb. 3.17, mit veränderter Darstellung der Arbeitslosigkeit

`true` und der Beruf Bäcker taucht im Endergebnis auf. Also liegt auch hier der Fehler beim Vergleich mit `null`. Die PostgreSQL Dokumentation liefert uns die entsprechende Erklärung: Gibt es keine passenden Werte im Unterausdruck auf der rechten Seite (hier `SELECT p.job FROM people AS p`) und ist mindestens einer von ihnen `null`, so ergibt das `NOT IN` Konstrukt `null` [Pos17e]. Genau das trifft hier zu.

Eine Möglichkeit, das korrekte Ergebnis zu erhalten, wäre die veränderte Darstellung der Arbeitslosigkeit in der Datenbanktabelle, wie oben durch die „What If“-Anfrage getestet. Wollen wir diese aber im Ursprungszustand belassen, wählen wir den nachfolgenden Weg. Die Fragestellung der Ausgangs-Query lautet ähnlich derer aus Bsp. 3.2. Wir suchen alle Berufe, für die es keine Person aus Tabelle **people** gibt, die diesen Beruf ausführt. Außerdem lässt sich jede Query, die `IN` bzw. `NOT IN` verwendet, problemlos in eine Query mit `EXISTS` bzw. `NOT EXISTS` umwandeln. Also können wir die Antwort auch durch Verwendung des `NOT EXISTS` Konstrukts finden (siehe Abb. 3.20a). Die Ausführung dieser Query liefert nun das erwartete Ergebnis aus Abb. 3.20b.

```
1 SELECT j.job
2 FROM jobs j
3 WHERE NOT EXISTS (  SELECT 1
4                     FROM people p
5                     WHERE j.job = p.job);
```

(a) korrigierte Query



(b) Ergebnis der korrigierten Query

Abbildung 3.20: Korrekte Ermittlung der nicht ausgeführten Berufe

Kapitel 4

Implementierung

Im ersten Teil dieses Kapitels werden die verwendeten Technologien, wie Programmiersprachen, Bibliotheken und ähnliches erläutert. Später wird die Implementierung der wichtigsten aus Kapitel 3 bekannten und weiteren Funktionalitäten behandelt.

4.1 Verwendete Technologien

Die im Rahmen dieser Arbeit entwickelte Benutzeroberfläche für Habitat besteht aus einer HTML-Datei, einer CSS-Datei sowie mehreren JavaScript-Dateien. Diese Sprachen werden im Folgenden genauer betrachtet. Danach folgt eine kurze Beschreibung der verwendeten Plugins, Bibliotheken und Frameworks, die für die Implementierung verwendet wurden.

4.1.1 Sprachen

Das Programm wurde in den bekanntesten und aktuellsten Sprachen der clientseitigen Webprogrammierung geschrieben. Im Folgenden werden diese sowie deren wichtigsten Eigenschaften im Bezug auf das entwickelte Programm erläutert.

HTML

Die Hypertext Markup Language (kurz HTML) ist der Standard zur Darstellung von Webinhalten. Webbrowser interpretieren HTML Deklarationen und stellen sie dar. Nach [H+14] besteht jedes HTML-Dokument aus einem Baum von Elementen und Text. Ein „start tag“, wie z. B. „`<div>`“ und ein „end tag“, wie z. B. „`</div>`“, kennzeichnen jedes der Elemente. Die Baumstruktur entsteht durch das Schachteln der Elemente. Überlappungen sind dabei nicht erlaubt. Elemente können Attribute besitzen, die ihre Funktion bestimmen.

Die Attribute „id“ und „class“ ermöglichen es, bestimmte Elemente anzusprechen. Für die im Rahmen dieser Arbeit entwickelte Benutzeroberfläche wird HTML5 verwendet. Ein HTML-Dokument startet mit der Deklaration des „DOCTYPEs“. Danach folgt die Wurzel des Dokuments - das „html“-Element. Sie enthält die beiden Elemente „head“ und „body“. Das „head“-Element dient der Speicherung von Metadaten des Dokuments, wie z. B. dem Titel. Das „body“-Element enthält den gesamten Inhalt des Dokuments. [H+14]
Zum Ansprechen der Elemente über CSS oder eine Skriptsprache existiert eine standardisierte Schnittstelle des W3C - das Document Object Model (DOM). Hierdurch kann der Inhalt des Dokuments sowie dessen Struktur und Layout verändert werden. DOM stellt die Objekthierarchie in einer baumartigen Struktur dar, welche die Beziehungen der Elemente zueinander zeigt. Die Schnittstelle erlaubt eine nahezu browserunabhängige Programmierung. [Koc07, S. 165 f]

CSS

Die Gestaltung einer Website erfolgt üblicherweise durch CSS (Cascading Style Sheets). Die Technologie schafft die Trennung zwischen dem Inhalt und dem Layout eines Dokuments [Koc07, S. 157 f]. Mit Hilfe von CSS kann z. B. die Größe, Farbe oder Transparenz eines Elements festgelegt werden. CSS-Definitionen können direkt am „tag“ eines Elements angegeben werden, im „head“ des Dokuments oder in einer separaten Datei [Lab06, S. 63 f]. CSS spricht Elemente üblicherweise über den „tag“-Namen, eine Klasse oder eine ID an. Für komplexere Aufgaben sind kombinierte Selektoren sowie die Verwendung von Pseudo-Klassen möglich [Lab06, S. 41 ff].

JavaScript

Die Skriptsprache JavaScript kann sowohl clientseitig als auch serverseitig eingesetzt werden. Hauptsächlich wird sie für die clientseitige Webprogrammierung eingesetzt, in der sie sehr weit verbreitet ist. Als Skriptsprache wird sie vom Browser des Anwenders interpretiert [Koc07, S. 15 ff]. Durch die Verwendung einer Skriptsprache, wie JavaScript, ist die Darstellung dynamischer Webseiten möglich [Koc07, S. 157]. JavaScript ermöglicht es, direkt auf das DOM einer Webseite zuzugreifen. Das geschieht durch den „tag“-Name eines Elements oder deren ID [Koc07, S. 172]. Durch diesen Zugriff auf das DOM können Elemente zum Dokument hinzugefügt, gelöscht oder verändert werden. JavaScript kann auch die CSS-Definitionen verändern [Koc07, S. 165]. Eine der für die Entwicklung der Oberfläche interessantesten Funktionalitäten, ist das Anlegen von Event-Handlern. Durch sie ist es möglich, auf Aktionen des Benutzers zu reagieren [Koc07, S. 181]. In der entwickelten browserbasierten Benutzeroberfläche macht JavaScript den Großteil des Programms aus.

4.1.2 Bibliotheken, Frameworks und Plugins

Die entwickelte Benutzeroberfläche verwendet eine Reihe an Bibliotheken, Frameworks und Plugins. Sie sind alle frei verfügbar und die meisten werden laufend weiterentwickelt und verbessert.

jQuery

jQuery[1] ist eine weit verbreitete JavaScript-Bibliothek, die u. a. die DOM-Manipulation, Ajax-Aufrufe (Asynchronus JavaScript and XML) und die Arbeit mit Events vereinfacht. Der Zugriff auf Gruppen von Elementen zur Animation gestaltet sich über JavaScript oft schwierig. Durch jQuery lassen sich mehrere Zeilen kompakt zusammenfassen, was den Programmcode übersichtlicher macht [BK08, S. 2]. Die Benutzeroberfläche verwendet Version 3.1.1.

jQuery UI Diese Erweiterung von jQuery enthält Funktionen zur Unterstützung der Nutzerinteraktion auf Benutzeroberflächen [BK08, S. 299 f]. Mit Hilfe der von jQuery UI[2] bereitgestellten Funktion `draggable` kann in der entwickelten Benutzeroberfläche die Größe des Editors vom Benutzer verändert werden. Wir verwenden Version 1.11.4.

Bootstrap

Bootstrap[3] ist ein Open-Source-Framework zur Gestaltung von Front-Ends. Entwickelt wurde Bootstrap durch Mitarbeiter von Twitter [Ott12]. Das Framework enthält Vorlagen zur Gestaltung der meisten HTML-Elemente, wie z. B. Tabellen, Schaltflächen oder Eingabefelder. Auch für Elemente, wie z. B. Menüs, bietet das Framework Gestaltungsmöglichkeiten. Außerdem stellt Bootstrap Bildzeichen („glyphicons“) bereit. Über die Zuweisung vordefinierter CSS-Klassen können Entwickler Elemente ihre Webseiten nach den Bootstrap-Vorlagen gestalten (vgl. [OTa] und [OTb]). JavaScript-Funktionen erlauben es u. a., die Reaktion von Elementen auf Aktionen des Benutzers zu aktivieren [OTc]. Für die Benutzeroberfläche verwenden wir Version 3.3.7.

Bootstrap FileStyle HTML-Felder zum Hochladen von Dateien lassen sich, anders als andere Elemente, schwer über CSS-Regeln gestalten. Deshalb verwendet die Benutzeroberfläche das Plugin Bootstrap Filestyle[4] (Version 1.2.1) zur Gestaltung dieser Felder. Das in JavaScript geschriebene Plugin benötigt jQuery und Bootstrap [Lim].

[1] http://jquery.com/
[2] http://jqueryui.com/
[3] http://getbootstrap.com/
[4] http://markusslima.github.io/bootstrap-filestyle/

Editor

Der Editor ist eins der wichtigsten Elemente bei der Nutzung der Oberfläche. Um die Bedienung möglichst komfortabel zu gestalten, ist es sinnvoll, anstelle eines einfachen HTML-Textfelds einen vordefinierten Editor zu verwenden. Zu Beginn der Implementierung standen zwei in JavaScript implementierte Texteditoren zur Auswahl: ACE[5] und CodeMirror[6]. Beide wurden zur Verwendung in Webanwendungen entwickelt. Sie stellen die Eigenschaften und Funktionen der normalen Editoren bereit. Darunter zählen u. a. Syntax-Highlighting und das automatische Schließen von Klammern (vgl. [Ace] und [Cod]). Das wichtigste Kriterium für die Verwendung in HabitatUI ist das Anlegen von Markierungen im Text. Auch das gehört zur Ausstattung beider Editoren.
Zur Entwicklung der Benutzeroberfläche fiel die Entscheidung auf CodeMirror (Version 5.20.2). Das Handbuch hierzu ist übersichtlich und enthält eine ausführliche Erklärung der Funktionen. Die Programmierung und Konfiguration des Editors ist intuitiv und gut auf unterschiedliche Verwendungszwecke anzupassen. Außerdem passt der Editor Grenzen eines markierten Bereichs automatisch an, falls sich der Inhalt des Editors ändert.

Dragtable

Habitat UI stellt die Beobachtungen zu den markierten Teilausdrücken in einer oder mehreren Tabellen dar. Hierbei kann es für den Benutzer bei vielen Teilbeobachtungen leicht unübersichtlich werden. Um zwei Beobachtungen nebeneinander zu stellen, nutzen wir das jQuery-Plugin dragtable[7] (Version 1.0). Es ermöglicht dem Nutzer, Spalten einer Tabelle zu verschieben [Van08]. In ursprünglicher Form funktioniert das Plugin nur für Tabellen, die keine Zellen enthalten, die mehrere Spalten überspannen. Da die Tabelle in der Benutzeroberfläche aber solche Zellen enthält, mussten wir das Plugin im Rahmen dieser Arbeit anpassen. Es ist speziell an die Verwendung in Kombination mit dem entwickelten Programm angepasst und kann deshalb in dieser Form nicht ohne Weiteres in andere Umgebungen übernommen werden.
Andere Plugins, die dieselbe Funktion implementieren, stellten teilweise größere Probleme bei der Anpassung an unsere Anforderungen dar. Oder sie starteten das Verschieben schon bei einem Klick in den Tabellenkopf, sodass dort das Auslösen anderer Funktionen nicht möglich ist, wie z. B. das Löschen von Spalten, was später genauer erläutert wird.

[5] https://ace.c9.io/
[6] https://codemirror.net/
[7] http://www.danvk.org/wp/dragtable/

DataTables

Die Beispielszenarios aus Kapitel 3 liefern Beobachtungen, die wenige Tabellenzeilen belegen. Sie werden schnell geladen und schnell angezeigt. Realistische Datenbanktabellen enthalten viel mehr Tupel, daher werden die Beobachtungen schnell sehr groß. Die Anzeige großer Tabellen ist langsam und führt dazu, dass der Benutzer mehrere Minuten auf die Darstellung warten muss. Um das zu vehindern, zeigt Habitat UI nur einen Teil der Tabellenzeilen auf einmal an. Diese und noch weitere Funktionalitäten stellt das jQuery-Plugin DataTables[8] bereit. Für die Benutzeroberfläche nutzen wir außerdem die Suchfunktion des Plugins. So ist es möglich, in einer Tabelle nach Tupeln zu suchen, die einen bestimmten Wert enthalten [Spr17]. Die Benutzeroberoberfläche verwendet das Plugin in der Version 1.10.12.

4.2 Überblick über die Benutzeroberfläche

Vor der detallierten Erklärung der wichtigsten Funktionen verschaffen wir uns einen Überblick über die Benutzeroberfläche. Abb. 4.1 zeigt das Debuggingszenario aus Bsp. 3.1. Die Oberfläche besteht aus der Navigationsleiste, einem Formular für den Verbindungsaufbau zur Datenbank sowie dem Eingabe- und dem Ausgabebereich. Die Navigationsleiste enthält neben der Schaltfläche zum Aus- und Einblenden des Verbindungsformulars („Connect“) und dem „reset“-Button zum Zurücksetzen der Markierungen und Filter, auch ein Popover zum Anzeigen einer minimalen Anleitung und eine Schaltfläche zum Hochladen einer Datei.
Über das Verbindungsformular kann sich der Nutzer mit seinem Benutzernamen und dem Namen der Datenbank mit dieser verbinden. Optional kann er ein Passwort angeben, welches zur Authentifizierung auf der Datenbank dient. Bevor das Programm die Daten in einer Verbindungsanfrage an den Server weitergibt, testet es, ob die benötigten Daten - Nutzername und Name der Datenbank - einen Wert enthalten. Ist eines dieser Felder leer, erfährt dies der Nutzer durch Rotfärbung des Eingabefeldes. Das Programm sendet die Anfrage erst, wenn beide Felder ausgefüllt sind, um den Datenverkehr zwischen Client und Server zu minimieren. Die Anmeldung auf der Datenbank muss vor dem Anlegen der ersten Markierung geschehen. Während der Arbeit auf der Oberfläche kann sich der Nutzer jederzeit erneut anmelden.
Durch die Schaltfläche zum Hochladen einer Datei ist es nicht notwendig, die Query jedesmal von Hand in den Editor einzugeben. Zur Implementierung dieser Funktion verwenden wir das HTML-Element `input` mit dem Typ `file` und den JavaScript FileReader. Das Programm erlaubt nur Text- oder SQL-Dateien. Die Metainformationen zur ausgewählten Datei enthalten diese

[8] https://datatables.net/

Informationen. Der FileReader stellt verschiedene Funktionen zum Lesen einer Datei bereit. In unserem Fall ist nur die Datenbankabfrage in Textform interessant. Daher verwenden wir die Funktion `readAsText`. Dabei fügt das Programm den Inhalt der Datei in den Editor ein. Zur Gestaltung der Schaltfläche verwenden wir das Plugin FileStyle. Mit dessen Hilfe entspricht die Gestaltung dem Rest des Dokuments.

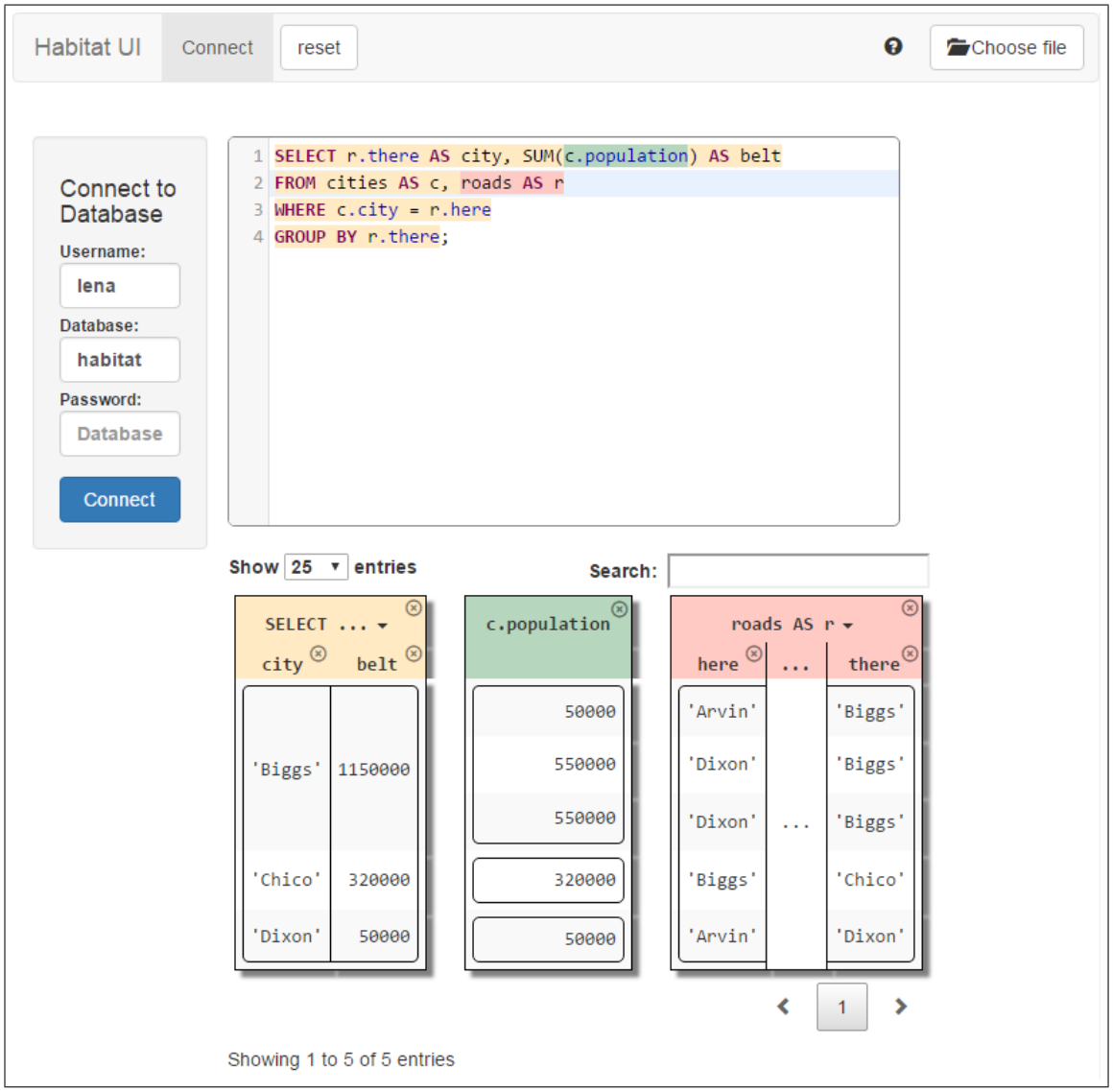


Abbildung 4.1: Überblick über die Benutzeroberfläche HabitatUI

Der Eingabebereich enthält den Editor zur Eingabe und Markierung der Query. Er ist so konfiguriert, dass er wie ein normaler Editor wirkt, dem Syntax-Highlighting für SQL-Queries bekannt ist. Für verschiedene Szenarios ist es

nützlich, die Größe des Editors zu verändern. Da CodeMirror hierfür keine entsprechende Funktion liefert, nutzen wir die Funktion `draggable`, die jQueryUI bereitstellt. Sobald ein Nutzer das unsichtbare `div`-Element mit der ID „resizer“, welches in der rechten unteren Ecke des Editors platziert ist, verschiebt, verändert sich die Größe des Editors so, dass sich die Position des `resizers` immer in der rechten unteren Ecke des Editors befindet. Ist der Nutzer mit einer Datenbank verbunden, sendet das Programm bei jeder Änderung des Editorinhalts die Query über die entsprechende Anfrage an den Server, welche in Kapitel 2.2 besprochen wurde.
Der Ausgabebereich enthält die Beobachtungen der markierten Ausdrücke in Tabellenform. Zusammen mit dem Eingabebereich stellt er den Hauptteil des Dokuments dar. Hier findet die Hauptaktion des Nutzers statt.

4.3 Kommunikation mit dem Server

Die entwickelte Benutzeroberfläche kommuniziert über die in Abschnitt 2.2 besprochene Schnittstelle über HTTP mit dem Server. Bei synchronen (im Programmfluss ablaufenden) Anfragen an den Server, wartet das Programm auf dessen Antwort. Erst wenn der Client diese erhalten hat, arbeitet das Programm weiter. Benötigt der Server einige Zeit zum Antworten, kann das den Browser zum „einfrieren“ bringen. Der Nutzer kann in dieser Zeit nichts tun, bis die Antwort eingetroffen ist.
Zur Verhinderung solcher Einschränkungen läuft die Kommunikation mit dem Server asynchron ab. Dafür verwenden wir Ajax, was für Asynchronus JavaScript and XML steht. Ajax ist eine Methode, mit der ein JavaScript-Programm Anfragen an den Server sendet, sobald Nutzeraktionen dies erfordern. Solange der Server die Anfrage verarbeitet, kann der Nutzer die Webseite weiter bedienen. Unabhängig davon, ob die Antwort des Servers eingetroffen ist oder nicht, kann die Bedienung der Oberfläche ohne Einschränkungen weitergeführt werden. Der Server sendet die Daten meist als XML oder JSON-Objekt [Koc07, S. 331 ff]. In unserem Fall verwendet er JSON-Objekte.
Die Datei `serverCommunication.js` enthält die Definition aller Funktionen zur Kommunikation mit dem Server. Über den von jQuery bereitgestellten Befehl `$.ajax()` sendet das Programm eine asynchrone Anfrage. Die Konfigurationen entsprechen den Festlegungen aus Kapitel 2.2. Der Habitat-Server lauscht auf dem Port 8081. Daher setzt ihn das Programm standardmäßig auf diesen Wert. Da die Anfragen asynchron ablaufen, stellen sie die weitere Bearbeitung vor kleinere Probleme. Nehmen wir im Folgenden an, wir fordern vom Server die Beobachtung für das aktuelle Szenario an. Das Programm sendet also eine Ajax-Anfrage an den Server. Der nächste Schritt ist die Übersetzung des erhaltenen JSON-Objekts in eine HTML-Tabelle, die der Browser darstellt. Da alle Anfragen asynchron ablaufen, führt der Browser den Pro-

grammfluss fort, ohne auf die Antwort des Servers zu achten. Deshalb kommt es dazu, dass zuerst die Übersetzung des JSON-Objekts gestartet wird, bevor die aktuelle Beobachtung ankommt. Das führt zu einem Fehler, da das JSON-Objekt, in dem die Beobachtung enthalten sein soll, leer ist oder eine veraltete Beobachtung enthält. Um dieses Problem zu umgehen, gibt es den jQuery-Befehl `$.when().then()`, welcher die Ausführung der Befehle im `then`-Teil erst ausführt, nachdem der Aufruf im `when`-Teil beendet ist [jQu]. So findet die Abarbeitung der Befehle in korrekter Reihenfolge statt, während der Nutzer die Oberfläche weiterhin uneingeschränkt bedienen kann.

4.4 Umgang mit Markierungen

Eine der Grundfunktionen der Benutzeroberfläche ist der Umgang mit Markierungen von Teilabfragen. Dazu zählt das Anlegen von Markierungen und deren Erhaltung bei Überlappungen sowie das Entfernen. In diesem Abschnitt geht es hauptsächlich um die visuelle Darstellung der markierten Teilabfragen, aber auch um die interne Speicherung der notwendigen Informationen.

4.4.1 Anlegen und Erhaltung der Markierungen

Bei jedem `onmouseup`-Event, das im Editor stattfindet, testet das Programm den ausgewählten Abschnitt der Datenbankabfrage, ob dieser eine gültige Markierung ist. Das geschieht durch das Anlegen dieser Markierung auf dem Server. Als Parameter übergibt das Programm den Start- und Endpunkt der Auswahl im Editor. Antwortet dieser mit einem Fehler, geschieht nichts. Ansonsten speichert das Programm die vom Server erhaltenen Informationen der Markierung ab und legt einen „Marker“ an der Stelle der Textauswahl an. Dazu nutzen wir die von CodeMirror bereitgestellte Klasse `TextMarker`. Ein solcher Marker speichert die Anfangs- und Endposition der Markierung und definiert deren Hintergrundfarbe, welche für die Unterscheidung der Markierungen sorgt. Der Marker erhält die Zuweisung zu einer CSS-Klasse, welche dessen Hintergrundfarbe festlegt. Als Positionen, die den Marker begrenzen, verwenden wir die in der Antwort des Servers enthaltenen Start- und Endpositionen der Markierung. Zum jetztigen Zeitpunkt ist es möglich, stattdessen die Positionen zu übernehmen, die wir an den Server übergeben. Da aber Habitat in Zukunft Markierungen zu einer korrekten Teilabfrage erweitern soll, falls sie es nicht schon sind, können sich die vom Server gelieferten Positionen von den lokal Bestimmten unterscheiden.
Beim Anlegen eines solchen Markers sorgt eine Funktion dafür, alle Markierungen sichtbar zu halten. Dazu vergleicht sie die Grenzen der neuen Markierung mit allen Anderen. Liegt eine andere Markierung innerhalb der Neuen, muss sie neu hinterlegt werden, sodass sie sichtbar bleibt. Zur

Realisierung dieser neuen Hinterlegung ist die Verwendung der CSS-Klassen, die die Gestaltung beim erstmaligen Anlegen des Markers festlegen, nicht möglich. Zur Erklärung müssen wir uns die DOM-Darstellung der Marker in CodeMirror ansehen. Die Stelle des Textes, an der ein Marker angelegt ist, separiert CodeMirror vom Rest des Textes durch die Verwendung eines `span`-Elements, das diese umschließt. Der Marker kann aus mehreren `span`-Elementen zur Darstellung des Syntax-Highlightings bestehen. Alle zu diesem Marker gehörenden `spans` erhalten die ihm zugewiesene Klasse, die den Hintergrund in der entsprechenden Farbe einfärbt. Ist nun ein weiterer Marker innerhalb eines Anderen angelegt, erhalten die zu ihm gehörenden `spans` die Klassen beider Marker. Verwenden wir nun die definierten Klassen zur Anlegung der Markierung, erhält der Marker die Hintergrundfarbe der zuletzt angelegten Markierung. Das liegt an der Reihenfolge der Definition der CSS-Klassen. Für die Zuordnung der Farben verwenden wir die CSS-Klassen in der Reihenfolge, in der sie angelegt sind. Nach [B+11, S. 91] setzt sich bei zwei gleich spezifischen und gleich wichtigen Klassen aus derselben CSS-Datei, die beide dieselbe Eigenschaft beeinflussen - wie bei uns die Hintergrundfarbe - diejenige durch, die zuletzt definiert ist. Wie oben erwähnt, ist das diejenige, die den Hintergrund der zuletzt angelegten Markierung bestimmt.

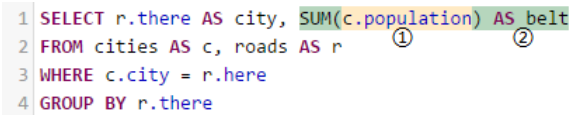


Abbildung 4.2: Beispiel zur Erhaltung der Markierungen bei Überschneidungen

Legen wir in Abb. 4.2, welche die Query aus Bsp. 3.1 enthält, Markierung ② vor Markierung ① an, funktioniert das Hinterlegen automatisch, während die Markierung in umgekehrter Reihenfolge die oben beschriebenen Probleme hervorruft. Um auch bei Markierungen in dieser Reihenfolge alle Marker sichtbar zu machen, legen wir beim Erhalten der alten Markierung eine neue CSS-Klasse an, welche dieselben Vorschriften enthält, wie auch die ursprünglich zugewiesene Klasse. Diese Klasse ist nun die letzte Definierte. Deshalb erhält der markierte Teil des Textes die richtige Hintergrundfarbe. Die Funktion zur Erhaltung der Markierung ruft sich rekursiv auf und prüft dann für die in der neuen Markierung enthaltenen Markierung, ob diese weitere Markierungen enthält. Das Ende der Rekursion ist erreicht, sobald die betrachtete Markierung keine Andere mehr enthält.
Für eine gute Verwendbarkeit der Oberfläche für den Benutzer enthält das Programm mehrere kleine Funktionalitäten. Eine davon dient dem Schutz vor dem Anzeigen riesiger Ergebnisse, die nicht unbedingt notwendig sind. Bisher prüft das Programm, ob die angelegte Markierung der Beobachtung einer

Ausgangstabelle entspricht. Da diese im Normalfall sehr groß sind, ist das eine der Situationen, in denen eine Warnung sinnvoll ist. Das Programm erkennt eine solche Abfrage an der Position in der Query. Datenbanktabellen werden in der `FROM` Klausel einer Query angesprochen. Daher testen wir, ob die Markierung zwischen `FROM` und einem weiteren SQL-Schlüsselwort plaziert ist. Nur das Schlüsselwort `AS` ist von diesen ausgeschlossen, da es in der `FROM` Klausel vorkommt. Hat der Nutzer eine solche Teilabfrage markiert, erhält er einen Hinweis darauf, dass das gewünschte Ergebnis möglicherweise sehr groß ist. Möchte er das Ergebnis trotzdem anzeigen, reicht das Drücken der `Enter`-Taste aus. Ansonsten genügt eine Nutzeraktion, wie z. B. ein Klick, um die Markierung rückgängig zu machen. Abb. 4.3 zeigt den Hinweis nach der Markierung einer Ausgangstabelle in der Query aus Bsp. 3.4. Will der Nutzer einen Teil der Query ändern, ist es oft angenehm, einen Teil davon auszuwählen. Um in diesem Fall keine Markierung anzulegen, muss die Auswahl allein über die Tastatur geschehen oder die `Shift`-Taste bei der Auswahl über die Maus gedrückt sein.

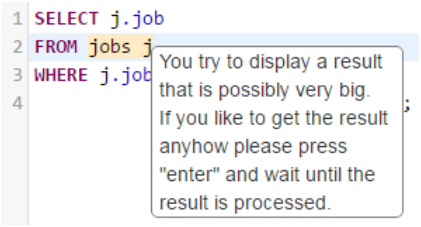
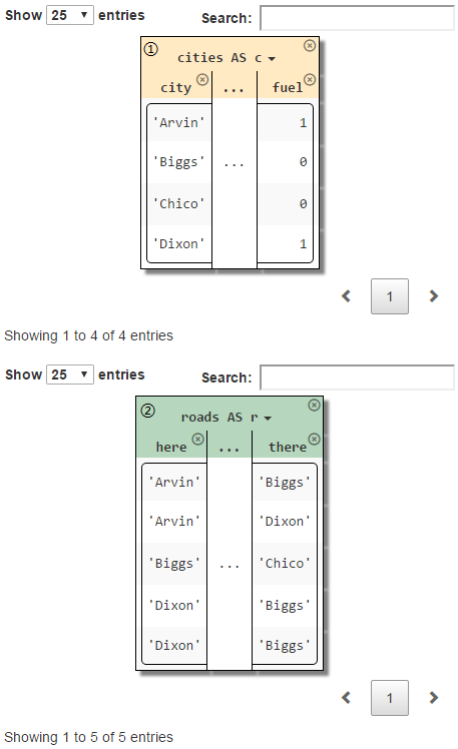


Abbildung 4.3: Hinweis auf ein möglicherweise großes Ergebnis

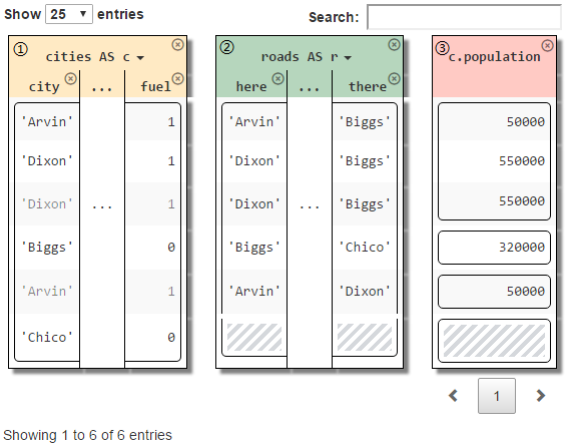
4.4.2 Markierungen löschen

Oft bemerkt man erst nach dem Anlegen einer Markierung, dass diese nichts Interessantes zur bisherigen Beobachtung beiträgt. Sind mehrere solcher Markierungen angelegt, führt das schnell zu einer unübersichtlichen Ergebnistabelle. Außerdem muss der Server große Beobachtungen an den Client liefern, die der Benutzer gar nicht benötigt. Um dieser Situation zu entkommen, bietet die Benutzeroberfläche die Möglichkeit, eine Markierung zu entfernen. Dazu genügt das Drücken des ⓧ-Buttons im Tabellenkopf der entsprechenden Beobachtung.
Beim Löschen deaktiviert das Programm die Markierung auf dem Server, um diesem mitzuteilen, dass für weitere Beobachtungen das Ergebnis dieser Markierung nicht mehr benötigt wird. Die `id` des Tabellenkopfes, der den gedrückten ⓧ-Button enthält, verrät die Zuordnung zur korrekten Markierung, da sie aus deren `mid` besteht. Das Programm entfernt den entsprechenden Marker und passt alle Strukturen an. Das Anpassen der Tabelle ist der anspruchvollste Teil des Löschvorgangs. Es gibt zwei Möglichkeiten, diesen durchzuführen. Die erste Möglichkeit ist, die Beobachtung erneut vom Server abzurufen. Nachdem die entfernte Markierung deaktiviert ist, liefert der Server nur die Beobachtungen der anderen Markierungen zurück. Also enthält die neu dargestellte Tabelle nur die Beobachtungen, die gewünscht sind. Der Nachteil daran ist die erneute Kommunikation zwischen Client und Server. Deshalb haben wir uns

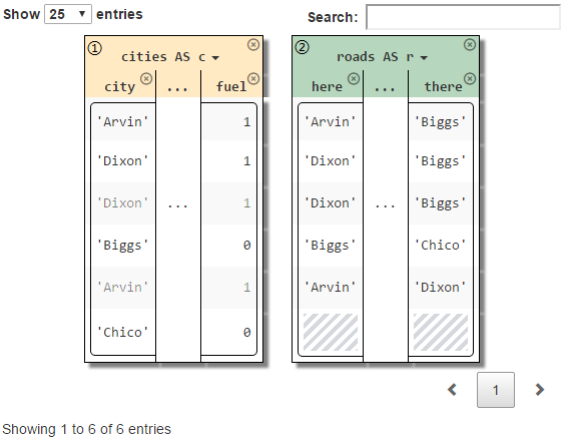
für die zweite Möglichkeit entschieden. Diese entfernt den Teil der Tabelle, der die ungewollte Beobachtung enthält. Der Nachteil dieses Ansatzes ist, dass die Beobachtungen weiterhin in einer gemeinsamen Tabelle angezeigt werden, auch wenn die Beobachtung, die dafür sorgt, dass es eine gemeinsame Tabelle ist, entfernt wird. In Abb. 4.4a sehen wir Markierung ① und ②, die in separaten Tabellen enthalten sind, da sie nichts miteinander zu tun haben. Erst durch das Hinzufügen der Markierung ③ (siehe Abb. 4.4b) können sie zueinander in Bezug gestellt und damit in einer gemeinsamen Tabelle dargestellt werden. Entfernen wir nun Markierung ③ bleiben die beiden ersten Markierungen in einer gemeinsamen Tabelle erhalten, obwohl nun der Bezug zwischen ihnen unklar ist (siehe Abb. 4.4c).



(a) Beobachtung vom Server geliefert



(b) Die Beobachtungen stehen in Beziehung zueinander



(c) Beobachtung nach dem Löschen

Abbildung 4.4: Darstellung der Beobachtungen vor und nach dem Löschen einer Markierung

Beim Löschen ist es wichtig darauf zu achten, dass eine Beobachtung möglicherweise mehrere Tabellenspalten überspannt. Ist das der Fall, müssen all diese Spalten entfernt werden. Auch Beobachtungen, die so groß sind, dass nur ein Teil der Tupel angezeigt wird, benötigen besondere Beachtung. Tupel, die nicht auf der aktuellen Tabellenseite sind, sind nicht über den DOM-Baum der Seite ansprechbar. Hierzu müssen wir die restlichen Tupel über die von DataTables bereitgestellten Funktionen ansprechen. Sind nach dem Löschen aller zur gelöschten Markierung gehörenden Tabellenzellen entfernt, kann es dazu kommen, dass es nun Zeilen gibt, die in keiner Spalte eine Beobachtung enthalten. Diese Zeilen löscht das Programm aus der Tabelle, um nur aussagekräftige Informationen in der Tabelle zu erhalten.

In verschiedenen Szenarien ist es sinnvoll, die Oberfläche auf den Urzustand zurückzusetzen. Hat der Benutzer zu viele Teilausdrücke markiert und den Überblick verloren, bietet es sich an, alle Markierungen zu löschen und von vorn zu beginnen. Dazu stellt das Programm eine Funktion bereit, welche alle Markierungen auf dem Server deaktiviert und im Editor entfernt. Sie verbirgt sich hinter dem Button **reset**. Auch Filter lassen sich dadurch zurücksetzen. In Abschnitt 4.6 wird dies genauer beschrieben.

4.5 Die Funktion `getTable`

Neben dem Anlegen und Erhalten von Markierungen ist die Übersetzung des JSON-Objekts der Beobachtung in die HTML-Tabelle die Kernfunktion der entwickelten Benutzeroberfläche. Der Nutzer erwartet die Darstellung der Ergebnisse seiner markierten Unterausdrücke in geeigneter Form. Dazu gehört die unterschiedliche Darstellung tabellarischer und atomarer Beobachtungen sowie die unterschiedliche Darstellung verschiedener Datentypen. Außerdem ist die Darstellung von in Bezug stehenden Beobachtungen wichtig.

Das vom Server gelieferte JSON-Objekt enthält die Beobachtungen wie in Abschnitt 2.2.1 Beobachtungen abrufen. Betrachten wir das Beispiel aus Kapitel 3.1. Das zur Beobachtung des fehlerhaften Endergebnisses aus Abb. 3.3 gehörende JSON-Objekt zeigt Abb. 4.5. Man sieht die 3 Objekte, die jeweils eine Tabellenzeile im Ergebnis ausmachen. Im Folgenden betrachten wir das erste Objekt (Zeilen 3 bis 17) zur Erklärung, die Übersetzung der beiden Anderen erfolgt analog. Das Objekt beinhaltet die drei Schlüsselwerte `context`, `observations` und `nested`. Hinter `observations` verbergen sich die tatsächlichen Beobachtungen. In diesem Fall lautet die `mid` - die Kennung der Markierung - `m1`. Die Unterabfrage, die sich hinter dieser Markierung verbirgt, liefert ein tabellarisches Ergebnis. Das lässt sich daran erkennen, dass der Wert für den Schlüssel `m1` ein weiteres Objekt ist. Die Schlüsselwerte dieses Objekts ent-

```
1  [
2    [
3      {
4        "context" :
5        {
6          "group_tid_c" : "(Biggs)"
7        },
8        "observations" :
9          {
10           "m1" :
11           {
12             "city" : ["Biggs"],
13             "belt" : [1150000]
14           }
15         },
16       "nested" : null
17     },
18     {
19       "context" :
20       {
21         "group_tid_c" : "(Chico)"
22       },
23       "observations" :
24       {
25         "m1" :
26           {
27             "city" : ["Chico"],
28             "belt" : [320000]
29           }
30       },
31       "nested" : null
32     },
33     {
34       "context" :
35       {
36         "group_tid_c" : "(Dixon)"
37       },
38       "observations" :
39       {
40         "m1" :
41         {
42           "city" : ["Dixon"],
43           "belt" : [50000]
44         }
45       },
46       "nested" : null
47     }
48   ]
49 ]
```

Abbildung 4.5: JSON-Objekt der Beobachtung aus Abb.3.3

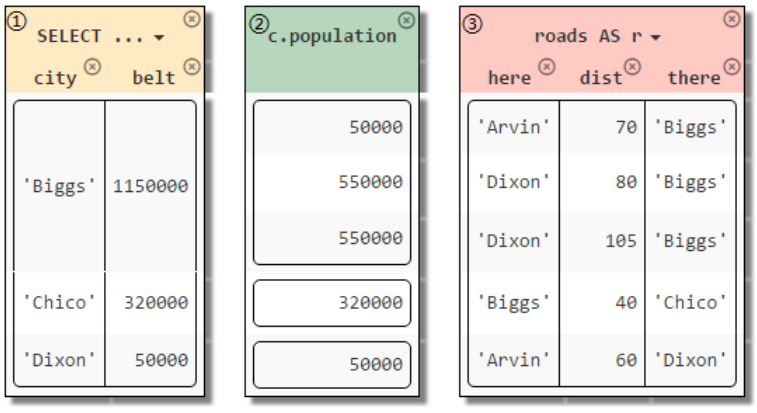
sprechen den Spaltennamen der tabellarischen Beobachtung. Als Werte tragen sie die eigentlichen Beobachtungen in einelementigen Arrays. In der Tabelle sieht man diese Zuweisung in der obersten Zeile.
Zur Erklärung der beiden anderen Schlüsselwerte `context` und `nested` betrachten wir das JSON-Objekt zu Abb. 3.5, entfernen aber den Filter. Einen Ausschnitt davon (a) sowie die resultierende Tabelle (b) sehen wir in Abb. 4.6. Eine Herausforderung für die Implementierung stellt die Zuweisung der Werte aus Beobachtung ② zur korrekten Zeile dar. Auch die Unterscheidung zwischen geschachtelten Beobachtungen und solchen, die zwar im `nested`-Teil des Objekts übergeben werden, aber nicht Teil der Gruppe sind, muss beachtet werden.
Im `nested`-Teil des Objekts sind weitere Beobachtungen gespeichert. Diese sind meist Beobachtungen, die einer Gruppe angehören. Bei gruppierten Ergebnissen sind das tatsächlich Werte der einzelnen Mitglieder einer Gruppe. Im Beispiel ist Markierung `m2` eine solche Beobachtung. Sie zeigt die Bevölkerungszahlen jeder Stadt, die einer der Gruppen angehört. So kommt es dazu, dass die Beobachtung für Biggs drei Ergebnistupel liefert. Das JSON-Objekt hält die Werte für diese Markierung in `nested` (siehe Zeilen 32, 53 und 74). Auch die Beobachtungen der Markierung `m3` sind in diesem Teil des Objekts enthalten (siehe Zeilen 26, 47 und 68). Wie Abb. 4.6b zu erkennen gibt, enthält diese Markierung die komplette Tabelle **roads**, deren Tupel nicht zur Gruppe gehören. Wie erkennt das Programm, ob ein Ergebnis geschachtelt darzustellen ist oder nicht?

```
[
  [
    {
      "context" :
      {
        "group_tid_c" : "(Biggs)"
      },
      "observations" :
      {
        "m1" :
        {
          "city" : ["Biggs"],
          "belt" : [1150000]
        }
      },
      "nested" :
      [
        {
          "context" :
          {
            "tid_c" : "(0,1)",
            "tid_r" : "(0,1)"
          },
          "observations" :
          {
            "m3" :
            {
              "here" : ["Arvin"],
              "dist" : [70],
              "there" : ["Biggs"]
            },
            "m2" :
            {
              "population" : [50000]
            }
          },
          "nested" : null
        },
        {
          "context" :
          {
            "tid_c" : "(0,4)",
            "tid_r" : "(0,4)"
          },
          "observations" :
          {
            "m3" :
            {
              "here" : ["Dixon"],
              "dist" : [80],
              "there" : ["Biggs"]
            },
            "m2" :
            {
              "population" : [550000]
            }
          },
          "nested" : null
        },
        {
          "context" :
          {
            "tid_c" : "(0,4)",
            "tid_r" : "(0,5)"
          },
          "observations" :
          {
            "m3" :
            {
              "here" : ["Dixon"],
              "dist" : [105],
              "there" : ["Biggs"]
            },
            "m2" :
            {
              "population" : [550000]
            }
          },
          "nested" : null
        }
      ]
    },

    ...

  ]
]
```

(a) Ausschnitt des JSON-Objekts der Beobachtung aus (b)



(b) übersetzte Tabelle aus dem JSON-Objekt von (a)

Abbildung 4.6: Beobachtung zur Demonstration von `getTable`

Jede Markierung hat einen Kontext (vgl. Abschnitt 2.2.1 Markierung anlegen). Mit dessen Hilfe erkennt das Programm, ob das gerade betrachtete Ergebnis geschachtelt ist oder nicht. Betrachten wir zuerst Markierung `m2`, welche in Abb. 4.7a zu sehen ist. Ihr Kontext besteht aus `group_tid_c, tid_c` und `tid_r`. Beim Vergleich mit dem Kontext der äußeren Beobachtung (Abb. 4.6a, Zeile 6) fällt auf, dass der Kontext der Markierung `m2` den kompletten äußeren Kontext `group_tid_c` enthält. Das zeigt, dass die Markierung `m2` eine Unterbeobachtung der Äußeren ist. Deshalb stellt das Programm diese Beobachtung geschachtelt dar. Der Kontext der Markierung `m3` (siehe Abb. 4.7b) enthält den äußeren Kontext nicht. Daran erkennt das Programm, dass die Beobachtung zu dieser Markierung nicht in geschachteltem Zusammenhang zur äußeren Beobachtung steht.

```
1  {
2    "active":true,
3    "mid":2,
4    "type":
5    [
6      {"population":"int4"}
7    ],
8    "context":
9    [
10     "group_tid_c",
11     "tid_c",
12     "tid_r"
13   ],
14   "to":40,
15   "from":28
16 }
```

(a) `m2`

```
1  {
2    "active":true,
3    "mid":3,
4    "type":
5    [
6      {"here":"varchar"},
7      {"dist":"int4"},
8      {"there":"varchar"}
9    ],
10   "context":
11   [
12     "tid_r"
13   ],
14   "to":78,
15   "from":68
16 }
```

(b) `m3`

Abbildung 4.7: Markierungen `m2` und `m3` zu Beobachtung aus Abb. 4.6b

Mit Hilfe des in `context` gespeicherten Kontexts der Beobachtung, findet auch die Zuweisung der Zeilen statt. Zusätzlich legt das Programm eine Struktur `tids` an, die für jede Zeile jeder Markierung den Kontext speichert. Sie wird zur Erkennung doppelter Tupel verwendet. Die Zeilenzuweisung über den Kontext ist wichtig für Objekte, deren `nested`-Wert nicht `null` ist. Eine rekursive Funktion führt die Übersetzung des `nested`-Teils der Beobachtung durch. Diese arbeitet so lange, bis der `nested`-Wert `null` ist. Beim Aufruf übergibt die Funktion `getTable` der rekursiven Funktion unter anderem den äußeren Kontext. Dieser erlaubt die Suche nach der richtigen Zeile, denn jede Zeile erhält den Kontext als Klasse. Gibt es mehrere Zeilen, die dieselbe Klasse besitzen, gibt die Position im Array der `nested`-Objekte Aufschluss über die Auswahl der Zeile.

Die Funktion `getTable` implementiert die Übersetzung des JSON-Objekts in eine darstellbare Tabelle, die das Ergebnis lesbar macht. Während der Iteration im Objekt der Beobachtung, legt sie zuerst einen Tabellenkopf für jede aktive Markierung an, die in der aktuellen Tabelle enhalten ist. Dieser erhält zur besseren Übersicht die Hintergrundfarbe der entsprechenden Markierung. Jeder Tabellenkopf enthält das Schema der Beobachtung. Beim Anlegen der Tabellenköpfe achtet die Funktion darauf, dies in der Reihenfolge auszuführen, die bisher herrscht. Für jede Beobachtung arbeitet die Funktion die drei Schlüsselwerte `context, observations` und `nested` ab. Den Kontext fügt es den entsprechenden Zeilen als Klasse hinzu. Außerdem speichert es ihn in diesem Schritt im Objekt `tids` ab, um später durch den Kontext auf die korrekte Zeile zuzugreifen. Die Tiefe des `nested`-Objekts gibt Aufschluss über die Anzahl der Zeilen, die die entsprechende Beobachtung überspannt. Die Werte, die das Feld `observations` enthält, stellt das Programm in der der Markierung entsprechenden Spalte der ermittelten Zeile dar. Enthält eine Spalte eine tabellarische Beobachtung, erhält sie eine spezielle Klasse. Diese ist für jede tabellarische Beobachtung einzigartig. Zur Darstellung der Werte, die im `nested`-Objekt enthalten sind, ruft `getTable` eine rekursive Unterfunktion auf. Neben der Berechnung der korrekten Zeile bearbeitet die Funktion die drei Felder genauso wie `getTable`. Zusätzlich ermittelt sie nach dem oben erklärten Verfahren, ob es sich bei einem Wert um ein geschachteltes Ergebnis handelt. Ist das der Fall, erhält jede Gruppe der Beobachtung eine separate Klasse, die für die Darstellung tabellarischer Werte verwendet wird. Für weitere `nested`-Objekte innerhalb eines Anderen, ruft sie sich rekursiv auf, bis sie am tiefsten Punkt der Struktur angekommen ist.

4.5.1 Gestaltung der Tabellen

Die Funktion `getTable` liefert die Beobachtungen in Tabellenform. Erst danach erhält die Tabelle ihre Gestaltung. Dabei erhält sie Rahmen für die inneren Tabellen, welche durch die speziellen Klassen zur Darstellung tabellarischer Werte ermittelt werden. Außerdem blendet das Programm erst in diesem Schritt unerwünschte Spalten aus und fügt Dropdown-Menüs zu jeder Beobachtung hinzu, die aus mehr als einer Spalte besteht. In einem weiteren Schritt fügt das Programm Spalten zur Separierung der Beobachtungen hinzu. Weitere Gestaltungsmaßnahmen, wie das Hervorheben von Zeilen, nach denen diese Beobachtung im letzten Schritt gefiltert wurde und das Hervorheben von Werten, die beim letzten „What If“-Filter verändert wurden und die Gestaltung verschiedener Datentypen, werden hier realisiert.
Durch die Anzeige der Tabelle auf mehreren Seiten, beschränkt sich die Gestaltung auf wenige Tupel, die auf der aktuellen Seite dargestellt werden. Die Beschränkung der Anzeige auf einen bestimmten Teil der Tupel bringt auch

einige Hindernisse mit sich. Überspannt eine Beobachtung mehr als die angezeigten Zeilen, entsteht ein Problem beim Wechseln der Seite. Auf der nachfolgenden Seite ist an dieser Position dann keine Tabellenzelle enthalten. Um das zu verhindern, stellt das Programm beim Wechseln einer Seite sicher, dass die überspannende Beobachtung an die Position der ersten Tabellenzeile kopiert wird. So erkennt der Nutzer auch wieder den Bezug der Beobachtungen zueinander. Ein weiteres Problem entsteht beim Verwenden der von DataTables implementierten Suche nach Tupeln. Suchen wir in der Tabelle aus Abb. 3.18 nach dem String „Bob“, erhalten wir nur die dritte Zeile der Tabelle als Ergebnis. Das Problem hierbei liegt darin, dass die Werte der Beobachtung ① in der ersten Zeile, die sie überspannen, enthalten sind. Bei der uns angezeigten Zeile handelt es sich um die Zweite dieser Zeilen. Sie enthält keine Werte für Beobachtung ①, sondern nur eine unsichtbare Tabellenzelle. Um auch hier das korrekte Ergebnis anzuzeigen, kopiert das Programm in so einem Fall die Werte der ersten Zeile der Beobachtung ① und fügt sie in die zweite Zeile ein, sodass diese korrekt dargestellt wird. Beim Entfernen der Suche wird diese Kopie wieder gelöscht.

4.6 Filtern der Beobachtungen

Wie wir in Kapitel 3 gesehen haben, ist es nützlich, Ergebnisse nach bestimmten Kriterien zu filtern. Vor allem bei großen Ausgangstabellen erleichtern gefilterte Ergebnisse die Konzentration auf die wichtigsten Tupel. Auch das Ersetzen eines Wertes zur vorläufigen Überprüfung, ob das DBMS dann das erwartete Ergebnis liefert, kann das Debugging erleichtern. So kann überprüft werden, ob der Fehler wirklich dort liegt, wo er vermutet wird, ohne die Query dafür zu verändern. Dieser Abschnitt zeigt die Implementierung der Filter.

4.6.1 Filter

Um auch für das Filtern des Ergebnisses ein Minimum an Nutzeraktion zu verlangen, kann der Nutzer einen Filter für die interessante Zeile direkt durch einen Doppelklick darauf anwenden. Für das Anlegen mehrerer Filter ist es notwendig, beim Auswählen der Zeilen die `shift`-Taste gedrückt zu halten. Erst beim Doppelklick auf eine der Zeilen, sendet das Programm den Filter an den Server und erhält das gefilterte Ergebnis.
Wie oben erwähnt, verwenden wir die Variable `tids` zur Speicherung der Kontexte. Sie enthält ein Objekt für jeden markierten Teilausdruck. Dieses enthält für jede Zeile der Beobachtung den Kontext dieser Zeile. Dadurch können wir doppelte Tupel identifizieren und durch gleiches Verhalten sowie eine besondere Darstellung erkennbar machen. Doppelte Tupel kommen u. a. dann vor, wenn die Oberfläche den Join zweier Tabellen anzeigt. Abb. 4.12 zeigt in Zei-

le 2 der Beobachtung ① ein Duplikat des Tupels aus Zeile 1. Zum besseren Verständnis des `tid`-Objekts betrachten wir das Beispiel aus Abschnitt 3.1. Die Abb. 4.8 zeigt das `tid`-Objekt der Beobachtung aus Abb. 3.5. Die Nummerierung der Tabellenzeilen beginnt bei 2, d. h. die erste Zeile hat den Index 2. Das liegt daran, dass das Objekt nur die Zeilen des Tabellenkörpers aufnimmt. Zeile 0 und 1 jeder Tabelle enthalten die Tabellenköpfe, die für das Objekt uninteressant sind. Wir beobachten die drei Teilausdrücke `m1, m2` und `m3`. Für jede dieser Beobachtungen existieren drei Tabellenzeilen. Jede dieser Zeilen besitzt einen Kontext, der sie identifiziert. Besitzen zwei Zeilen denselben Kontext, enthalten sie dieselbe Beobachtung. Einen Spezialfall sehen wir bei `m1`. Alle drei Zeilen dieser Beobachtung besitzen denselben Kontext. Die zugehörige Tabellenspalte enthält nur eine einzige Zeile. Für eine einheitliche Darstellung erhält eine Beobachtung für jede Zeile, die sie überspannt, einen Eintrag in `tids`. Sie enthalten alle denselben Wert, da es sich tatsächlich um die identische Beobachtung handelt. Die anderen beiden Beobachtungen enthalten für jede Zeile den Kontext der dort enthaltenen Beobachtung. Da sich diese Beobachtungen alle voneinander unterscheiden, stimmt dort kein Kontext mit einem Anderen überein.

```
{
    "m1":
    {
        "2":
        {
            "group_tid_c":"(Biggs)"
        },
        "3":
        {
            "group_tid_c":"(Biggs)"
        },
        "4":
        {
            "group_tid_c":"(Biggs)"
        }
    },
    "m2":
    {
        "2":
        {
            "group_tid_c":"(Biggs)",
            "tid_c":"(0,1)",
            "tid_r":"(0,1)"
        },
        "3":
        {
            "group_tid_c":"(Biggs)",
            "tid_c":"(0,4)",
            "tid_r":"(0,4)"
        },
        "4":
        {
            "group_tid_c":"(Biggs)",
            "tid_c":"(0,4)",
            "tid_r":"(0,5)"
        }
    },
    "m3":
    {
        "2":
        {
            "tid_r":"(0,1)"
        },
        "3":
        {
            "tid_r":"(0,4)"
        },
        "4":
        {
            "tid_r":"(0,5)"
        }
    }
}
```

Abbildung 4.8: `tids` der Beobachtung aus Abb. 3.5

Beim Anwählen einer Zeile fügt das Programm den Kontext der Beobachtung dem Filter hinzu. Im Beispiel von oben ist das `"group_tid_c":"(Biggs)"`. Abb. 4.9a zeigt den angewandten Filter. Er enthält den Schlüsselwert `"group_tid_c"`. Da wir nur nach einem Wert des Ergebnisses von `m1` filtern, erwarten wir keine anderen Schlüsselwerte, denn deren Kontext enthält keine weiteren Schlüsselwerte. Der Schlüssel verweist auf einen einelementigen Array, der den Wert `"(Biggs)"` enthält. Dank diesem Filter erhalten wir lediglich die Zeilen der Beobachtung vom Server zurück, deren Kontext mit dem des Filters übereinstimmen. In unserem Beispiel sind das all diejenigen, die der Gruppe von Biggs angehören. Besteht der Kontext einer Beobachtung aus mehreren Teilstücken, enthält der Filter all diese als Schlüsselwerte.

```
{
  "filter":
  {
    "group_tid_c":["(Biggs)"]
  }
}
```

(a) Filter der Beobachtung aus Abb. 3.5

```
{
  "filter":
  {
    "tid_j":
    [
      "(0,1)",
      "(0,5)"
    ]
  }
}
```

(b) Filter der Beobachtung aus Abb. 3.18

```
{
  "filter":
  {
    "tid_j":
    [
      "(0,1)",
      "(0,5)"
    ],
    "tid_p":
    [
      "(0,3)"
    ]
  }
}
```

(c) Filter angewandt aus 2 verschiedenen Beobachtungen

Abbildung 4.9: Beispiele für Filterobjekte

Filtert der Nutzer nach mehreren Tupeln derselben Beobachtung, enthält der Array mehrere Werte. In Abb. 4.9b sehen wir den Filter, unter dessen Anwendung das Ergebnis aus Abb. 3.18 entsteht. In diesem Beispiel aus Abschnitt 3.4 filtern wir nach den beiden Tupeln `Carpenter` und `Baker` in der Tabelle **jobs**. Diese Tupel besitzen den Kontext `"tid_j":"(0,1)"` bzw. `"tid_j":"(0,5)"`. Im Filter sehen wir den zweielementigen Array, der den Kontext der beiden Tupel enthält.

In manchen Situationen ist es nützlich, zuerst den Filter auf der einen Markierung und später zusätzlich auf einer anderen Markierung anzuwenden, ohne dabei den Ersten zu verwerfen. Auch das ist möglich. Wenden wir im eben genannten Beispiel zusätzlich einen Filter an, der die Tabelle **people** auf diejenigen Tupel einschränkt, die Informationen zur Person Nadja Smith liefert, resultiert der Filter aus Abb. 4.9c. Er enthält weiterhin die Einschränkung von oben auf der Tabelle **jobs** und zusätzlich die Einschränkung auf Tabelle **people**.

Es gibt auch Situationen, in denen es nützlich ist, den Filter zu entfernen, um einen größeren Teil des Ergebnisses zu sehen. Das Programm reagiert auf onclick-Events in der untersten Zeile der Tabelle einer gefilterten Beobachtung (siehe Ⓐ in Abb. 3.6a), indem es den Filter der entsprechenden Beobachtung löscht und die Beobachtung erneut vom Server empfängt. Das Programm erkennt anhand der Position, an der das Event ausgelöst wurde, die zugehörige Markierung. Es überprüft jeden Schlüsselwert des Filters, ob er mit einem Wert des Kontexts der ermittelten Markierung übereinstimmt. Ist das der Fall, so schränkt der Filter die ermittelte Beobachtung ein. Arrays, die sich hinter einem solchen Schlüsselwert befinden, leert das Programm, um die Filter, die die ermittelte Markierung betreffen, zu entfernen.
Beim Löschen einer Markierung prüft das Programm, ob ein Kontext im Filter existiert, den keine der aktiven Markierungen enthält. Ist das der Fall, leert es den zu diesem Schlüsselwert passenden Array. Somit stellt es sicher, dass der Filter beim Anlegen neuer Markierungen nicht für Verwirrung sorgt. Wie schon in Abschnitt 4.4.2 besprochen, existiert ein Button, der die Oberfläche in die Ausgangssituation zurücksetzt. Dabei leert das Programm auch den Filter, sodass dieser für spätere Markierungen nicht hinderlich ist.

4.6.2 What If

Das Verändern von Beobachtungen an bestimmten Stellen kann sehr hilfreich für das Debugging sein. Im Abschnitt 3.1 verwenden wir diese Funktion zum Test, ob wir den Fehler an der richtigen Stelle vermuten. Der Server erwartet für die Ausführung von „What If“-Anfragen ein Array von Objekten, welche die in Kapitel 2.2.1 - Gefilterte Beobachtungen anzeigen - beschriebene Struktur besitzen. Das Objekt, das zum Ergebnis aus Abb. 3.6b führt, zeigt Abb. 4.10a. Jede Tabellenzelle besitzt die Klasse ihrer `mid`. Dadurch lässt sich diese leicht finden. Auch der Name der Spalte ist in einer der Klassen enthalten. Den Kontext finden wir in der Variable `tids` unter `m2` in der entsprechenden Zeile. Im Beispiel ist das Zeile 4. Da wir den Wert 550.000 durch 0 ersetzen, beträgt `"substitute"` den Wert 0.

Ändert der Nutzer einen Wert in der Tabelle, startet er durch Drücken der `Enter`-Taste die Berechnung des Objekts. Das Programm ermittelt die einzelnen Felder des Objekts wie beschrieben. Möglicherweise ändert der Nutzer denselben Wert mehrfach. Um auch für diesen Fall die richtige Ersetzung durchzuführen und um den Filter möglichst klein zu halten, fügt das Programm das berechnete Objekt nur ein, falls es kein Objekt gibt, das dieselbe Tabellenzelle anspricht. Ansonsten ersetzt es im bereits im Filter enthaltenen Objekt das Substitut. Zwei Objekte sprechen nur dann dieselbe Zelle an, wenn die `mid`, die Spalte und der Kontext beider Objekte, übereinstimmen.
Um die Nutzerfreundlichkeit hoch zu halten, hebt das Programm alle Werte hervor, die sich durch die Ersetzung ändern. Gruppen, die sich verkleinern oder vergrößern, erhalten einen blauen Rahmen, um sich abzuheben. Damit dies erreicht werden kann, vergleicht das Programm die neue Beobachtung mit derjenigen vor der Anwendung des Ersetzungsfilters.

```
{
  "what-if":
  [
    {
      "mid":"m2",
      "column":"population",
      "context":
      {
        "group_tid_c":"(Biggs)",
        "tid_c":"(0,4)",
        "tid_r":"(0,5)"
      },
      "substitute":"0"
    }
  ]
}
```

(a) „What If“-Filter der Beobachtung aus Abb. 3.6b

```
{
  "filter":
  {
    "group_tid_c":["(Biggs)"]
  },
  "what-if":
  [
    {
      "mid":"m2",
      "column":"population",
      "context":
      {
        "group_tid_c":"(Biggs)",
        "tid_c":"(0,4)",
        "tid_r":"(0,5)"
      },
      "substitute":"0"
    }
  ]
}
```

(b) Filter der Beobachtung aus Abb. 3.6a

Abbildung 4.10: Filter zu Abb. 3.6a

Der Nutzer kann gleichzeitig „What If“-Debugging anwenden und das Ergebnis filtern. In Abb. 3.6a sehen wir das Ergebnis einer solchen Anfrage. Dort wenden wir den Filter und die Ersetzung von oben an. Das JSON-Objekt, das der Server erhält, enthält den Filter aus Abb. 4.9a und das Objekt aus Abb. 4.10a. In Abb. 4.10b sehen wir das komplette Objekt. Als Antwort auf die Anfrage erhalten wir die gewünschten Tupel der Beobachtung, die gleichzeitig mit dem ersetzten Wert berechnet sind.

Führt die Ersetzung eines Werts nicht zum gewünschten Ergebnis, kann ihn der Nutzer auf den ursprünglichen Wert zurücksetzen. Dazu muss er lediglich das Symbol zum Entfernen der Ersetzung drücken, welches in Abb. 3.6b mit Ⓑ markiert ist. Das Programm ermittelt dann die Werte für die `mid`, die Spalte und den Kontext für die Tabellenzelle, in der der Nutzer das Event ausgelöst hat. Durch den Vergleich dieser Werte mit denen aus dem „What If"-Teil des Filters, findet es das gesuchte Objekt und entfernt dieses aus dem Filter. Danach ruft er vom Server erneut die Beobachtungen ab, aber mit dem angepassten Filter.

4.7 Manipulation der Tabellen

Beim Arbeiten mit den tabellarischen Beobachtungen der markierten Teilausdrücke ist es oft nützlich, nur das anzuzeigen, was den Nutzer interessiert. Neben der eben besprochenen Filterung, die uninteressante Tabellenzeilen entfernt, bietet die Benutzeroberfläche die Möglichkeit, überflüssige Spalten einer Beobachtung zu verbergen. Außerdem ist es oft leichter, Beobachtungen in Bezug zu stellen, wenn sie direkt nebeneinander platziert sind. Um dem Nutzer freizustellen, in welcher Reihenfolge er die Beobachtungen anzeigen lässt, hat er die Möglichkeit, diese zu verschieben. Im Folgenden wird die Implementierung dieser Manipulationen der Tabellen erklärt.

4.7.1 Spalten aus- und einblenden

Realistische Datenbanktabellen enthalten nicht nur oft viele Tupel, sondern auch viele Spalten. Zeigt die Benutzeroberfläche für jede Beobachtung alle Spalten an, kann daraus oft ein unübersichtliches Ergebnis werden. Aus diesem Grund zeigt sie nur einen Teil der Spalten an. Bisher erachtet das Programm die jeweils erste und letzte Spalte jeder Beobachtung als diejenigen, die dem Benutzer angezeigt werden. Zusätzlich existiert eine weitere Möglichkeit zur Auswahl der angezeigten Tupel bei der ersten Anzeige einer Beobachtung. Dazu nehmen wir an, das vom Server gelieferte Objekt zur Spezifikation einer Markierung enthält ein weiteres Feld `columns`, welches einen Array mit den Namen der Spalten enthält, die interessanter sind als die Anderen. Solch ein Feld ist bisher noch nicht im Objekt enthalten, wird aber möglicherweise bald hinzugefügt.
Zusätzlich kann der Nutzer Spalten ein- oder ausblenden. Das erfolgt über das Dropdown-Menü im Tabellenkopf oder über die ⊗-Buttons im Schema der Beobachtung. Um genau die Spalten, die der Nutzer sehen will, anzuzeigen, auch nachdem neue Beobachtungen hinzukommen, speichert das Programm im Objekt `active` für jede Markierung die eingeblendeten Spalten. Bei der neuen Darstellung der Beobachtungen zeigt die Oberfläche nur die Spalten

an, die `active` enthält. Blendet der Benutzer eine Spalte aus, verschwindet deren Name aus dem Objekt und wird so nicht mehr angezeigt. Erst wenn er sie wieder einblendet, fügt das Programm den Spaltennamen wieder hinzu und zeigt sie dann weiterhin an.

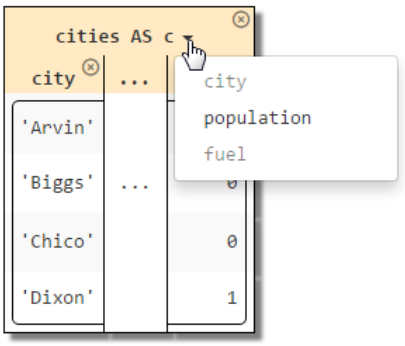


Abbildung 4.11: Beispiel zur Demonstration des Dropdowns

Beim Auswählen einer Spalte im Dropdown-Menü prüft das Programm, ob sie bisher ein- oder ausgeblendet ist. Wählen wir in Abb. 4.11 Spalte `population` aus, blendet das Programm diese Spalte ein. Das geschieht über die Funktion `fadeInColumn`. Durch das Entfernen der Klasse `hidden` und die jQuery Funktion `fadeIn`, blendet sie die ausgewählte Spalte ein. Zusätzlich testet sie, ob alle Spalten der Beobachtung sichtbar sind und entfernt ggf. den Platzhalter, der fehlende Spalten anzeigt. Außerdem passt die Funktion das Objekt `active` an, indem es die gerade eingeblendete Spalte dem Objekt an der entsprechenden Stelle hinzufügt.
Wählen wir im Beispiel Spalte `city` aus, blendet das Programm sie aus. Dasselbe geschieht beim Drücken des ⊗-Buttons in der Spalte. In beiden Fällen ruft das Programm die Funktion `fadeOutColumn` auf. Sie blendet die entsprechende Spalte aus, indem sie jeder ihrer Zellen die Klasse `hidden` zuweist, welche jedes Element unsichtbar macht. Durch die jQuery Funktion `fadeOut` animiert das Programm das Ausblenden. Ist die ausgewählte Spalte die einzige Ausgeblendete dieser Beobachtung, fügt die Funktion eine Spalte ein, die anzeigt, dass es ausgeblendete Spalten gibt. Aus dem Objekt `active` löscht die Funktion die eben bearbeitete Spalte.

4.7.2 Verschieben der Spalten

Neben dem Ausblenden von Spalten, hilft auch das Verschieben der Beobachtungen der Übersichtlichkeit. Markiert der Nutzer sehr viele Unterausdrücke, erhält er eine sehr breite Ergebnistabelle. Ist sie breiter als das Browserfenster, erschwert das die Erkennung des Zusammenhangs der letzten Beobachtung zur Ersten. Aus diesem Grund erlaubt die Oberfläche dem Nutzer, die Reihenfolge der Beobachtungen zu verändern.
Das Plugin dragtable ermöglicht dem Nutzer einer Anwendung, Tabellenspalten nach seinen Wünschen zu verschieben. Es sorgt durch Animation der ausgewählten Spalte für ein direktes Feedback an den Nutzer. Er erkennt sofort, ob er die gewünschte Spalte verschiebt und an welcher Stelle sie sich einfügt, wenn er sie loslässt. Für einfache Tabellen funktioniert das Plugin ohne Probleme. In den Tabellen, die die Ergebnisse der markierten Ausdrücke darstellen, existieren Zellen, die mehrere Zeilen bzw. Spalten überspannen. Das Plugin liefert nicht das erhoffte Ergebnis beim Verschieben einer Beobachtung, die mehr als eine Spalte enthält. In diesem Fall verschiebt es nur die erste Spalte, was zu einer verfälschten Tabelle führt. Zur Bewältigung des Verschiebens von Beobachtungen, die mehr als eine Spalte umfassen, benötigt das Plugin eine Überarbeitung. Das Ergebnis der Überarbeitung wird im Folgenden beschrieben.

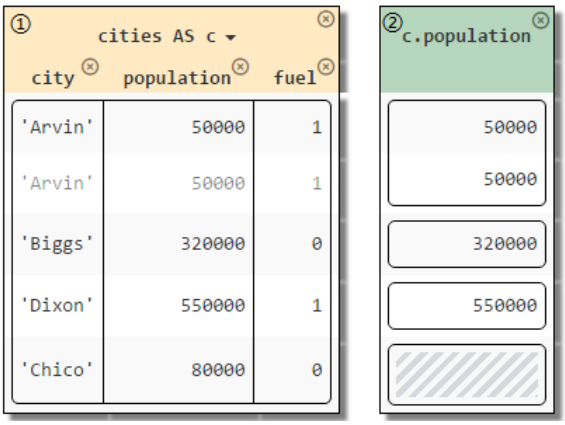


Abbildung 4.12: Beispiel zur Demonstration von dragTable

Wir ermöglichen lediglich das Verschieben ganzer Beobachtungen. Einzelne Spalten einer Beobachtung können nicht verschoben werden. Das Plugin implementiert die drei Phasen `dragStart`, `dragMove` und `dragEnd`. Im Folgenden betrachten wir das Beispielszenario aus Abb. 4.12 und nehmen an, das Ziel ist es, Beobachtung ① an das rechte Ende der Tabelle zu verschieben. Beim Greifen einer Spalte führt das Programm den Block `dragStart` aus. Das entspricht dem `mousedown`-Handler. In diesem Block erstellt das Programm

eine Kopie der ausgewählten Spalte. Hier liegt der erste Teil der Modifizierung. Ursprünglich erstellt das Plugin bei einer Spalte, deren `colspan`-Attribut größer ist als 1, trotzdem nur eine Kopie der ersten darunterliegenden Spalte. Um für diesen Fall eine Kopie anzufertigen, die alle Spalten der Markierung enthält, berücksichtigt das Plugin nach der Anpassung das `colspan`-Attribut der Zelle. Dadurch kopiert es alle Spalten der Markierung.
Sobald der Nutzer die Maus bewegt, löst er das `mousemove`-Event aus. Darauf reagiert die Funktion `dragMove`. Dieser Block sorgt für die Bewegung der kopierten Spalten, welche der Mausbewegung folgt. Dieser Teil des Plugins verlangt keinerlei Anpassung für unseren Zweck.
Die wichtigste Aufgabe führt der Programmblock `dragEnd` aus. Er reagiert auf ein `mouseup`-Event, welches der Nutzer beim Loslassen des Objekts auslöst. Hat der Nutzer die Tabellenspalte(n) weit genug bewegt und befindet sich die Maus zum Zeitpunkt des Loslassens über der Tabelle, veranlasst das Plugin das Verschieben der ausgewählten Spalte(n) an die richtige Stelle in der Tabelle. Die Funktion `moveColumns` führt das aus. Ihre Grundlage bildet die Funktion `moveColumn` des Plugins, welche das Verschieben einer einzelnen Spalte implementiert. Für das Verschieben mehrerer Spalten ermittelt die Funktion zusätzlich zum Index der Spalte in der `header`-Zeile (`fIdx`) den Index in der `schema`-Zeile (`fIdxInner`). Diese Beiden können sich unterscheiden, da jede der Beobachtungen mehr als eine Spalte überspannen kann. Für die Beobachtung ② im Beispiel beträgt der `fIdx` = 1 und der `fIdxInner` = 3. Für jede der in der Beobachtung enthaltenen Spalten verschiebt das Plugin Zeile für Zeile die Zellen vom Start- zum Endindex. Das Verschieben entspricht der Implementierung der ursprünglichen Funktion. Zusätzlich ist eine Anpassung der Indizes nötig. Beim Verschieben von rechts nach links vergrößert sich beispielsweise der Index der zu verschiebenden Spalte nach jeder Spalte um eins.
Nach der Verschiebung sorgt das Plugin für die korrekte Darstellung durch das Einfügen der Trennspalten am korrekten Ort. Außerdem speichert es die Reihenfolge der Beobachtungen, um die korrekte Reihenfolge beim erneuten Abruf der Beobachtungen zu erhalten.

Durch die zusätzlichen Spalten, die die Beobachtungen optisch trennen sowie die zusätzliche Spalte, die als Ersatz für ausgeblendete Spalten angezeigt wird, in Verbindung mit DataTables, gibt es weitere Probleme bei der Ermittlung der korrekten Position. Das Hinzufügen einer Spalte zum DataTables-Objekt ist mit Aufwand verbunden, den wir uns ersparen. Deshalb fügen wir die Trennspalte zwischen den Beobachtungen wie auch die `expand`-Spalte nur im sichtbaren Bereich der Tabelle ein. Da wir bei mehrseitigen Tabellen die Verschiebung auf allen und nicht nur auf der aktuellen Seite durchführen, arbeiten wir auf dem DataTables-Objekt der Tabelle. Die Ermittlung der Start- wie auch der Endspalte arbeitet nicht auf dem DataTables-Objekt, sondern

auf der DOM-Darstellung der Tabelle. Diese enthält sowohl Trennspalten als auch `expand`-Spalten. Um im DataTables-Objekt die korrekten Spalten zu verschieben, müssen wir die Indizes anpassen. Dafür laufen wir durch die Spalten der `header`-Zeile und der `schema`-Zeile und dekrementieren die aus der DOM-Darstellung ermittelten Indizes bei jeder Trenn- bzw. `expand`-Spalte, die vor dem jeweiligen Index liegt.

Das Plugin entspricht nach der Anpassung genau den Anforderungen der Benutzeroberfläche. Für andere Zwecke ist es nicht ohne Veränderungen einsetzbar, da es auf der speziellen Klassenverteilung aufbaut und Funktionen verwendet, die in `HabitatUI.js` definiert sind.

Kapitel 5

Fazit

Die Benutzeroberfläche für Habitat stellt alle Funktionen für das Debuggen bereit. Vom Anlegen der Markierungen über die Anzeige bis zum Filtern der Beobachtungen und Ersetzen einzelner Werte, können alle vom Backend bereitgestellten Funktionen über die Oberfläche verwendet werden. Sie unterstützt den interaktiven Debuggingprozess, indem ein Nutzer nach und nach verschiedene Markierungen anlegen und die Auswahl anhand der bisher vom Server erhaltenen Ergebisse treffen kann. Außerdem ermöglicht die Verteilung großer Ergebnisse auf mehrere Seiten eine übersichtliche Darstellung der Beobachtungen. Durch die Möglichkeit des Filterns und der Ausblendung überflüssiger Spalten, behält die Benutzeroberfläche den Fokus auf den interessanten Bereichen der Beobachtung. Auch das freie Verschieben von Spalten erleichtert das Erkennen von Zusammenhängen. Die Oberfläche reagiert direkt auf Nutzeraktionen. Dadurch begrenzt sich die Nutzeraktion auf ein Minimum. Die Bedienung ist möglichst einfach gehalten, sodass der Nutzer seine volle Konzentration auf das Debugging lenken kann.

Die Oberfläche ist in diesem Zustand nicht für das Arbeiten mit Teilabfragen ausgelegt, deren Beobachtungen die Größenordnung von mehreren 100 Tupeln übersteigen. Die Verzögerung findet hauptsächlich bei der Verarbeitung des JSON-Objekts der Beobachtung statt. Der Grund dafür ist die große Menge der Zugriffe auf das DOM. Zur Verbesserung der Performanz dieses Prozesses, kann die Funktion `getTable` insoweit angepasst werden, dass sie nur diejenigen Tupel bearbeitet, die auf der aktuell angezeigten Seite zu sehen sind. Dasselbe Prinzip wird schon beim Gestalten der erarbeiteten Tabelle angewendet. Nur die sichtbaren Tupel der Tabelle werden dabei berücksichtigt, alle Anderen werden erst dann gestaltet, wenn sie sichtbar werden. So spart sich das Programm unnötige Zugriffe auf Tupel, die nicht zu sehen sind. Besser ist eine Änderung der Schnittstelle, sodass der Server nur den Teil der Tupel liefert, welche auf der aktuellen Seite im Frontend benötigt werden.

Beim Löschen von Beobachtungen kann es zu fehlerhaften Darstellungen der restlichen Beobachtungen kommen. Bei der Implementierung des Löschens wird nur die Markierung aus der Darstellung entfernt. Die Funktion achtet nicht darauf, ob die restlichen Funktionen in Bezug zueinander stehen. Zur Vermeidung solcher fehlerhaften Darstellungen, kann in zukünftiger Arbeit beim Löschen die komplette Beobachtung neu abgerufen werden. Möglich ist auch, eine solche Abfrage nur dann durchzuführen, wenn die übrigen Beobachtungen nicht in Bezug zueinander stehen. Dadurch kann der Datenverkehr zwischen Client und Server reduziert werden.

Während der Entwicklung wurde nicht auf die Kompatibilität mit dem Internet Explorer geachtet. Funktionen wie z. B. das Hochladen von Dateien werden erst vom Internet Explorer 11 oder später unterstützt. Einige der CSS Regeln werden von den Browsern unterschiedlich interpretiert. Die Benutzeroberfläche wurde in Chrome (Version 55) und Firefox (Version 51) getestet und auf deren Interpretationen angepasst. Für die zukünftige Arbeit an der Oberfläche kann die Kompatibilität mit anderen Browsern verbessert werden, um einheitliches Auftreten und Funktionalität zu gewährleisten. Außerdem wäre es sinnvoll, die Barrierefreiheit zu verbessern. Bei der bisherigen Entwicklung wurde dieser Aspekt vernachlässigt.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Ace] *Ace - The High Performance Code Editor for the Web.* https://ace.c9.io/.

[B+11] Bos, Bert u. a.: Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification / W3C. Version: 2011. https://www.w3.org/TR/CSS2/css2.pdf(Abruf:23.01.2017). 2011. – W3C Recommendation.

[BK08] Bibeault, Bear ; Katz, Yehuda: *JQuery in Action.* 3. Aufl. Birmingham : Manning, 2008.

[Cod] *CodeMirror.* https://codemirror.net/.

[DG15] Dietrich, Benjamin ; Grust, Torsten: A SQL Debugger Built from Spare Parts: Turning a SQL: 1999 Database System into Its Own Debugger. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* New York, USA : ACM, 2015.

[Die] Dietrich, Benjamin: *Readme.* git@dbworld.informatik.uni-tuebingen.de:Habitat. – Datei README.md im Git Repository Habitat, Stand: 17.01.2017.

[GR13] Grust, Torsten ; Rittinger, Jan: Observing SQL Queries in Their Natural Habitat. In: *ACM Trans. Database Syst.* 38 (2013), Nr. 1. http://db.inf.uni-tuebingen.de/staticfiles/publications/habitat.pdf(Abruf:08.01.2017).

[H+14] Hickson, Ian u. a.: HTML5 / W3C. Version: 2014. https://www.w3.org/TR/2014/REC-html5-20141028/single-page.html(Abruf:13.01.2017). 2014. – W3C Recommendation.

[jQu] jQuery Foundation: *jQuery API Documentation: jQuery.when().* https://api.jquery.com/jquery.when/(Abruf:03.02.2017).

[Koc07] Koch, Stefan: *JavaScript - Einführung, Programmierung und Referenz - inklusive Ajax.* 4. vollst. überarb. Aufl. Dpunkt-Verlag, 2007.

[Lab06] Laborenz, Kai: *CSS-Praxis*. 4. Aufl. Bonn : Galileo Press, 2006.

[Lim] Lima, Markus: *Bootstrap FileStyle: Customization of input html file for Bootstrap Twitter*. http://markusslima.github.io/bootstrap-filestyle/.

[OTa] Otto, Mark ; Thornton, Jacob: *Components*. http://getbootstrap.com/components/.

[OTb] Otto, Mark ; Thornton, Jacob: *CSS*. http://getbootstrap.com/css/.

[OTc] Otto, Mark ; Thornton, Jacob: *JavaScript*. http://getbootstrap.com/javascript/.

[Ott12] Otto, Mark: Building Twitter Bootstrap. In: *A List Apart* (2012). http://alistapart.com/article/building-twitter-bootstrap(Abruf:11.02.2017).

[Pos17a] PostgreSQL Global Development Group: *About*. https://www.postgresql.org/about/(Abruf:05.02.2017). Version: 1996-2017.

[Pos17b] PostgreSQL Global Development Group: *PostgreSQL Documentation, Kapitel 7.8 WITH Queries (Common Table Expressions)*. https://www.postgresql.org/docs/9.6/static/queries-with.html(Abruf:13.02.2017). Version: 1996-2017.

[Pos17c] PostgreSQL Global Development Group: *PostgreSQL Documentation, Kapitel 9.2 Comparison Functions and Operators*. https://www.postgresql.org/docs/9.6/static/functions-comparison.html(Abruf:05.02.2017). Version: 1996-2017.

[Pos17d] PostgreSQL Global Development Group: *PostgreSQL Documentation, Kapitel 9.20 Aggregate Functions*. https://www.postgresql.org/docs/9.6/static/functions-aggregate.html(Abruf:05.02.2017). Version: 1996-2017.

[Pos17e] PostgreSQL Global Development Group: *PostgreSQL Documentation, Kapitel 9.22.3 NOT IN*. https://www.postgresql.org/docs/9.6/static/functions-subquery.html(Abruf:05.02.2017). Version: 1996-2017.

[Pos17f] PostgreSQL Global Development Group: *PostgreSQL Documentation, SELECT*. https://www.postgresql.org/docs/9.6/static/sql-select.html(Abruf:05.02.2017). Version: 1996-2017.

[Shi03] Shiflett, Chris: *HTTP Developer's Handbook*. 01. Aufl. Indianapolis, Indiana : Sams Publishing, 2003 http://proquest.tech.safaribooksonline.de/book/web-development/http/0672324547/http-requests/26?uicode=Tuebingen(Abruf:08.01.2017).

[Spr17] SpryMedia Ltd: *DataTables: Table plug-in for jQuery*. https://datatables.net/. Version: 2007-2017.

[Sta] Stack contributors: *The Haskell Tool Stack*. https://docs.haskellstack.org/.

[Van08] Vanderkam, Dan: *dragtable: Visually reorder all your table columns*. http://www.danvk.org/wp/dragtable/. Version: 2008.