Eberhard Karls Universität Tübingen

Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl für Datenbanksysteme

Bachelor Thesis Media Informatics

Binoculars for Habitat

Implementation of a Web-Based Frontend for an Observational SQL-Debugger

---

Philipp Moers
Sunday 21st September, 2014

Reviewer
Prof. Dr. Torsten Grust

Supervisor
Benjamin Dietrich

Abstract

Our digital world is not imaginable without database management systems (DBMS) and SQL is the de facto standard to access databases nowadays.

Since SQL is a declarative and non-imperative query language, finding faults in queries can be a difficult and laborious task. If the computed result does not come out as expected, the user often ends up replacing parts of the query arbitrarily to understand what causes the error. What the DBMS does exactly is basically hidden away from him.

The tool $^{\mathrm{PG}}\mathrm{HABITAT}$ delivers the opportunity to debug $\mathrm{POSTGRESQL}$ queries on the syntactic and semantic level of SQL itself. In order to do so, any subexpression of a potentially large and complex query can be marked and its intermediate step results observed.
The goal is to make query debugging more bearable by showing the user which values are computed and which are not. This has to be done in a user-friendly manner, in which neither unnecessary information nor technical details confuse him.

In this thesis we describe the way, a usable, web-based interface for $^{\mathrm{PG}}\mathrm{HABITAT}$ is developed and implemented as its frontend. A great variety of techniques are deployed. Based on a debugger written in Haskell, the frontend is created with HTML and CSS, built with PHP and equipped with JavaScript functionality. We incorporate features that make it possible to handle multiple markings easily and visibly observe the query computation in an understandable way.

In the end, we look at a suitable piece of software that helps to find faults in SQL queries straightforwardly or to understand the semantics of SQL.

Contents

1 Introduction

There are a lot of different fields of application in which database management systems (DBMS) and relational database management systems (RDBMS) in particular play an important role. The best known RDBMSs clearly are implementations of the SQL standard like PostgreSQL, a widely used, powerful, open source RDBMS.

In order to access the data, the user formulates queries. To be precise, we are talking about the SQL `SELECT` statement which is part of the data manipulation language (DML). Other statements like `INSERT`, `UPDATE` and `DELETE` or statements of the data definition language (DDL) like `CREATE` and `DROP` are not in focus here. Anyhow, `SELECT` is the most commonly used statement and - unlike most others - produces output of interest.

Like every code that is written by programmers, these queries may contain faults which can result in errors. Finding these faults - in other words "debugging the query" - can be a challenge, especially with increasing complexity. The purpose of a debugger is to help finding faults more easily. This saves time, money and nerves.

For SQL queries this requires a new approach for the following reason: Pure SQL is a declarative language, which means that a query describes what should be computed rather than how it should be computed. It is up to the RDBMS to apply appropriate steps to achieve that goal. The latter makes it difficult to have a look at the RDBMSs cards: Chosen algebraic plans can diverge dramatically from what queries adumbrate. Conventional step-by-step debugging is not the path to take. If we just extract intermediate step results, we need strong comprehension of RDBMS internal matters.

That is where Habitat comes into play: It offers the opportunity to debug queries

on the semantic level of SQL itself - "in their natural habitat". The user wants to find logical flaws in his query without having to understand the program execution from a RDBMSs perspective. Torsten Grust and Jan Rittinger introduced HABITAT, a true language-level observational SQL debugger [GKRS11, GR13]. The concept called **mark and observe** allows it to mark arbitrary subexpressions that are suspected of being flawed and observe the values they assume. Benjamin Dietrich implemented a version for POSTGRESQL [Die14] for his thesis for diploma: PG HABITAT.

However, PG HABITAT is still in development and its usage from the command line interface is ponderous.

In the context of this thesis, we developed and implemented a web-based frontend for PG HABITAT. It acts as a Graphical User Interface (GUI) for the tool. The goal is to make the usage easy and user-friendly.

With the frontend, we are able to visualize the evaluation of SQL queries on the level of their own language. The observation of queries is "brought near" to the user: We constructed "Binoculars for Habitat".

The handling of ternary logic can serve as a good example to show how useful the software is (Figure 1.1).

Figure 1.1: Comparison with NULL

```
SELECT x
FROM t
WHERE x = NULL
```

t.x	x = NULL	x
42	null	[no value]
43	null	[no value]
null	null	[no value]
44	null	[no value]

(a) Incorrect Comparison

```
SELECT x
FROM t
WHERE x IS NULL
```

t.x	x IS NULL	x
42	false	[no value]
43	false	[no value]
null	true	null
44	false	[no value]

(b) Correct Comparison

Comparison with `NULL` values in SQL is a common source of errors. Because `NULL` stands for the absence of a value, it can not be compared to other values. Thus, the comparison result is `NULL`, too.[1] SQL offers the `IS [NOT] NULL` construct to check for `NULL` values.

In a comprehensible way, the differences are made clear by our GUI.

This thesis not only presents the current piece of software, but also describes the way it was created, which problems occurred and how they were solved.

As usual in web development, we did not engage ourselves with complicated, scientific matters. Rather, the creation of the frontend can be compared to handicraft work and included a lot of trial and error. The sum of many little varying parts is what constitutes our work.

[1] For the purposes of the `WHERE` predicate, this is understood as `false`.

2 Using the Frontend

The best software serves little purpose if the user does not know how to use it. This chapter introduces what the debugger is all about and contains a handful of examples to show the advantages of PGHabitat and our frontend.

At the same time we will explain the ideas to make the interface usable and practical. We show the included **features** and why they are useful.

As a motivation and impression of the project, let us introduce an example that can be used to demonstrate the debugger.
We consider two small table instances with kids and toys (Figure 2.1)

Figure 2.1: Tables: Happy Kids

name	gender	age
"Rob"	"m"	10
"Sue"	"w"	16
"Dannie"	"w"	5

(a) *kids*

name	weight	minage
"Cube"	80	8
"Lightsaber"	660	15
"Wooden Train"	112	5

(b) *toys*

Now, to find out the names of all happy kids that are allowed to play with at least one toy, one could formulate the query shown in Figure 2.2. The query result on the given table instances is not what we expect to see: A table with two rows for Sue and Rob (Figure 2.3). We want to know why Dannie is missing.

Figure 2.2: Query: Happy Kids

```
1 SELECT name
2 FROM kids
3 WHERE EXISTS (SELECT 1
4                 FROM toys
5                 WHERE kids.age > toys.minage)
```

Figure 2.3: Result Table: Happy Kids

name
"Rob"
"Sue"

2.1 Basic Features

In his thesis for diploma, Benjamin Dietrich already thought of two aspects a useful GUI should cover [Die14, 7.3]:

> "A nice and clear illustration of merged observation tables. First and foremost, this includes a nested display of tables, as it is needed to present table-valued observations as well as indirect relations between observations."

Since tables are the output of a SQL query anyway, they are the most obvious and comprehensible way to display observation results. PG HABITAT itself already has the opportunity to merge many observation tables into one.

What is meant by nested tables will be discussed in Section 2.3 Nested Tables.

> "The possibility to mark arbitrary code pieces via mouse dragging instead of braces."

PG HABITAT provides that expressions are marked with braces.[1] The user really can not be expected to write and delete braces in his query manually all the time. Because

[1] We will have a closer look on that in Section 3.3 Execution of Habitat.

the mouse is an intuitive interface used since the 1990s and everything in the web is designed for it, the most simple and obvious way to mark an expression is to use the mouse.

In our frontend, a marking is set automatically every time the user selects a non-empty substring of the query. This substring can also be the whole query.

Even though the feature of **automatic marking setting** is nice and practical, it can be inappropriate sometimes. When editing text, it is common to select parts in order to cut, copy or delete them. If the user wants to do so, we give him the opportunity to disable automatic marking setting with a *checkbox*. Then, he can transform a selection into a marking by pressing a button.

Having said that it is easy to create markings, the same applies for deleting them. If the user has seen what he wanted to see from a temporary observed expression, he should be allowed to get rid of it afterwards without being interrupted in his workflow. Markings can be deleted with one click, too.

A First Debugging Process

The user can paste the query from Figure 2.2 and start to debug it by simply selecting `name` with the mouse as a first step. A new marking is created.

Automatically, a table with two rows for Sue and Rob (Figure 2.4) appears. It contains exactly the two rows of our result from Figure 2.3. PGHabitat also shows us the row values the results come from. In fact, `name` is not just a column name, but an expression that is evaluated with respect to each row value. Of course, in this case the expression is trivial.

Figure 2.4: Observing Happy Kids, Step 1

```
SELECT name
FROM kids
WHERE EXISTS (SELECT 1
              FROM toys
              WHERE kids.age > toys.minage)
```

(a) Marking

kids.age	kids.gender	kids.name	name
16	"w"	"Sue"	"Sue"
10	"m"	"Rob"	"Rob"

(b) Observation Table

We now want to know why Dannie is not allowed to play with any toy according to this query result. The flaw, of course, is the comparison operator > which should be >=. How is the debugger going to help the user finding it?

2.2 Interesting Rows

If we create an additional observation on the `EXISTS...` predicate as a second step, a new observation table with a third row for Dannie shows up. Explicitly visualized, the predicate has been evaluated to `false` here (Figure 2.5). This is the reason for Dannie not to occur in the query result. Now the distinction between the row value and the expression of the result column is clear: Because the predicate was not satisfied, there is no need for the expression to be evaluated. Hence, the observation `name` does not have a value for Dannie.
The essence here is that observations can be non-existent.

Since Dannie's row is the one we are interested in, it would be nice if we could mark it somehow. We introduce the **interesting rows** feature. On mouse click, a row is marked as interesting or uninteresting, respectively. Interesting rows are highlighted

Figure 2.5: Observing Happy Kids, Step 2

```
SELECT name
FROM kids
WHERE EXISTS (SELECT 1
              FROM toys
              WHERE kids.age > toys.minage)
```

(a) Markings

kids.age	kids.gender	kids.name	EXISTS (SELECT 1 ...	name
16	"w"	"Sue"	true	"Sue"
5	"w"	"Dannie"	false	[no value]
10	"m"	"Rob"	true	"Rob"

(b) Observation Table

with a light yellow background color. Especially for big tables this feature can be useful.
By default, none of the rows is interesting.

Additionally, another estimable feature is the **persistence** of interesting rows. If observations are created or removed, interesting rows stay interesting. In our example, Dannie's row will still be interesting after we added a third observation.

Naturally, it is possible to mark multiple rows as interesting simultaneously. We combine that with another feature: **invertible interestingness**. With just one click the user can make every interesting row uninteresting and vice versa.

2.3 Nested Tables

In order to find out why the predicate for Dannie was not satisfied, we add yet another observation on the predicate of the subquery (Figure 2.6 (a)).

What can be seen here is that this inner predicate needs to be evaluated multiple times:

Figure 2.6: Observing Happy Kids, Step 3

```
SELECT name
FROM kids
WHERE EXISTS (SELECT 1
              FROM toys
              WHERE kids.age > toys.minage)
```

(a) Markings

kids.age	kids.gender	kids.name	EXISTS (SELECT 1 ...	name				
					toys.minage	toys.name	toys.weight	kids.age > toys.m...
16	"w"	"Sue"	true	"Sue"	8	"Cube"	80	true
5	"w"	"Dannie"	false	[no value]	8	"Cube"	80	false
					15	"Lightsaber"	660	false
					5	"Wooden Train"	112	false
10	"m"	"Rob"	true	"Rob"	8	"Cube"	80	true

(b) Observation Table

kids.age	kids.gender	kids.name	EXISTS (SELECT 1 ...	name				
					toys.minage	toys.name	toys.weight	kids.age > toys.m...
16	"w"	"Sue"	true	"Sue"	[one row]			
5	"w"	"Dannie"	false	[no value]	[multiple rows]			
10	"m"	"Rob"	true	"Rob"	[one row]			

(c) Observation Table with Collapsed Subtables

kids.name	kids.age	kids.gender					name	EXISTS (SELECT 1 ...
Sue	16	w	toys.name	toys.weight	toys.minage	kids.age > toys.m...	Sue	true
			Cube	80	8	true		
Dannie	5	w	toys.name	toys.weight	toys.minage	kids.age > toys.m...	[no value]	false
			Cube	80	8	false		
			Lightsaber	660	15	false		
			Wooden Train	112	5	false		
Rob	10	m	toys.name	toys.weight	toys.minage	kids.age > toys.m...	Rob	true
			Cube	80	8	true		

(d) Observation Table with Nested Heads

subtables occur (Figure 2.6 (b)). For each outer row value of `kids`, `toys` assumed multiple values. For Sue and Rob, one row was enough to fulfill the predicate. For Dannie, all three rows of `toys` were consulted. In other words, the content of table cells may be non-atomic.

2.3.1 Depiction of Nested Tables

Actually, subtables are what can make observation results look complicated. We adapt the idea of nested tables and display tables heads detached from table bodies.
Figure 2.6 (d) shows what nested observation tables look like without this. Thanks to the **lifting of table heads**, we improve an overview, because the head is only displayed once. Also, nesting is perceived as a natural thing.

One can imagine that an observation table can become huge if we display every subtable. This is why we invented **hideable tables**: every table can be replaced with a space-saving label. Figure 2.6 (c) shows a version of the observation table with collapsed subtables. By default, subtables are only shown if they contain one row. Big subtables are not shown from the beginning. Hereby, we give the user the opportunity to orient himself first. He can show and hide subtables ad lib.

The idea of hideable tables happens to go hand in hand well with interesting rows: If a row is marked as interesting, its subtables are shown, otherwise they are collapsed. This way, the user can easily toggle the visibility of subtables by clicking on the containing row. Thanks to invertible interestingness it is possible to pull out all subtables from a totally collapsed state immediately.

Because Dannie's row was persistently marked as interesting in step 2, the user does not need to do that this time.

2.3.2 Theory on Nested Tables

Nested tables in HABITAT can be understood as an example of Non-First-Normal-Form [Kor86, 6.6] according to Edgar F. Codd who lay the foundations for RDBMSs.

Since SQL is a compositional language, a query can contain an arbitrary number of subqueries. Therefore, we need to be prepared for arbitrarily deep nested observation tables.

Row variable bindings do not influence the semantic of subqueries. That causes an identical schema of subtables in every row: Two adjacent rows can not contain subtables with different columns. This is why we can lift subtable headers. Note that we consider originally different columns[2] as one column.

Similar to the scope of a variable in imperative languages, we have scopes in our $^{\mathrm{PG}}\textsc{Habitat}$ result. Each row variable binding in the `FROM` clause of a query spans a new scope in subqueries. For instance, the `WHERE` or the `SELECT` clause can contain subqueries. Scopes can either occur alongside each other or nested. Also, the top-level query is enclosed in a single scope.

A RDBMS only returns values of the top-level scope. To observe intermediate step results, a general way to extract subexpressions must be found. It is possible that subqueries are correlated: They can contain variables that refer to bindings in enclosing queries.
This problem is solved in a general way:

> "For any marked subexpression e, $\textsc{Habitat}$ consistently understands e as a function of its free row variables." [GR13, 2]

With this approach, any expression can be separately evaluated "without its context", if we apply this function with the right variable bindings. $^{\mathrm{PG}}\textsc{Habitat}$ creates appropriate queries to receive all the values of marked subexpressions.

2.4 Aggregates

Let us expand our example with kids and toys. Consider two other tables: one for the number of remaining toys and one that tells us which kid is interested in which toy (Figure 2.7).

[2] Columns from subtables in adjacent rows

Figure 2.7: Tables: Remaining Toys

name	remaining
"Cube"	42
"Lightsaber"	2
"Wooden Train"	0

(a) *toystorage*

kid	toy
"Rob"	"Lightsaber"
"Sue"	"Lightsaber"
"Sue"	"Cube"

(b) *interests*

Figure 2.8: Query: Remaining Toys

```
1 SELECT name
2 FROM toystorage
3 WHERE remaining >= (SELECT COUNT(kid)
4                     FROM interests
5                     WHERE name = toy)
```

The query in Figure 2.8 computes the names of toys that are stored often enough to satisfy the demand of the kids. Luckily, in the result table every toy is listed.

PGHabitat has one advantage that can be seen nicely here: In the query, the SQL aggregate COUNT is used. Aggregates have something to do with so-called groups: They compute a value for every group of rows. Generally, these individual rows that build a group are not visible to the user. PGHabitat allows to observe them (Figure 2.10).

Besides, this table is suitable to motivate another feature: The **Hiding of Row Values**. In many cases, a lot of columns are redundant. With a simple *checkbox* the user can get rid of the row values. Obviously, tables will look much more compact (Compare Figure 2.10 (b) and (c)).

The Count Bug

In order to optimize the execution of computations, correlated subqueries can be unnested [Kim82]. An unnested version of the query in Figure 2.8 can look like the query in Figure 2.11. The idea is to use a join instead of evaluating a subquery for every toy. *toystorage* is joined with a temporary table that counts the number of interested

Figure 2.9: Result Table: Remaining Toys

name
"Cube"
"Lightsaber"
"Wooden Train"

kids for each toy.

This time, it may not be that obvious that the unnested query is flawed. The result table does not contain the Wooden Train. The reason for that is that there is no Wooden Train in the *interests* table at all. Thus, it can not be grouped and after the join there is no row that can fulfill the equality predicate.

While the explained fault is comprehensible with these particular table instances, it is hard to find with only the abstract unnesting instructions given. It was not found for years and got famous under the name "count bug" [GW87].

With the debugger, the temporary count table with only two rows can be illustrated.

2.5 Row Filter

PGHABITAT can handle advanced SQL constructs like recursive queries. A common example is based on a table of flights (Figure 2.13).

The query in Figure 2.14 computes all cities we can reach from Berlin. It needs to be recursive because we do not only want direct connections: Every city we can reach possibly offers new destinations.

Figure 2.10: Showing Row Values

```
SELECT name
FROM toystorage
WHERE remaining >= (SELECT COUNT(kid)
                    FROM interests
                    WHERE name = toy)
```

(a) Markings

toystorage.name	toystorage.remaining	name					
			count	(SELECT COUNT(kid...	interests.kid	interests.toy	kid
"Cube"	42	"Cube"	1	1	"Sue"	"Cube"	"Sue"
"Lightsaber"	2	"Lightsaber"	2	2	"Rob"	"Lightsaber"	"Rob"
					"Sue"	"Lightsaber"	"Sue"
"Wooden Train"	0	"Wooden Train"	0	0	[no value]		

(b) Observation Table with Row Values

name		
	(SELECT COUNT(kid...	kid
"Cube"	1	"Sue"
"Lightsaber"	2	"Rob"
		"Sue"
"Wooden Train"	0	[no value]

(c) Observation Table without Row Values

Figure 2.11: Unnested Query for Remaining Toys

```
WITH cnt (toy, cnt) AS
  (SELECT toy, COUNT(kid)
     FROM interests
     GROUP BY toy)
SELECT name
FROM toystorage, cnt
WHERE remaining >= cnt AND name = toy
```

Figure 2.12: Incorrect Result Table: Remaining Toys

name
"Cube"
"Lightsaber"

Figure 2.14: Query: Flights

```
WITH RECURSIVE destinations (departure, destination) AS
(SELECT  f.departure,f.destination
    FROM flights f
    WHERE f.departure = 'Berlin'
  UNION
    SELECT l.departure, n.destination
    FROM  destinations l, flights n
    WHERE l.destination = n.departure)

SELECT departure, destination
  FROM destinations;
```

The intermediate steps can be observed with PGHabitat. We can draw on the inversion feature here to show every subtable.

As we see in Figure 2.15 (a), tables can become big easily. It frequently happens that the user only is concerned about a small amount of the presented data. This is why we introduce **row filter**. Within a dropdown menu for each column, a particular value can be picked. As a result, the table is cleared from rows with other values. In our

Figure 2.13: Table: Flights

departure	destination	carrier	flight_num	ticket
"Chicago"	"Munich"	"AA"	"123"	500
"Chicago"	"London"	"AA"	"124"	400
"Munich"	"Frankfurt"	"GW"	"234"	50
"Frankfurt"	"London"	"BA"	"235"	70
"London"	"Stuttgart"	"BA"	"345"	80
"Berlin"	"Madrid"	"BA"	"150"	150
"Madrid"	"Honkong"	"BA"	"1400"	1400
"Honkong"	"Peking"	"BA"	"100"	100
"Honkong"	"Berlin"	"BA"	"1000"	1000
"Peking"	"Rom"	"BA"	"1000"	1200

example, the table becomes much more helpful if we filter the predicate with `true` (Figure 2.15 (b)).

Note that we filtered each subtable thanks to lifting of subtable heads. If they had their own headers like in Figure 2.6 (d), we would have had to filter them one by one. Filtering them all definitely is the better alternative.

The filter functionality is conjunctive. That means, if two columns of a table are filtered, only the rows that have equal values for both columns remain. Consequently, rows can be found quickly even in big tables.

Of course, each dropdown menu contains an entry with the opportunity to show all rows again. Also, one can reset every applied filter with a button.

The software can visualize how the flight connections come about.

Figure 2.15: Observing Many Flights

l.departure	l.destination	l.rid.l	n.carrier	n.departure	n.destination	n.flight_num	n.ticket	l.destination = n...
"Berlin "	"Madrid "	1	"AA "	"Chicago "	"Munich "	"123 "	500	false
			"AA "	"Chicago "	"London "	"124 "	400	false
			"GW "	"Munich "	"Frankfurt "	"234 "	50	false
			"BA "	"Frankfurt "	"London "	"235 "	70	false
			"BA "	"London "	"Stuttgart "	"345 "	80	false
			"BA "	"Berlin "	"Madrid "	"150 "	150	false
			"BA "	"Madrid "	"Honkong "	"1400 "	1400	true
			"BA "	"Honkong "	"Peking "	"100 "	100	false
			"BA "	"Honkong "	"Berlin "	"1000 "	1000	false
			"BA "	"Peking "	"Rom "	"1000 "	1200	false
"Berlin "	"Honkong "	1	"AA "	"Chicago "	"Munich "	"123 "	500	false
			"AA "	"Chicago "	"London "	"124 "	400	false
			"GW "	"Munich "	"Frankfurt "	"234 "	50	false
			"BA "	"Frankfurt "	"London "	"235 "	70	false
			"BA "	"London "	"Stuttgart "	"345 "	80	false
			"BA "	"Berlin "	"Madrid "	"150 "	150	false
			"BA "	"Madrid "	"Honkong "	"1400 "	1400	false
			"BA "	"Honkong "	"Peking "	"100 "	100	true
			"BA "	"Honkong "	"Berlin "	"1000 "	1000	true
			"BA "	"Peking "	"Rom "	"1000 "	1200	false
"Berlin "	"Peking "	1	"AA "	"Chicago "	"Munich "	"123 "	500	false
			"AA "	"Chicago "	"London "	"124 "	400	false
			"GW "	"Munich "	"Frankfurt "	"234 "	50	false
			"BA "	"Frankfurt "	"London "	"235 "	70	false
			"BA "	"London "	"Stuttgart "	"345 "	80	false
			"BA "	"Berlin "	"Madrid "	"150 "	150	false
			"BA "	"Madrid "	"Honkong "	"1400 "	1400	false
			"BA "	"Honkong "	"Peking "	"100 "	100	false
			"BA "	"Honkong "	"Berlin "	"1000 "	1000	false

(a) Unfiltered Observation Table

l.departure	l.destination	l.rid.l	n.carrier	n.departure	n.destination	n.flight_num	n.ticket	l.destination = n...
"Berlin "	"Madrid "	1	"BA "	"Madrid "	"Honkong "	"1400 "	1400	true
"Berlin "	"Honkong "	1	"BA "	"Honkong "	"Peking "	"100 "	100	true
			"BA "	"Honkong "	"Berlin "	"1000 "	1000	true
"Berlin "	"Peking "	1	"BA "	"Peking "	"Rom "	"1000 "	1200	true
"Berlin "	"Berlin "	2	"BA "	"Berlin "	"Madrid "	"150 "	150	true
"Berlin "	"Rom "	1	[no rows]					

(b) Filtered Observation Table

2.6 Query Management

The user probably exploits PGHabitat with a bunch of different queries or different versions of a query, respectively. We give him the opportunity to go back to a query used before.

However, saving queries manually quite cumbersome. We take that off the user's shoulders by saving every observed query to a query history automatically. In addition to that, the user can manually pin and unpin queries that are important to him for any reason.

Since browsing recent queries and pinning them is not part of the main functionality of the debugger, it should not disturb the workflow on the main page and therefore be on a separate page. This leads to the following more general concept.

2.7 A Tidy Table

We need to make sure that our GUI - while meeting all of the requirements - still looks nice and not overloaded, so that working with it over a longer period of time is as pleasant as possible.

Lifting of subtable heads is an important step not to make the table look knotty.

Since the observation table is the core of the debugger and the user easily spends hours inspecting it, it should make a tidy impression. Thus, the access to the features is concealed, yet still they can be reached easily.

The inversion of interesting rows is suggested when the user moves the mouse over a scope header. It is performed on mouse click. Likewise, if a column is filterable, the dropdown menu is shown when the header is clicked.

In addition to that, a button makes it possible to hide every subtable to start from a compact top-level table. If everything is hidden, a second click shows every subtable again, so that the whole PGHabitat result is visible. In difference to the inversion

feature, this does not only affect the rows of one scope but every row in the table.

To free a space for the observation table is a good way to give the page an open layout. That is why there is nothing else than the necessary control elements.

Moreover, there is an option to **hide the editor**. This makes the observation table the center of interest all the more.

3 Designing the Frontend

In the previous chapter we saw what the frontend looks like, which features it includes and how it is used. But how is it brought to practice? We will sketch the architecture in general, which technologies we use and how they interlock.

3.1 Technologies

To implement the GUI, we use the HTML5 and CSS3, which is state of the art right now. The new elements *article*, *header*, *nav* et cetera help us to structure the page.

When writing applications for the web, it is common to use a framework, a template engine or something that prevents us from writing basic HTML repetitiously. But to use a big CMS like TYPO3 would mean an unnecessary huge overhead to us. Although it would have been fun to create our frontend in HAPPSTACK - a Haskell based web application server - we stick with the popular APACHE HTTP server, version 2.2.22. APACHE can run on a wide range of operating systems and can be installed easily.
We build the frontend with the scripting language PHP version 5.3, which comes with APACHE at will. PHP allows us to create dynamic web pages and run scripts on the web server.

Most of the functionality of the frontend is built with JavaScript. JavaScript is used in nearly every modern web page and offers endless possibilities. This is what we need to make our software interactive.
Furthermore, we use D3.JS, which is not as well known as the previous technologies. D3, as we will call it from now on, is a JavaScript library which makes handling and

visualizing data more easy. We can "join" data to existent or not-yet-existent HTML elements. Also it allows us to implement animations. D3 can be compared to the more popular JQuery.

What we need to keep in mind when developing is where both of these scripts - PHP and JavaScript - are executed. PHP is an HTML preprocessor: A PHP script is lying on the server waiting to be requested. It will be executed to create a document that can be delivered to the client. So the client never sees any PHP code.
A totally different purpose is served by JavaScript: It is the job of the client - meaning the web browser - to interpret and execute functions of a JavaScript document. With JavaScript we can react to mouse events and manipulate the document on the client side.

3.2 Stateful Web Pages

3.2.1 PHP Superglobals

PHP knows the concept of superglobal variables [PHP, Predefined Variables]. To be concrete, we use `$_SESSION` and `$_POST`. These superglobals are available in all scopes within a script.

`$_POST` is an associative array that stores every variable passed to the script with the HTTP POST method [RFCb]. In practice this means that we can use a regular HTML *form* to transmit a query to the server.

But there is more information the server needs from the client. Most of it does not change when we send requests multiple times in a row and should not be transmitted that often anyway. Here is where another superglobal comes into play: `$_SESSION`. This variable is independent from individual requests. Its lifetime is unspecified. By using it, we can generate and keep a state on the server side, which can last as long as the user wants to work with PGHabitat. Each PHP script that wants to use the `$_SESSION` variable correctly needs to call `session_start()` at the beginning.

We use `$_SESSION` to maintain a state on the server side. For instance, we store the database connection parameters there.

3.2.2 Asynchronous JavaScript and XML

Asynchronous JavaScript and XML - or short **AJAX** - is not a single technique but stands for one idea that can be summarized like this: A web application can send and retrieve data independently from loading or refreshing the page. AJAX allows us to communicate with the server while keeping the same page and its state on the client's side.

This suits our frontend perfectly: We want to get PGHabitat results from the server several times but we do not want it to interfere with the displayed web page. If the user creates a new marking we need to consult PGHabitat again, but it would be unpleasant to reload the whole page every single time.

As a consequence, we call a JavaScript function that sends HTTP requests and retrieves their results. The JavaScript object type that is prefabricated for this is called `XMLHttpRequest`. Beyond that, the function that sends the request calls another one, which is responsible for handling the result.

Because what we receive is formatted in JSON [1] and not XML, the more precise term would be AJAJ (Asynchronous JavaScript and JSON) here. This actually is a thing, but probably thanks to pronunciation difficulties the term AJAX is used, even if JSON is the transmitted data format.

Moreover, the lifetime of JavaScript variables can extend over several requests. This means that their values represent a state that can last until the user leaves the web page. We therefore use them to store information about markings for instance.

[1] More on that in Section 3.4 JSON Tables

3.3 Execution of Habitat

There are two main reasons for us to do some thinking about an appropriate interface between PG-HABITAT and its GUI:

1. PG-HABITAT is written in Haskell while our frontend runs as a web page delivered by a web server. Somehow the JavaScript engine needs to interact with PG-HABITAT.

2. The process of debugging a query consists of creating and deleting a great amount of markings most of the time. Keeping a consistent state, PG-HABITAT has to be consulted many times in a row.

The solution we did choose is a binary. It is executed by a PHP script called `habitat.php`, which returns the PG-HABITAT result to the client. So the web server runs on the same machine where the binary is located.

Furthermore, the executing user must have `habitat` in his `PATH`-variable and must have the rights to execute it. APACHE is started by the root user and switches to another configurable specified user to perform such actions [APA, Security Tips]. On Linux machines like ours, the default user created for this is `www-data`.

3.3.1 Usage of Habitat

When operating the PG-HABITAT executable in the terminal, we need to feed it with a special string. PG-HABITAT reads from *stdin*, so we can use the `echo` command in combination with powerful UNIX pipe.

But how is this special string made? Basically it is just the original query. In fact, PG-HABITAT can be executed without errors with plain SQL queries. But in order to make profit out of PG-HABITAT, of course we need to mark expressions. This is done with curly brackets, also known as braces ("{" and "}"). Any marking represents an observation and can and should have a unique identifier (**marking ID**).

Now consider the query

```
1 SELECT name FROM kids
```

Let us suppose, we want to observe which names are evaluated. Randomly using the ID `xy42`, the string PGHabitat wants to see would look like

```
1 SELECT {xy42{name}} FROM kids
```

To work properly, the binary also needs a connection string in manner of PostgreSQL to connect to the database [POS, 31.1.1].

We need the following information:

- a username,
- a password,[2]
- the host,[3]
- the TCP/IP port on which the PostgreSQL daemon is to listen.

We choose the connection URI variant.

In normal mode, PGHabitat produces some SQL statements like `CREATE TABLE...` and `CREATE FUNCTION...` that can create tables which contain the values of the marked expressions (2.3.2). The output also comes with statements to query those tables and merge them into one table. However, there is an option `-JSON` that executes all of these statements and wraps the result in a JSON string.

Figure 3.1 shows the command we need to execute.

[2] This is only necessary if this is the users authentication method. PostgreSQL offers different methods.

[3] We are running the database process on the same machine PGHabitat and the frontend are running on, so that would be `localhost`. But technically, any machine that is reachable through the network would work here.

Figure 3.1: Habitat Usage

```
1 echo 'SELECT {xy42{name}} FROM kids' |
2 habitat user:password@host:port/database -JSON
```

3.3.2 The Habitat Script

The `habitat.php` script does execute the command we just explained. But there are a few more notable aspects.

In difference to others, this PHP script does not echo an HTML document but the output of PGHabitat. Hence, the so called "Content-Type", which is declared in the HTTP header via the `header` command [PHP, Other Services - Network Functions] should not be "text/html" but "text/plain" instead [RFCa].

In fact, the script is completely detached from the user interface and does not manipulate or prepare the PGHabitat output in any way. Other applications that are no GUIs could take advantage of this.

The script gets its input from `$_POST` and `$_SESSION` (3.2.1). It reads the original query as well as the marked query from `$_POST` and writes or overwrites the corresponding variables in `$_SESSION`, respectively, if and only if they are set in `$_POST`. After this, the `$_SESSION` variable is used. For this reason, we can execute the script again with an empty `$_POST` variable and still get a result.

The database connection parameters are kept in `$_SESSION`. In addition to that, we store a boolean flag that tells whether the executable should be searched in `PATH` or the one included in the project should be used. This list of variables could possibly be extended in the future.

Moreover, the query is logged in another variable, the query history.[4] The history is implemented with an array, in which the query is prepended if it can not be found already. We do not want the same query to appear multiple times.

[4]Why this is needed can be seen in Section 4.5.1 Implementing the Query Management Page.

Now, we have to prepare the command we want to execute. For the connection string this is a little tricky: Since the username is actually an optional parameter, we can not use the `@` character in every connection string. Instead, we create a variable that contains the username with an optional password and the `@` character that only will be set if a username is given. Similarly, the password's `:` character only is prepended if a password is given. The same applies for port and the database with `/`.

Execution in Shell

Finally, the command must be executed. This is done by PHP's `shell_exec` function, which returns the output of the executed command on *stdout*. PHP uses the SH shell for this.

Since we are also interested in everything that comes out of the *stderr* stream, we simply redirect *stderr* to *stdout*. In SH this is done with `2>&1`.

What caused more trouble is the escaping of single quotes. It does make sense to define the command in double quotes. This is how we can insert our variables easily, because they are expanded [PHP, Types - Strings]. The query string that should be echoed is put in single quotes, so that every character is taken literally except the single quote itself.
This works fine as long as there is no single quote inside the query string. In case there is one, it terminates the string and the subsequent characters are not interpreted as they should anymore. This would not only break the proper functioning of the software, but also be a huge security issue, because one could execute arbitrary commands on the server by choosing a suitable string. What we need to do to prevent this, is to escape all the single quotes.

When trying out PGHABITAT in the terminal it is important, which shell we use. In FISH for instance, we can escape a single quote with `\'` as it is common in nearly every programming language. In BASH or SH however, this is not possible, because characters are read one by one.

To solve this problem, we can replace every `'` with `'\''`. This terminates the string,

puts a single quote character behind and starts a new string. The scripts still works, because the `echo` command can take more than one argument.

Instead of doing that manually, we now use a cleaner solution and call the PHP function `escapeshellarg` which is made for that purpose.

3.4 JSON Tables

JavaScript Object Notation (JSON) is a standard to transmit structured data in textual, human-readable form. JSON is a great interface for our purposes for several reasons:

- It is really simple: There are a few basic data types (null, integer, boolean, string) that roughly match the SQL data types.
- More complex types can be built using arrays (`[...,  ...]`) and objects with key-value pairs (`{...:..., ...:...}`). This also allows nesting.
- Correct JSON per se represents a correct JavaScript Object when parsed and evaluated.[5] The choice could not be better since this is the language we work with.

A table can be expressed in JSON as follows:

- A row is represented by a JavaScript object. Column names are the keys of the object's attributes and map to the row specific values.
- Multiple rows are summarized in arrays.

The *kids* table from Figure 2.1 (a) would have the JSON representation shown in Figure 3.2.

This is the format we use to represent our data. Firstly, this is how PGHabitat delivers the results. Secondly, this is the notation to have in mind when handling tables and building HTML in the end.

[5] This is obvious if we bring home the fact that JSON was derived from JavaScript originally.

Figure 3.2: Table represented in JSON

```
[ { name: "Rob", gender: "m" age: 10 },
  { name: "Sue", gender: "w" age: 16},
  { name: "Dannie", gender: "w" age: 5 } ]
```

For a table that is represented in JSON format - a JSON table - we will use the abbreviation **JT**. With **JTAN** we refer to the JSON table attribute name - the key of an attribute.

To get an impression what a PGHabitat result looks like, let us consider the example from Figure 2.6 again. A part of what PGHabitat spits out is shown in Figure 3.3.

3.5 Project Structure

On a request, a web server like Apache consults a default page, conventionally called `index.php`. Ours just sets a default query in `$_SESSION` if it was not set before and redirects to the welcome page.

We separate files of different types in different folders: There are folders for JavaScript files (`js`), CSS files (`css`) and image files (`img`). Every accessible web page is saved as a PHP file in `pages`. The `php` folder, however, contains other scripts that do not represent a web page. To avoid redundancy we do not save parts of web pages that occur on every one of them - like the header and navigation - multiple times. These template parts are saved in `parts`.
We also have a `bin` folder containing the PGHabitat binary. For this reason it is not necessary to have a binary of one's own. Hence, PGHabitat works out of the box. Code from external sources like CodeMirror or D3 can be found in `vendor`..

The `php/config.php` file is included by every web page. Its point is to provide some global constants. We define constants with paths to the document root and to the project folder. This is useful for inclusions of other files. For JavaScript and CSS files on the one hand, absolute paths begin with the document root of the web server.

Figure 3.3: Output of Habitat

```
{
    "xScope.rid.0.0": [{
        "rid.0.0": 0,
        "xScope.rid.1.1": [{
            "rid.1.1": [1],
            "rvar.s1:kids": {
                "name": "Sue",
                "age": 16,
                "gender": "w"
            },
            "vObs4:rzQo": true,
            "vObs5:jlsH": "Sue",
            "xScope.rid.2.2": [{
                "rid.2.2": [1],
                "rvar.s2:toys": {
                    "name": "Cube",
                    "weight": 80,
                    "minage": 8
                },
                "vObs3:qjMz": true,
                "exists:vObs3:qjMz": true
            }],
            "exists:vObs4:rzQo": true,
            "exists:vObs5:jlsH": true,
            "exists:xScope.rid.2.2": true
        }, {
            "rid.1.1": [2],
            "rvar.s1:kids": {
                "name": "Dannie",
                "age": 5,
                "gender": "w"
            },
            "vObs4:rzQo": false,
            "vObs5:jlsH": null,
            "xScope.rid.2.2": [{
            ...
        }],
        "exists:xScope.rid.1.1": true
    }],
    "exists:xScope.rid.0.0": true
}
```

For PHP inclusions on the other hand, absolute paths begin with the root directory of the machine the web server is running on.

For now, the project consists of the following pages:

- **The Welcome Page:** A welcome is extended to the user. This page is animated with D3 and therefore requires running JavaScript. The script redirects the user to the debugger page after a few seconds (not done by PHP this time!). This deliberately does not work with JavaScript disabled, because the debugger is designed for JavaScript anyway. If it is disabled, the user is asked to enable it via HTML *noscript*.
- **The Debugger Page:** This is the heart of the whole project. Here the user can observe evaluated expressions and refine the query.
- **The Query Management Page:** Within the debugging process the user probably changes the query multiple times. On this page he can go back to older versions or pin queries.
- **The Setup Page:** This is where the database connection and similar things are established.
- **The Help Page:** Any good software should have a help section to assist the user when he gets started. On this page, he can find some basic instructions. He can read about the features in case they are not obvious and self-explanatory enough to him.

Except for the Welcome Page, all of those can be accessed in the navigation on the top right of the header.

If a page has some page specific JavaScript functionality, it can be found in a separate file which has the same basename (for instance `pages/debugger.php` and `js/debugger.js`). The same applies for CSS files. These files are included by the corresponding pages.

4 Implementing the Frontend

The requirements and the general functioning of the project are clear now. Finally, this chapter deals with the realization of the frontend.

We created a web page that is plain and unobtrusive instead of florid and posey, because this is what fits a debugging tool best.
A straight red header with a navigation section at the right and a footer, they both build the frame for the content. Nonetheless, we added some shadow and zooming animation on mouseover with $\mathrm{D3}$ to give it a modern look and feel. Last but not least, the Welcome Page should make a high quality impression on the user.

But what we should give priority to is the functionality of the debugger. Most of it is provided by JavaScript functions. Writing, debugging and improving these definitely was the bulk of the work.

In the following, we will go through some aspects of the scripts. This chapter might be understood as some kind of documentation. But if we leave out insignificant implementation details, this can be a good way to show the inner workings of the software. Also, we will mention some problems that occurred during the development process. In fact, the implementation is topic of the thesis anyway.

The Debugger Page

The Debugger Page has a simple layout: There is the editor on the left containing the query and the observation table on the right. If the table is too big, it is displayed below the editor. The relationship between markings in the editor and in the observation table is visually clear because of two things:

- The texts of markings (or parts of them) appear as labels in the table header cells.
- The colors of markings in the editor and the colors of the accompanying table header cells are the same.

We reserve a *div* container for every part of the page. The horizontal layout works thanks to the CSS attribute *float* set to `left`, although *divs* are block elements which are displayed one below the other by default in web browsers.

In many cases it is a good idea to divide parts of the script into multiple files to have things tidy. After trying this for a while, we decided not to follow this principle. The reason for that is the fact that handling a single file of code is easier sometimes and some of the functions share global variables to interact with each other. Knowing that this can cause trouble, it turned out to be the best way to get things working for us. There are ways to use JavaScript in a more "functional" way for sure.

From the creation of markings to the rendering of the observation table, many functions are called. To make everything less confusing, let us draw a call graph with some of them (Figure 4.1). This should explain the relationship of the functions we elaborate on in this chapter.

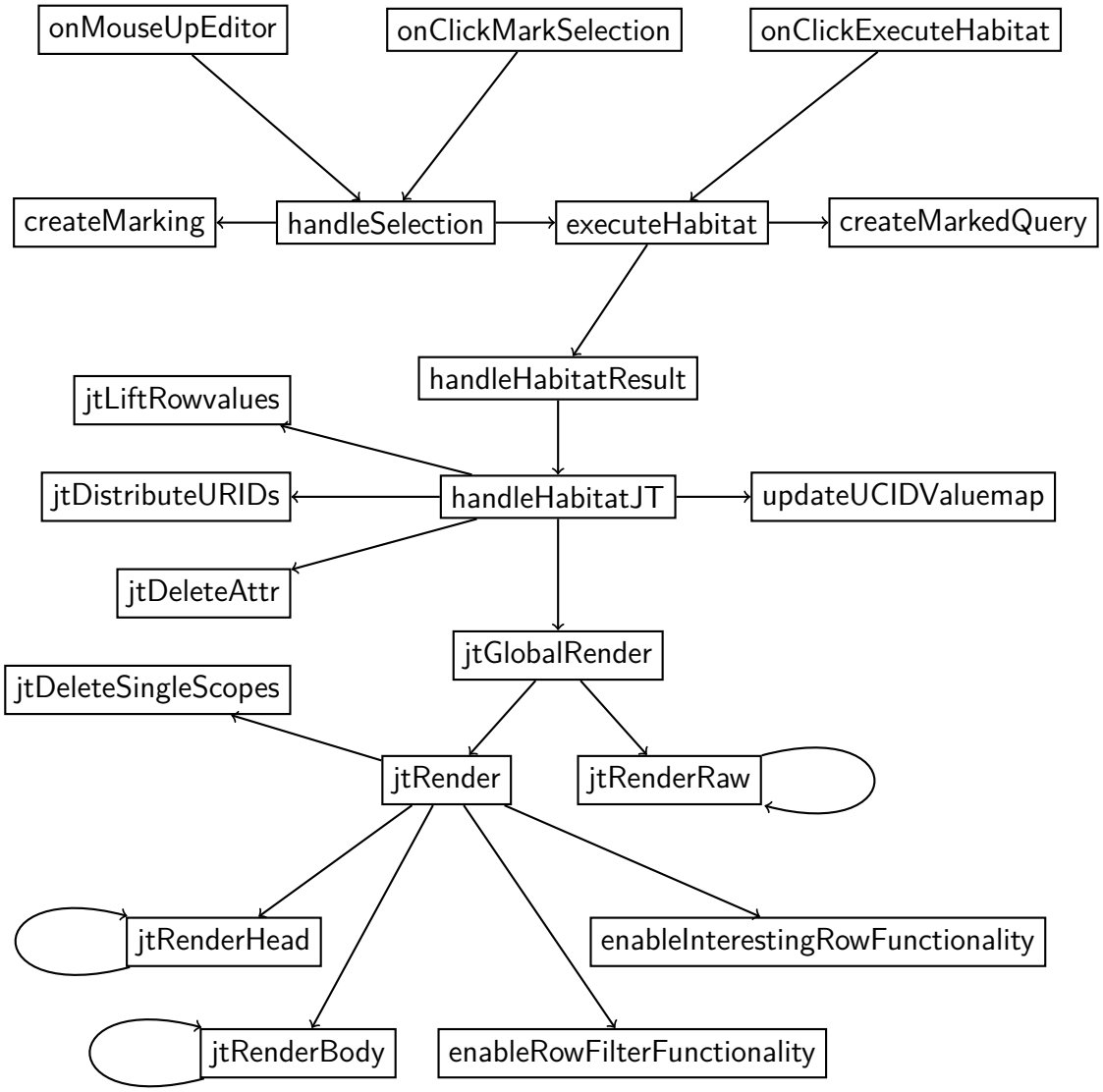
In the context of our work, with **rendering** we mean the access of the DOM to display a JT, not the actual work that the browser does.

4.1 Markings

4.1.1 Saving Markings

As explained before (3.2.2), we use JavaScript variables to remember a state. This is why we can save markings in a JavaScript array. A marking is represented by an object with a bunch of attributes: ID, textual content of the marking, a list of CodeMirror marking objects, start position, end position and color.
Positions are objects themselves, containing two integers for a line number (`line`)

Figure 4.1: Call Graph



and the column number within the line (`ch`).
We absolutely do need the CodeMirror marking objects, because there is no other way to delete them later on.

If a new marking is created, we have to do several things:

- We must generate a new ID and a new color.
- We must append a CSS class to the document.
- We must save the marking persistently.
- We must update the view for the user.

The ID allows us to identify a marking. Hence, we define a function `createRandomID` that randomly creates strings from a predefined set of characters of a given length. To make sure that marking IDs really are unique as they should be, we define a wrapping function `createMarkingID` that creates an ID, checks that it is different to all existing marking IDs and creates new ones if necessary before returning it.

The braces mentioned in Section 3.3.1 are not inserted into the query when a marking is created. Rather, when we are about to send a request to the `habitat.php` script, we call a function `createMarkedQuery` that inserts them simultaneously. Otherwise, the implementation would have been more complicated since we had to find out the exact positions with respect to existing markings.

4.1.2 Colors

The color of a marking is what makes it recognizable at first sight. Their choice should not be underestimated. Thanks to definitions like the CIE 1931 color space [SG] we can describe colors in a mathematical way. According to Grassmann's first law, a color is a three-dimensional quantity [Pog53]. We can divide colors in three components: brightness, hue and colorfulness. Hue and colorfulness together form what is called chromaticity. If we only adjust the brightness component, we get monochrome colors

that feel close. There are endless tools around the web to create such groups of colors.

For our markings we predefine some colors in the `config.js` file. These are preferred if markings are created. If there are no predefined colors left, but more markings created, we create random colors. Just as `createRandomID` did, `createRandomMarkingColor` checks all existing markings first.

Examining if two colors are the same, is not sufficient here anymore. Instead, we need to make sure that they are "not too similar". Thanks to their mathematical representation, we can compute a color distance, which is implemented as `colorDistance`. In web development it is common to express colors in the RGB color model. Distances in RGB color spaces do not perfectly match the human perception of color distances. This is of minor importance for our frontend, but a clean solution would be to convert to LAB color space.

4.2 The Editor

To enter and edit a query, we need to have a text editor, rather than a simple text area. This way, we can profit by features like syntax highlighting, automatic bracket matching and custom keybindings.

There are a bunch of JavaScript based source code editors for the web: Ace, CodeMirror, jsvi, Orion and YMacs, just to name a few. We choose CodeMirror, which is highly customizable, open source and has a rich API.

The editor is instantiated by a JavaScript function, replacing a simple text area. The content of this text area has been set to the current query before, saved in the PHP `$_SESSION` variable.

When we create the editor, we can define a configuration object to customize it. We do that by setting the theme, enabling automatic closing brackets, et cetera. Also we set the mode to `text/x-sql` enabling syntax highlighting. CodeMirror offers support for different dialects of SQL out of the box, each defined in JavaScript files. Also it is possible to define own modes.

We want to make the editor interactive, so we register event handlers for different event types. With `on` we can link the events that CodeMirror fires to arbitrary functions.

When the content is changed, the query gets a new semantic meaning. Totally different values could be observed. Maybe the user is about to debug a whole new query. We react on the "change" event by resetting all markings and hiding the observation table.

4.2.1 Creating Markings

As explained (2.1), we want to set markings when the user selects some text with the mouse, which happens after the mouse button is released. Sadly, CodeMirror emits the events "mousedown", "dblclick", "keyup" and a few more but not "mouseup" [CMA, Events], so we can not get this functionality to work this way. What we do to implement it nonetheless, is to use the JavaScript function `onmouseup` on the wrapper element of the editor (an ordinary *div* element).

Besides, the execution of `habitat.php` is set in motion automatically whenever a marking is created or deleted, too. This feature can also be disabled with a *checkbox* and performed manually. Mouse-hating users can mark expressions with F6.

If PGHabitat is invoked with an invalid marking, it responses with a parse error. Currently we allow every possible selection to act like a valid marking. In case of a parse error we print an error message where the observation table would have been. There is a function that checks the validity of a marking which just returns true yet. To implement it, we would need knowledge of the semantics of the query, which means we would need a parser or some other possibility to figure out whether the selection is an observable expression.

To highlight a marking in CodeMirror, the API offers a `markText` function. It takes start and end position as well as an option object that allows us to configure the marking [CMA]. We used to call it directly in `createMarking`, but do not do this anymore. The reason is explained below (4.2.2).

What we do nonetheless is create a CSS class. This class can be specified in the option object and allows us to define how a marking should be visualized. Let us use the marking specific color as a background color.

After marking some text in CodeMirror, the text selection (gray background) is not visible anymore because it is overlaid by the marking color. When it is still present, the default behavior on mouse click and movement is drag and drop, so that the text can be moved to another place. This is not a wanted feature in our frontend and leads to an ugly bug: It is not possible to set a new marking on top of another without clicking somewhere else first. Our solution is to set the cursor to a new position with CodeMirrors `setCursor` function. This way, the selection is removed every time a new marking is created.

4.2.2 The Redraw-Markings Bug

The `redrawMarkings` function is responsible for rendering the colored markings in the editor. It is called every time a new marking is created or an existing marking is deleted. This is the easiest way to make sure the illustration of markings is correct. That may not be the case without further adjustments, because markings may overlay each other.

Figure 4.2: Overlaid Markings

Marking m_2 (orange) lies on top of marking m_1 (yellow).

In CodeMirror it is possible to draw a marking m_2 on top of another marking m_1. If the nested marking m_2 is created last, it will be displayed correctly. However, if we create the markings the other way round, m_1 will overlay m_2, so that m_2 is not visible anymore.

Our first idea was to redraw every marking in the correct order, meaning that markings with smaller `from`-positions are drawn earlier.

This requires to sort our `markings` array. The JavaScript function `sort` which can be called on arrays is capable of taking a sorting function f_{sort} that compares two elements of the array [ECM, 15.4.4.11]. Therefore, what we need to do, is to define a function that compares two markings. Again, this function requires a function that compares two positions: We implement it under the name of `posCompare`. Naturally, lines take precedence and only if they are equal we compare the character positions.

We sort all markings by `from`-position in ascending order, or if equal by their `to`-position in descending order. The latter does make sense for the following reason: The later a marking ends the earlier it should start, so that it surrounds other nested markings.

After sorting, the `redrawMarkings` algorithm did iterate over all markings, deleted them via `clear` [CMA] and immediately drew them again.
The result was not what we expected: Still surrounding markings overlaid others if we created them afterwards. Both sorting and clearing/redrawing (for single markings) seemed to work properly.

After a while, we figured out that CodeMirror apparently does not delete a marking completely on a call of `clear`. At least, drawing an inner marking that was created first again does not have the desired effect.

We fixed this issue with a new algorithm that is kind of a workaround: CodeMirror markings are not overlaid anymore. Instead, a PGHabitat marking can consist of multiple CodeMirror markings, depending on the existence of other markings within. The algorithm we developed to compute the positions of markings is quite interesting. Let us have a look (Figure 4.3).

When drawing a marking m, we check every marking within (line 4 and 5) and draw from a current position cur to the beginning of the inner marking mj. Then we set cur to the end of the inner marking. It is important that we leave out inner markings that are nested deeper than one level, because we do not want to draw m where mj will be! We assure this by having incremented cur before: If we find a marking that is nested too deeply, its $from$ position must be smaller than cur (line 7).

Figure 4.3: The Redraw-Markings Algorithm

```
sort markings
for each m in markings
    cur = m.from
    for each mj in markings starting after m
        if mj.from > m.to
            break
        else if mj.from >= cur
            draw from cur to mj.from
            cur = mj.to
    draw from cur to m.to
```

4.2.3 The Marking Control Section

The user should have easy control over all set markings. It comes with the territory that markings should be deletable one by one, as mentioned in Section 2.1. In order to implement that, we introduce the Marking Control Section to the editor's right. We go without an extra heading. The section is a simple list of all current markings, each with a red cross to delete it. This does not require explanations for the user, because it is self-explanatory. If there are two or more markings, we append an additional cross to delete all of them.

We also thought of more features here like changing the marking's text, changing the marking's position or activate and deactivate markings. But since they can be created and deleted so easily and the query can be changed inside the editor, this really does not make sense.

The list of markings is not existent when the page is loaded, but completely appended to the page accessing the DOM with JavaScript. We use D3 for this and join the data from `markings` with new *tr* elements. Because the row should contain a cell with the marking's text which can be quite long, we cut it off at a configurable length. We use the same CSS class as used in the editor to style the table cell. The red cross image is equipped with an alternative text and a title that is shown as a tooltip, as well as the most important thing: a function that deletes the marking, updates the

view and optionally executes PGHabitat again.

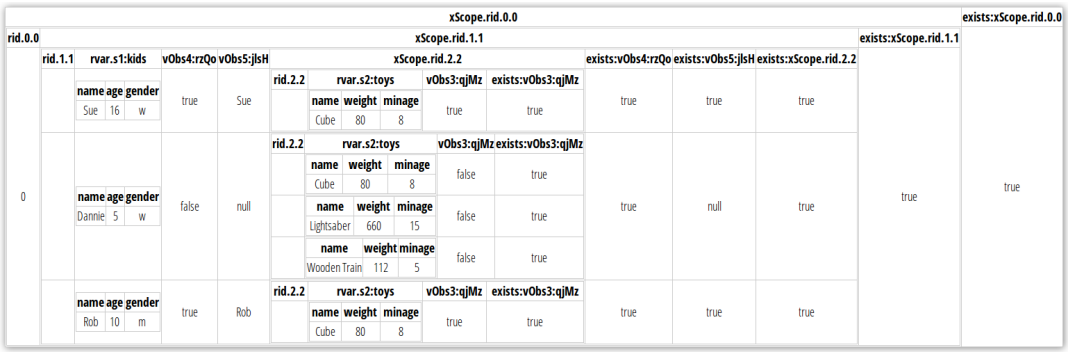
A tiny animation increases the width of the table cells. The implementation caused another of these typical, unexpected, little issues, that nonetheless needs to be solved: The observation *div* also moved with the growing width. So, we set the CSS attributes *min-width* and *max-width* of the marking *div* in order to fix this.

4.3 JT Processing

If we bring it down to its simplest level, the job of the frontend is to display the JT that was produced by PGHabitat. Of course there are several things that need to be done: We need to process the JT.

For development we also have a rendering function that omits these processing steps and shows the raw results. The table from Figure 2.6 could look like the table in Figure 4.4.

Figure 4.4: Rawly Rendered Table



As we have seen, JTANs eventually become column names when it comes to rendering, even though textually adjusted and not every one of them. The prefixes of JTANs that PGHabitat produces are listed in `config.js`, because we do not want to hard-code them in our functions.

4.3.1 Existence of Scopes and Observations

In addition to every scope or observation, a JT carries a boolean attribute that tells us if it exists (Figure 3.3). Its JTAN differs from the accompanying one only in the special prefix `"exists:"`. How non-existing values can occur was explained in Section 2.2.

If we just filled non-existent values with NULL values, like it was in former PG HABITAT versions, we could not tell whether there is no observed value or SQL NULL. Since this can make an important difference (Figure 1.1), we use the additional attribute that is set to `false` if and only if a scope or observation does not exist.
Of course, we will not render these additional attributes visible for the user. But we can not delete them until we render the table, because they constitute important information of our JT.

4.3.2 Arranging a State

The PG HABITAT result is considered to be part of the state we mentioned before. We save different things in global variables:

- `habitatResult`: This is the unmodified textual result we get from the `habitat.php` script as a string. It also contains potential error messages. `executeHabitat` is the function that writes it.
- `habitatJT`: This is the parsed result as a JavaScript object. `handleHabitatResult` is the function that writes it.
- `jtGlobal`: This is a modified version of `habitatJT`. We perform some processing on it to get the object that represents the observation data. `handleHabitatJT` is the function that writes it first.

All of the JT processing can be divided in two parts: The processing that happens to arrange a global representation of the observation data (`jtGlobal`) and the processing that happens afterwards to render the table for the user. The rendering function does consult the `jtGlobal` instance but does not modify it anymore.

In case of an error, we present an appropriate message to the user in order to help him to fix the problem. Errors can occur for various reasons: The query could be syntactically wrong, PGHabitat could have difficulties with the database connection, the query could be incompatible with the database state, et cetera. Also, parsing the script's result could fail.

In case of success, the next step is to make a deep copy of the `habitatJT` object for `jtGlobal`. There is no built-in JavaScript function to do that. One of the easiest implementations is the conversion to a string which is parsed afterwards. Amazingly, this can be faster than the copying function from jQuery, for instance.[1]

4.3.3 General Modifying

A handful of modifications are made to `jtGlobal`, and to its copy for rendering, respectively. Some of the functions return an object, some do not, but all work directly on a JT.[2] Since a JT can be nested arbitrarily (Section 2.3.2), these functions have to be recursive. Most of them follow a common pattern: Inspect the current array (table) or object (single row), do the work and recurse where necessary. Their usage is independent from the JT it is used for as a general rule. This means, they can be used for arbitrary JT instances in further development on demand.

The following JT processing functions are notable:

- `jtDeleteAttr`: It is useful to delete certain attributes. In most cases we know the prefix of the key of an attribute we want to delete. This is why this function takes a JT and a key prefix. For the prefix, the known prefixes from `config.js` suggest theirselves.
- `jtDistributeURIDs`: This function adds some new attributes - called URIDs - to a JT, that helps us to identify rows. More on that in Section 4.3.4.
- `jtLiftRowValues`: As we have seen in Figure 3.3, a row value is a single

[1] For details, see `http://web.archive.org/web/20140328224025/http://jsperf.com/cloning-an-object/2`.

[2] This principle is known as programming "is situ".

attribute that has an object as a value. This object again contains one attribute per SQL column. If we rawly render a table, it appears as a nested subtable with one row (Figure 4.4). Many of them may look confusing. This function lifts every row value attribute into the parent row. The new JTAN is built as a concatenation of the row value JTAN, a delimiter and a string to identify the column. The key of the row value object would not be enough for this string, since there easily could be another table with the same name: we would loose one of them. In SQL-syntactic style we concatenate the table alias, a period and the key.

- `jtDeleteIgnoredRows`: This function deletes some row objects from a JT. It takes a list of URIDs to identify these rows.
- `jtEditNonExisting`: With this function we can edit the value fields of observations or scopes that do not exist. They are replaced with a special string.
- `jtDeleteSingleScopes`: This function simplifies a JT by substituting it with its own content under certain circumstances. More on that in Section 4.3.5.

4.3.4 URIDs and UCIDs

Unique Row Identifiers

It is always a good idea to have unique identifiers for objects. Therefore, we should have the opportunity to identify rows. In SQL databases, row values can occur multiple times, so it is no option to use them. Luckily, $^{\mathrm{PG}}$HABITAT already delivers row identifiers (the JTAN is "rid") for every SQL data row. Also for groups, as they occur when we use the GROUP BY clause or SQL aggregates, there are group identifier ("gid"). The problem is: They are only unique within their innermost scope. What we need are globally unique row identifiers. We will call them **URID**s.

If we think about it, the concatenation of a RID and potentially nested pairs with the JTAN and the next RID is unique. This is how we build URIDs. Let us quickly have a closer look on how a URID is structured (Figure 4.5):

Figure 4.5: URID Example

```
urid$[2]$xScope.rid.2.2$$[3]
```

Figure 4.6: URID Regular Expression

⟨prefix⟩ (⟨d⟩ (⟨rid⟩⟨d⟩)?)* ⟨d⟩⟨d⟩⟨rid⟩

Every URID starts with a determined prefix (`urid` to keep it simple) and ends with the RID of the last nested row (`[3]`). As we see, PG HABITAT'S RIDs are arrays of integers at the moment. Anyhow, we stringify[3] it. Between these two parts, the list of RIDs and JTANs we collect when we traverse the JT can be found. We separate them with a special delimiter (`$`).
For prefix ⟨`prefix`⟩, delimiter ⟨d⟩, JTANs ⟨`jtan`⟩ and RIDs ⟨`rid`⟩, we can formulate a regular expression for URIDs, shown in Figure 4.6.

The double delimiter right before the last RID is important. This way, we make sure that all URIDs build a prefix-free code.[4]

The function `jtDistributeURIDs` that processes a JT and distributes URIDs recursively is called in `handleHabitatJT` where we prepare `jtGlobal`. In its implementation, we attach a URID to exactly those rows that already have a RID/GID. The function just appends URIDs to a JT, which means after using it, the JT has URIDs and RIDs/GIDs. We now can benefit from `jtDeleteAttr` and delete RIDs and GIDs, because they are not needed anymore. URIDs, however, will not be deleted until we render the JT.

Unique Column Identifiers

Just like rows, we want to identify columns. Therefore, we introduce globally unique column identifiers (**UCID**s). They look basically like URIDs, but do not include RIDs,

[3] This is the catchy term for converting a JavaScript value to a JSON string.
[4] No URID is prefix of another URID.

since rows do not influence the structure of columns (2.3.2). Instead of the last RID after the double delimiter, we append the JTAN of the column. Also, naturally, the prefix is `ucid`.

We define helpful functions that for instance can calculate the UCID from a given URID and JTAN (`calcUCID`) or calculate if a URID matches a UCID (`matchesUCID`).

On this basis, we can prepare something that is of service for the row filter functionality[5]: A global hashmap, called `ucidValuemap`, in which for every column the existing values are stored. `updateUCIDValuemap` is the function that traverses `jtGlobal` and fills the value map. We decide whether a column is filterable on the run and draw up a few restrictions:

- The value should be atomic,
- it should be either a row value or an observation,
- the row should have a URID.

Before we store a value of an observation, we make sure that it exists. If not, we store `undefined` to mark the absence of a value. We do not choose `NULL`, because that would be a valid SQL entry (Recall Figure 1.1). This can be useful because it allows us to filter rows with non-existing values, too, which might be the interesting ones of all.
Naturally, a value should not occur in a list multiple times. We define a function that removes duplicates from a list.[6]

4.3.5 Deleting Scopes

In a JT, every subtable occurs within a scope (2.3.2).

For the user, scopes are implicit apparent through nested subtables. Besides that, he is not interested in their names that PGHabitat created artificially.

[5] We will come back on that in Section 4.4.3.
[6] ...and call it `nub` as an homage to Haskell.

One step of JT processing is to delete single scopes (`jtDeleteSingleScopes`): If there is a JT with only one row object that only contains one single scope attribute and nothing important besides (like other scopes or observations), we replace it with the value of the attribute. For instance the object

```
1 [{"xScope.rid.0.0":[{"xScope.rid.1.1":[<object>],
2    "exists:xScope.rid.1.1":true}],"exists:xScope.rid.0.0":true}]
```

only becomes

```
1 [<object>]
```

The deletion of single scopes only is performed if the scope is existent.

Note that the outer row object is the only element of the array. We could not have done this if there were more, because that would result in nested arrays. Since the elements of an array are considered as row objects, columns with integer JTANs would occur.
In difference to that, the innermost array could contain multiple rows. It still would be a valid JT.

In reality, these prerequisites are only fulfilled for the outer scopes of a $^{\mathrm{PG}}\textsc{Habitat}$ JT.

To call `jtDeleteSingleScopes` used to be the first thing to do when preparing `jtGlobal`. The goal was to make the JT smaller, nested shallower and to make URIDs shorter. It turned out to be a bad idea for the following reason: The URID of a row may not be persistent anymore, if we consult $^{\mathrm{PG}}\textsc{Habitat}$ again. It is possible that a new scope or observation occurs and a scope is not thrown away that was thrown away before. Rows marked as interesting would not be reliable (2.2, persistence of interesting rows). Now this function call is part of the rendering function. That leads us to the next section.

4.4 The Observation Table

The observation table is the central part of the debugger. Let us have a look on how it is rendered and equipped with functionality.

4.4.1 Rendering the Table

The observation table is created completely via DOM manipulation. We use D3 to simply append nodes. Our Debugger Page has a *div* container at its disposal called **OTA** ("observation table anchor") where we start from.

The function `jtGlobalRender` is responsible for starting the rendering with `jtGlobal` and acts as a wrapper of the main rendering function `jtRender`. It deletes child nodes from former tables or error messages. `jtRender` itself is capable of creating observation tables at any given place in the document and could be used to only render parts of a table again.

The part of JT processing that belongs to rendering is done now. Only the deletion of single scopes is performed (4.3.5), while other steps are deprecated.

If row values should not be displayed (Figure 2.10), one possibility is to delete them now via `jtDeleteAttr`. The more clever idea to translate this into practice will be explained in the next section.

We also used to call `jtDeleteIgnoredRows` (4.3.3) here in order to delete rows that do not pass the filter. But that led to errors when we wanted to render an empty table. Expressed as a JT, an empty table has no schema. Instead, we dismiss this function for this purpose and check if a row should be displayed just before appending it to the document.
Similarly, we do not edit non-existing observations and scopes (`jtEditNonExisting`) by processing the JT. Rather, we check whether we need to display a special label directly before rendering it. This way, meta-information is not merged with the SQL data as long as possible.

Because table headers are lifted (2.3.1), the rendering of table head and table body can be completely separated. Both `jtRenderHead` and `jtRenderBody` are recursive and invoked from non-recursive `jtRender`, operating on the same joined table data.

Rendering the Table Head

The table head is rendered within a *thead* element as a child of the given table selection.

As we know, tables can be nested deeply. If there is a scope with a subtable, we need to recurse and render its own header. But where to append it? We still want to visualize the nesting and not render a single header row. The solution is the following: There are two rows for every nesting level of JTs: a **headrow** and a **subheadrow**. In the headrow the columns of a JT like row values and observations can be found. If there is a scope with a nested subtable, we display a special string (the empty string) in the header row. The subheadrow contains just as many *th* elements as the headrow and - in case of a nested subtable - a new table element within as an anchor for the recursion. Thus, we achieved that the table head has just as many rows as the depth of the JT. The last subheadrow has no content and therefore is not visible automatically.

If we are about to render the table head for a subtable, we need to know its schema. On recursive function call, the choice of the row is important, because there may be rows for which this scope does not exist and these rows would not be able to offer the schema of the subtable. A helper function `jtArbitraryScopeContent` returns the subtable of a given scope name of a row, where this scopes exists. We can rely on the fact that there is at least one row with this property, because otherwise the scope would not make any sense.
Helpers for a function are a good idea in general to exhibit a clear structure. We define some for calculating the attributes of table header rows and their cells like CSS classes, colors and labels that are actually displayed. Having said this, it is more comprehensible that we should distinguish between JTANs and column names.

Remember that we did not delete attributes, which should not be displayed, yet. This affects the `exists` attribute and the URID, as well as row values. The clever thing is: we do not need to. The rendering works with a helper function that distills all

column names from a JT that should be rendered and returns them as an array. For every entry in this array, we look up the value in all row objects. Columns that should not be displayed are simply deleted from this array. Because arrays do have positions - unlike objects - we can also sort these column names - and therefore the display order of the columns - ad lib. A sorting function compares two column names according to the order defined in a variable in `config.js`, so that this can be changed easily.

In order to decide which column a *th* cell belongs to later on, we attach an attribute to the node. The *id* attribute is not appropriate here since an id really should be unique. In HTML5, any element can carry custom data with the *data-** attributes [W3C, 3.2.3.8]. We can make good use of that as we attach *data-ucid*.

Rendering the Table Body

The table body is rendered as a *tbody* element similarly to the table head. Instead of the keys, now the values of row objects are made visible. We iterate over the array of row objects and append a *tr* element to the body if

- it has no URID, because these kind of rows should be displayed all the time, or
- if its URID is marked as interesting or
- if its URID is not marked as ignored.

So, interesting rows are displayed even when they are ignored, which is a feature, not a bug.

Every SQL row represented in a *tr* element gets an HTML *id* attribute with the URID.[7] Just like in header cells, the data cells carry their UCIDs in *data-ucid*.

The content of a cell depends on the type of the JTAN: If we have an existing scope, we need to append a new subtable and recurse. Otherwise, we just call a helper that pretty-prints records, arrays, strings, numbers and booleans. We also use CSS to style the background color of boolean or non-existing values.

[7] This time, they are unique.

The function does not only append a *tbody* element to the given table. It also adds a *span* element as a table replacement. This element is shown when the table is collapsed. We make the label for this replacement dependent on the number of rows we appended to the body and distinguish between no rows, one row or multiple rows.

The fact that we rendered head and body separately leads to an annoying bug. If nothing else is specified, each table cell adopts the width it needs. In vanilla HTML tables, the browser of course takes care of the widths in every row and renders a perfectly aligned table. However, this is not done for us now because nested subtables in the head do not have anything to do with the accompanying subtables in the body. In case of varying lengths of their contents, header cells may not be above the right column of the body.
Luckily, D3 offers the opportunity to get and set the width of DOM objects. Even if the width is set in percentage, absolute values in pixels are returned. We work around the problem by figuring out the maximum width of each column and adjusting every relevant cell.

Now that we rendered the table, it is time to make it interactive.

4.4.2 Implementing Interesting Rows

Every row that has a URID is a SQL data row and should be markable as interesting. Interesting rows differ from non-interesting rows in their background color and visibility of subtables, as explained in Chapter 2.

The interestingness of rows should be persistent when PG HABITAT is consulted again (2.2). This is why we store URIDs of interesting rows in a global array that is encapsulated from the view.

But how do we identify a row when the user clicks on it in the browser to toggle its interestingness? Now we profit from the *id* attribute of the row, because D3 can fetch it. So, we can extrapolate the URID and make state changes that do not only affect the view.

When a row is marked as interesting or uninteresting, firstly, this state is saved. Secondly, the view is updated: We change the CSS class to manipulate the background color of the row. Also, we show subtables of an interesting row and hide subtables of an uninteresting row, respectively. What sounds easy caused some problems: We only want to select every direct subtable. A simple D3 subselection of a given row selection also selects subtables of subtables and their descenders.
A possible solution is to use a global selector, which allows us to subselect direct child nodes. It contains the unique URID of the row. Furthermore, the URID can not be used unmodified, since D3 interprets some of its special characters as enumerations (`,`) and classes (`.`), et cetera. To fix this, we need to escape these characters.

The effects on mouseover, mouseout and click are hit by event handlers. But it should be possible to turn the whole feature of interesting rows off. For this reason, we do not incorporate it into the rendering function, but instead define a function `enableInterestingRowFunctionality`.

With D3, we select every *tr* element within the table and filter just the ones that should have this functionality - the ones with valid URIDs. Then we register the event handlers.
As a result, another problem occurred. Consider an interesting row r_{outer} with a visible subtable. Say, the user wants to mark a row r_{inner} within the subtable as interesting with a click. Now two rows are toggled: r_{inner} and the r_{outer}, because both event handlers do their job. The user does not see the subtable anymore, because it was collapsed.
Once found, the solution is easy: we need to stop the propagation of the event in the D3 event global [Bos, d3.event].

Additionally, to the interactivity of rows, we do register event handlers for the header cells of scopes if they are in CSS class `invertableColumn`.[8] On mouse click, the interestingness of every accompanying row is toggled. In the implementation it pays off that we can easily check via `matchesUCID` if the URID of a row matches the UCID of the cell the user clicked on.

[8] The rendering function took care of that.

The functions to show or hide a table basically swap the *display* attribute of a table body and a table replacement.

As a third step after defining all these events, subtables of non-interesting rows are collapsed by default, except they only contain one row, as we explained in Section 2.3.1.

4.4.3 Implementing Row Filter

A row is said to be **ignored** if it should not be part of the observation table. This is currently the case if it is filtered out via `rowFilterWith`, but could have other causes in the future. As already mentioned, the interestingness of rows has a higher importance. While subtables of uninteresting rows are not displayed but still available as a DOM node, for ignored rows we follow a different approach: They will not be rendered at all. In many cases, this makes the table significantly smaller.

`rowFilterWith` is a function that accepts a UCID and a value. It traverses `jtGlobal` and checks if a row has a URID that matches the given UCID. If so, the row is marked as ignored in case the specified column has another value than the given one. If the given value is undefined, we interpret it as a special case: Then only rows with non-existing values are kept.
Just as for interesting rows, we maintain a global array with URIDs of ignored rows and do not actually throw data away.

In order to implement the interface for the row filter functionality (`enableRowFilterFunctionality`), we append an HTML *select* element to the table head cells that will be displayed on mouse click. The individual options are created with the help of `ucidValuemap`.
We assume two things: Firstly, `jtRenderHead` should have made the *th* elements member of the `filterableColumn` CSS class. Secondly, `ucidValuemap` should have been initialized and filled at this time, which is a step of JT processing (4.3.4)).

The HTML *option* element can be provided with an *onclick* attribute. Its value should be JavaScript code. We fill it with a function call of `rowFilterWith` to compute

the filtering and `jtGlobalRender` to render the new table.

The fact that we append textual code to the document should make us ask ourselves a few questions: What will happen to values of different types? What if a value contains special characters?
Actually, `rowFilterWith` does string based comparisons for this reason. We define helper functions that convert values to strings[9] and compare two values in their string representation. This approach should not cause complications because one SQL column can not contain values of multiple types.
All UCIDs and values within the appended code are encoded via `encodeURIComponent` because they should not be interpreted as JavaScript under any circumstances. The decoding is part of the code, immediately.

4.5 Additional Pages

4.5.1 Implementing the Query Management Page

The fact that query management is done on another page (2.6) is already reason enough to save queries on the server side in `$_SESSION`.

The Query Management Page consists of several CodeMirror instances, containing

- the current query,
- pinned queries,
- recently used queries from the history.

The query history is shown without the recent query, which would not be of any use.

Because this is not the place where queries should be edited, we modify the CodeMirror configuration object by setting `readOnly` [CMA] to `true`, so that the content can not be changed.

[9] This also includes `undefined` and `null`. Both result in a type error if `toString` is called on them.

To pin and and unpin queries or to use a query as the current one, there are buttons below each editor instance. We summarize them in an HTML *form* for a good reason: On click, a JavaScript function is called. It needs to know which query we are talking about. All of these JavaScript functions request a PHP script on the server in order to update the `$_SESSION` variable. The query is taken from the *form* and submitted via HTTP POST.

The PHP scripts are all similar and plain. They do not only update the state on the server, but also deliver the latest version of the Query Management Page. Because the exception proves the rule, the script to set the current query delivers the Debugger Page. Why should a user click this if he does not want to get started?

4.5.2 Implementing the Setup Page

The Setup Page is designed to configure the PGHABITAT frontend.

It basically contains one HTML *form* that links to itself via `action="#"`. When we request it, the `$_SESSION` variable is updated from `$_POST`.[10] The script is divided into two sections: updating variables and creating a document.

Our state in `$_SESSION` is only updated when `$_POST` is set. Otherwise we would possibly delete settings every time the user wants to have a look at his setup.

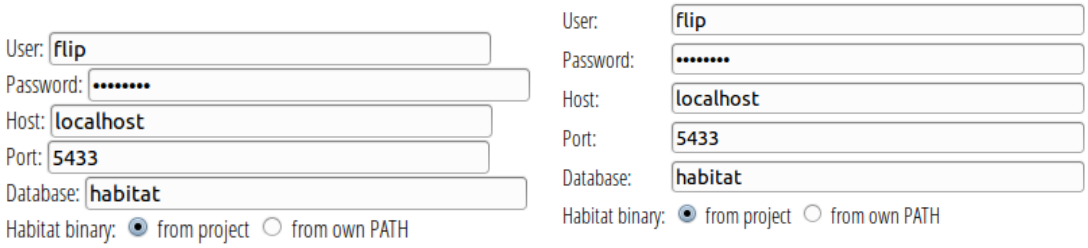
To set boolean values like `usePATHBinary` we compare the value with constant `"true"` via == instead of fetching the value directly. This is nice because it will always return a boolean value, no matter if we use another type to set it. Just like JavaScript, PHP comes with the non-strict comparison operator == that considers `true` (type boolean) and `"true"` (type string) as equal [PHP, Operators - Comparison Operators]. For type-strict comparisons there is the === operator.

Furthermore, we set default values if we do not have any yet, which should be the case on the first request of the page.

Last but not least, we want to note a web development trick that is old but gold

[10] Just like we did in the `habitat.php` script. No surprise here.

Figure 4.7: Alignment of Forms



(a) Crude (b) Aligned

and deserves mentioning: To get a nice uniform layout, the input fields are put inside an invisible HTML table. This way, we do not need explicit spaces or CSS rules and everything is perfectly aligned (Figure 4.7). Although the usage of HTML tables for layouting is notorious these days, it suits this situation fine.

5 Conclusion

PGHabitat can be a helpful tool, whether it is for debugging or for the illustration of SQL behavior. Unfortunately, the work could not be appreciated adequately so far. The interface simply was too ponderous.

The frontend finally makes PGHabitat usable. Now the software can take full effect.

A good field of application may be university education. Students will have an easier time following the lecture if learning content is visualized. Especially the meaning of abstract, textual SQL queries can be much more comprehensible when they are related to concrete values from a database state.

However, PGHabitat as well as its frontend are not ready to be released. Both can and should be developed further.

Future work can concern the following aspects:

- The observation table could be adjusted for special cases. For instance, the "show row values" feature might not be sufficient for tables with many columns. Giving the user the opportunity to select particular columns could be helpful.
- Until now, every selected substring of a query can become a marking (4.2.1). Using the mouse imprecisely results in invalid marking positions easily. It would be a great feature if selections that are slightly off would be corrected automatically.

Still, the project shows that it is possible to observe intermediate step results in declarative languages just as good as in imperative ones, or maybe even better.

List of Abbreviations

AJAJ	Asynchronous JavaScript and JSON
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CMS	Content Management System
DBMS	Database Management System
DDL	Data Definition Language
DML	Data Manipulation Language
DOM	Document Object Model
GID	Group Identifier
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocoll
JSON	JavaScript Object Notation
JT	JSON Table
JTAN	JSON Table Attribute Name
OTA	Observation Table Anchor
PHP	PHP Hypertext Preprocessor
RDBMS	Relational Database Management System
RID	Row Identifier
UCID	Unique Column Identifier
URI	Uniform Resource Identifier
URID	Unique Row Identifier
XML	Extensible Markup Language

List of Figures

Bibliography

[APA] Apache HTTP Server Version 2.2 Documentation. `http://httpd.apache.org/docs/2.2/en/`. [Online; accessed 21-September-2014].

[Bos] Bostock, Mike. D3.js API Reference. `https://github.com/mbostock/d3/wiki/API-Reference`. [Online; accessed 21-September-2014].

[CMA] CodeMirror API Reference. `http://codemirror.net/doc/manual.html#api`. [Online; accessed 21-September-2014].

[Die14] Dietrich, Benjamin. Rebooting Habitat on PostgreSQL - A Declarative Observational Query Debugger, January 2014.

[ECM] ECMAScript Language Specification - ECMA-262 Edition 5.1. `http://www.ecma-international.org/ecma-262/5.1/`. [Online; accessed 21-September-2014].

[GKRS11] Torsten Grust, Fabian Kliebhan, Jan Rittinger, and Tom Schreiber. True Language-level SQL Debugging. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 562–565, New York, NY, USA, 2011. ACM.

[GR13] Torsten Grust and Jan Rittinger. Observing SQL Queries in their Natural Habitat. ACM, 2013.

[GW87] Richard A. Ganski and Harry K. T. Wong. Optimization of Nested SQL Queries Revisited. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 23–33, New York,

NY, USA, 1987. ACM.

[Kim82] Won Kim. On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, September 1982.

[Kor86] Korth, Henry F. and Silberschatz, Abraham. *Database System Concepts*. McGraw-Hill Book Company, 1986.

[PHP] PHP Documentation. `https://php.net/manual/en/`. [Online; accessed 21-September-2014].

[Pog53] Poggendorff, J. C. *Zur Theorie der Farbenmischung - Poggendorff's Annalen de Physik und Chemie, Bd. 89*. 1853.

[POS] PostgreSQL Documentation 9.3. `http://www.postgresql.org/docs/9.3/static/`. [Online; accessed 21-September-2014].

[RFCa] RFC 2183 - Communicating Presentation Information in Internet Messages: The Content-Disposition Header Field. `http://www.ietf.org/rfc/rfc2183.txt`. [Online; accessed 21-September-2014].

[RFCb] RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1. `https://www.ietf.org/rfc/rfc2616.txt`. [Online; accessed 21-September-2014].

[SG] T. Smith and J Guild. The C.I.E. Colorimetric Standards and their Use. `http://iopscience.iop.org/1475-4878/33/3/301/`. [Online; accessed 21-September-2014].

[W3C] HTML5 - A vocabulary and associated APIs for HTML and XHTML. `http://www.w3.org/TR/2011/WD-html5-20110525/spec.html#contents`. [Online; accessed 21-September-2014].

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