Eberhard Karls Universität Tübingen

Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor Thesis in
Computer Science in Media

Implementation of a Browser-based Interface for Provenance Analysis

Janek Bettinger

2014/30/09

Reviewer

Prof. Dr. Torsten Grust

Database Systems Research Group
University of Tübingen

Advisor

Tobias Müller

Database Systems Research Group
University of Tübingen

Implementation of a Browser-based Interface for Provenance Analysis

Janek Bettinger
Bachelor Thesis in Computer Science in Media
(Bachelorarbeit in Medieninformatik)
Eberhard Karls Universität Tübingen
Period: 2014/05/30 – 2014/09/30

Abstract

Data provenance is a research topic dealing with the origins of generated data. An already existing tool by the Database Research Group of the University of Tübingen computes provenance relations of Python programs and outputs its results in the relational data format.
The primary purpose of this thesis is the development of an interactive browser-based interface that makes the analysis results human-accessible.

Contents

Chapter 1

Introduction

1.1 Provenance Analysis

Ultimately, most computer programs process data. They consume input values, operate on them and output a result. As return values depend on the input values, they are related to each other. Finding the values affecting another one is generalized as provenance analysis that is about deriving the origins of generated data.

The Database Systems Research Group at the University of Tübingen has developed a tool for provenance analysis concerning Python programs. It able to compute the provenance of incoming and outgoing value of functions.

1.2 Aims

The results of the provenance analysis, however, only exist in form of raw data in a relational database and not comprehensible for humans in an easy and intuitive manner. This work devotes with the development of an interface for visualizing the processed data together with the provenance relationships within. It aims to simplify the comprehension of the underlying program like in order to reveal sources of error.

1.3 Interface

The interface is a single-page web application that is backed by a server-sided component for retrieving the results of the provenance analysis. As illustrated in figure 1.1, each function of the analyzed Python program is listed together with all of its input and output parameters. Four different visualization respectively chart types allow the user to view the parameters' values in a proper representation:

- *Table* — displays the values without further processing
- *Scatter Plot* — maps numeric value pairs to a two-dimensional coordinate system
- *Histogram* — represents the distribution of numeric values
- *Line Mosaic Plot* — allows to recognize relationships between different categorical variables

By selecting one or more values by a simple mouse click all related values get highlighted in all other charts illustrating both provenances and impacts.

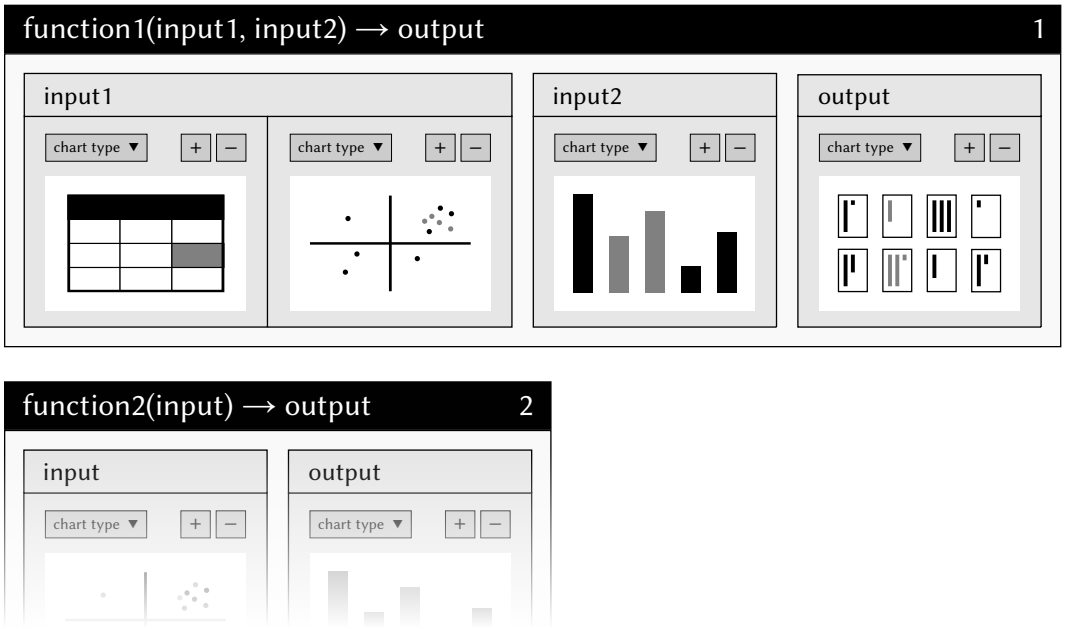

Figure 1.1: Scheme of the overall interface including (from left) adumbrated table, scatter plot, histogram and line mosaic plot whereby gray chart elements indicate highlighting

1.4 Document Structure

While this first chapter has given a short overview of the topic, the subsequent chapter 2 introduces provenance analysis and the back-end tool, as well as fundamental techniques and resources required for development and implementation. Latter is described in detail in chapter 3, before the last chapter provides a conclusion and outlook.

Chapter 2

Fundamentals

Data provenance forms the basis of the analysis tool whose results get visualized by the interface developed as part of this thesis. The theoretical aspects of data provenance and the functional principles of the analysis tool are introduced hereinafter. Furthermore, essential programming and markup languages as well as file formats and third-party tools that have been used for the implementation are expounded in this chapter. Server-sided techniques are used to access and deliver the analysis results, while client-sided ones are required for the concrete interface including charts. Ultimately addressed development tools aim to improve the workflow and the quality of source code.

2.1 Data Provenance

The field of data provenance research deals with the derivation history of generated data. In the context of database systems [BKT01] defines it as the description of the origins of data and the process by which it arrived in a database. Provenance analysis helps understanding data sets and improves debugging capabilities. However, those ideas are not limited to databases and can be transfered to other fields, like the basis of this work is data provenance concerning Python programs.

2.1.1 Program Slicing

In a manner analogous to data provenance is a technique called program slicing. First described by [Wei81] and further developed by [KL88], it reduces a program to a subset of statements that affect the value of a certain variable. So program slicing explains relationships using parts of the program, while data provenance uses parts of the dataset for an explanation.

2.2 Provenance Analysis Tool (PAT)

The Provenance Analysis Tool calculates bidirectional relationships between incoming and outgoing values of functions in Python programs. While it describes the data provenance, it makes use of program slicing techniques without revealing concrete statements in the result. However, the particular function call a value belongs to is acknowledged.

Currently, only a subset of the Python language is supported. Among other things, programs to analyze may not contain statements that perform multiple kinds of operations at once.

The tool is written in Python and stores its results in a PostgreSQL database. Both Python and PostgreSQL are addressed in section 2.3.

2.2.1 Sample Program

For a clearer understanding of PAT's functional principle a very basic example is given hereinafter and extended subsequently to illustrate more of its functionality.

```
def sum(arg_data):
  res = []
  for d in arg_data:
    a = d['a']
    b = d['b']
    sum = a + b
    r = {}
    r['v'] = sum
    res.append(r) # r == {'v': d['a'] + d['b']}
  return res

data = [{'a': 1, 'b': 2, 'c': 3},]
summed = sum(data) # [{'v': 3}]
```

Listing 2.1: Sample Python program that calculates sums

Listing 2.1 shows a simple program with one function `sum()` that takes and returns a list of dictionaries. For each dictionary the sum of the items with key `a` and `b` is calculated and stored in a new dictionary as item `v`. In the example, the given list contains only one dictionary with three items `a`, `b` and `c`. Obviously, the function's return value is `[{'v': 3}]`.

PAT logs the invocation of `sum()` including every value of the sole input parameter `data_in` and the output parameter `res`. However, the decisive computation by the analysis tool is the provenance of the resulting `{'v': 3}` that is `{'a': 1}` and `{'b': 2}`, but not the unused `{'c': 3}`.

The sample program gets extended in listing 2.2 in order to demonstrate function composition that is passing the result of a function as the argument of the next. In this example, the already established `sum()` and a new function `square()` are composed. Latter works similar to the former one and calculates the square of an item with key `'v'`.

Now the value `[{'v': 3}]` is used in two different contexts. On the one hand as the returned value of `sum()` and on the other hand as incoming argument value of `square()`. What matters is that PAT recognizes the composition and that it stores a link between the usage as outgoing and incoming value.

```
# ...
# summed == [{'v': 3}]

def square(arg_data):
  res = []
  for d in arg_data:
    v = d['v']
    square = v * v
    r = {}
    r['v'] = square
    res.append(r) # r == {'v': d['v'] * d['v']}
  return res

squared = square(summed) # [{'v': 9}]
```

Listing 2.2: Extension of the prior program

2.2.2 Database Structure

PAT writes its results into several database tables whereby many are linked to each other by foreign keys. Those needed for interpretation and the creation of the visualization are illustrated in figure 2.1 and further described hereinafter.

`functioncalls` Logs all, but only those functions that are called in the analyzed program in the correct order. For each call, the table contains among other details the `functionname` and an unique identifier `id`.

`argumentvalues` and `returnvalues` For each call in `functioncalls`, those tables contain all incoming respectively outgoing values whereby the particular call is referenced by a `callid`. While the tables are in first normal form as all data is atomic, they represent non-atomic nested structures called containers. Each list, dictionary, but also each of its items introduces a further deeper level respective container down to atomic values such as numbers or strings. The column `containerid` each references the superior container and is `NULL` at the highest level

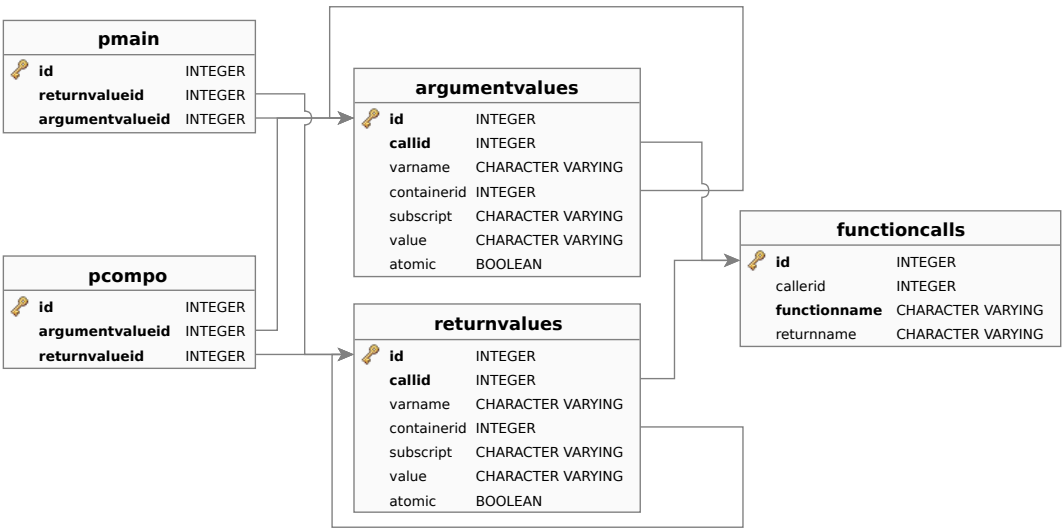

Figure 2.1: Diagram of the database tables holding the results of the provenance analysis by PAT where arrows indicate foreign-key relationships and non-`NULL`able columns are printed in bold

where, however, the column `varname` holding the parameter name is populated. Moreover, the column `subscript` contains the index if the value represents an item of an array or the key if is an item of a dictionary. If the column `atomic` indicates `TRUE`, a value can be found in the `value` column.

The following two tables describe the crucial part of the analysis, namely the data provenance.

`pmain` Contains the provenance of return values concerning the respective function's argument values. The relationships between incoming and outgoing values are each established by an `argumentvalueid` and `returnvalueid`. The referenced values do not necessarily have to be atomic. Relations between containers and atomic values may exist, too.

`pcompo` Constructed like `pmain`, this table contains the provenance of argument values in case of function composition. So it describes relationships between outgoing and incoming values of two functions when the result of the first one is passed as the argument of the next one.

Figure 2.2 illustrates the content of the two provenance tables on the basis of the previous sample code in listings 2.1 (`sum()`) and 2.2 (`square(sum())`).

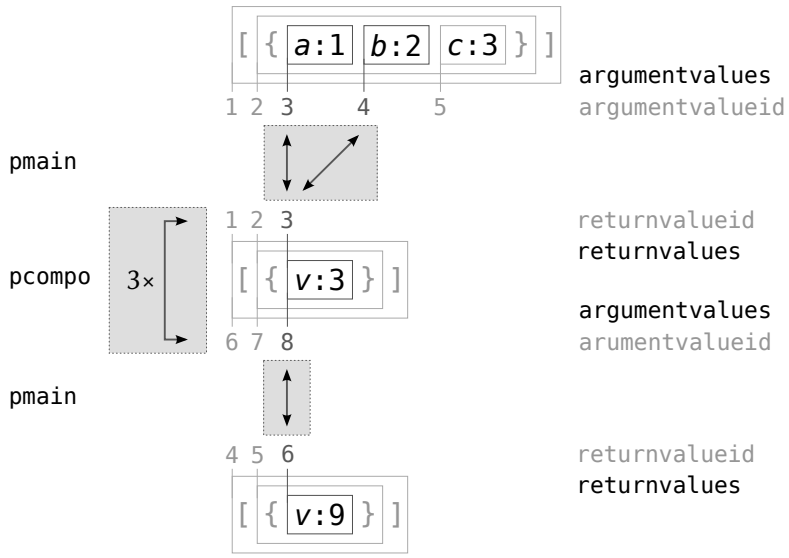

Figure 2.2: Abstract illustration of the value tables as well as the provenance tables `pmain` and `pcompo` containing the results of the analysis of the previous sample program (listing 2.2). Gray boxes indicate containers, black values are atomic ones and the arrows illustrate the provenance relationships.

2.3 Languages and Formats

While Python and JavaScript are the main programming languages on server respectively client side, several further languages or file formats are required for the overall implementation of the graphical interface. All of them are outlined below.

2.3.1 Server-side

Python

Developed in the early 1990s and constantly enhanced, Python[1] is an interpreted programming language that supports diverse programming paradigms. Similar to Java or C#, Python has a compiler that translates the source code into byte code that is executed by a virtual machine, the interpreter. It is cross-platform, so Python programs usually run without modification on every operating system that is supported by a Python interpreter. That includes all present-day major systems.

As the syntax of Python is very simple, no further details are given in this work. Sample programs are merely listed in the preceding section 2.2.1.

Python is used in a variety of ways in this work. While Python programs get analyzed, both PAT and the interface's server component are written in this language.

[1] `https://www.python.org/`

PostgreSQL

PostgreSQL[2] is an advanced open-source database system running on all major operating systems. As object-relational database management system (ORDBMS) it is based on the relational data model where data is organized in relations as sets of tuples of data. Besides, it implements an object-oriented model where the structure and queries support classes, objects and inheritance. However, latter aspect is immaterial for this work.

A PostgreSQL database holds the results of PAT, while PostgreSQL queries are used to retrieve and manipulate the data.

2.3.2 Communication Layer

JavaScript Object Notation (JSON)

JSON[3] is an open standard format derived by the JavaScript language. Nevertheless, it is language-independent as there are exist generators and parsers in a large number of different programming languages. The grammar specifying a valid JSON document is simple and intuitive. According to the proposed standard [Int14], a JSON value has to be either a string, number, array, boolean, `null` or an object that wraps name–value pairs. Nesting is possible and common due to arrays and objects that again may contain further atomic or non-atomic values.

Listing 2.3 shows a sample JSON document demonstrating the different value types. In this work JSON is used to exchange data between client and server.

```
{
  "key / name": "value",
  "key of an array": [
    "a string", "", 1, true, false, null,
    [ -0.7E10, 42 ],
    { "foo": "bar" }
  ]
}
```

Listing 2.3: Sample JSON document demonstrating different value types

2.3.3 Client-side

JavaScript

JavaScript is probably the most important programming language for client-side web development that even can be used on server side in the meantime. It is a

[2] http://www.postgresql.org/
[3] http://json.org/

dynamic-typed and interpreted scripting language.

In this work, JavaScript makes up the largest part as it is used to generate the visualizations on the client side. Moreover, server-sided JavaScript is used to generate the final code for release.

In the following, some more language details are given.

Objects Objects are JavaScript's fundamental data type. An object is an unordered collection of key–value pairs, called properties, whereby the key has to be a string (or an object with a `toString()` method). In contrast to many other programming languages, objects are dynamic, so properties can usually be added and deleted at runtime. Any value that is not a primitive one (string, number, boolean, `null` or `undefined`) is an object, so even a functions is one. A property value that is a function is called method.

Prototype-based inheritance In JavaScript there are no classes as known from object-oriented languages like Java or C++, although this term is used in many places. In prototype-based languages as JavaScript, objects may be associated as the prototype for another object allowing latter to share the fist object's attributes. By calling a constructor function with the `new` operator a new object is created, whereby the constructor's `prototype` property is used as the prototype of the new object. So all objects created by the same constructor function inherit from the same prototype object. Therefore, they are called members or instances of the same class. The constructor may be *any* function, but it usually is one that does initialization such as setting property values.
In some cases using `Object.create()`, specified in ECMAScript 5 in 2009, is more appropriate as the prototype's constructor function is not called, in contrast to a construction using `new` .

Listing 2.4 demonstrates inheritance with an `Animal` base class and two sub classes `WildAnimal` and `DomesticAnimal` whereby the prototype chain is established using two different ways. Using `new` is the traditional variant and makes it harder to pass arguments to the prototype, while `Object.create()` is a newish approach that is used in this work.

HyperText Markup Language (HTML)

HTML[4] is the common language for describing the content of web pages and usually interpreted by web browsers. The specification [WW14] states that a HTML document has to consist of a tree of elements and text, where each element is denoted by a start tag such as `<p>` and usually closed by an end tag as `</p>`. Tags have to be nested without overlapping and can have attributes as the `href` attribute in line 8 of listing 2.5.

[4] `http://www.w3.org/html/`

```
// Superclass / base class
function Animal(name) {                              constructor function of superclass
  this.name = name || 'yet another animal';
}
Animal.prototype.introduce = function() {            define a method of the prototype
  console.log('Hi, I\'m ' + this.name + '.');
};

// Subclasses / derived classes
function WildAnimal() {}                             empty constructor function of subclass
WildAnimal.prototype = new Animal();                 call Animal() & inherit its prototype
WildAnimal.prototype.constructor = WildAnimal;       override inherited constructor property

function DomesticAnimal(name) {
  Animal.call(this, name);                           execute Animal(name) with given this
}
DomesticAnimal.prototype = Object.create(Animal.prototype);   inherit prototype
DomesticAnimal.prototype.constructor = DomesticAnimal;
DomesticAnimal.prototype.introduce = function() { shadow prototype property
  console.log('Hi, I\'m ' + this.name + ' and I live together with humans.');
};

// Object creation and usage
var wild = new WildAnimal();                         create WildAnimal object
var garfield = new DomesticAnimal('Garfield');       create DomesticAnimal object

wild.introduce();     // Hi, I'm yet another animal.          call method of object
garfield.introduce(); // Hi, I'm Garfield and I live together with humans.
```

Listing 2.4: Sample JavaScript programm demonstrating inheritance

A HTML document starts with a document type declaration that defines the legal document structure as well as allowed elements and attributes. The subsequent `html` root element contains two elements: `head` and `body`. While the content of the `head` element is not directly visible as it usually contains meta data and references to scripts and style sheets, the `body` represents the actual content of the document.

In this implementation the future HTML5 standard, that is already mostly supported by all modern web browsers, is used.

Document Object Model (DOM) The DOM is a specification of an interface for accessing and manipulating the content, structure and style of XML-based documents such as HTML documents. The DOM represents documents in a hierarchical tree-like structure whereby changes to the tree are incorporated back into the presented document. "DOM events" allows to register event handlers on DOM nodes that are triggered once an event (such as a mouse action) occurs. The DOM tree can be inspected using the developer tools of modern web browser.

```
<!DOCTYPE html>                                          doctype
<html>
  <head>                                                 head containing meta data
    <title>Sample page</title>
  </head>
  <body>                                                 body containing visible content
    <h1 id="title">Sample page</h1>
    <p class="info">This is a <a href="target.html">simple</a> sample.</p>
    <p class="info highlight">Second info paragraph.</p>
  </body>
</html>
```

Listing 2.5: Sample HTML5 document

Cascading Style Sheets (CSS)

While HTML is required for denoting the content of web pages, CSS describes their look and formating. As specified in [al11a], it is not only possible to style HTML documents using CSS, but also other XML documents as SVG. A CSS file consists of a list of statements whereby the most important statement is a rule set that is usually simply called "rule". In turn, a rule consists of at least one selector followed by a list of declarations wrapped by curly brackets. A selector is a patterns to match a set of elements in the HTML document tree and may be an unique identifier, an element name, a class name or even a complex contextual pattern.

Listing 2.6 shows an exemplary style sheet using different types of selectors. Together with the HTML document of listing 2.5 it results in a web page with gray background, a large headline surrounded by a red border that is followed by two underlined paragraphs where the first one is black and the second red.

For the implementation of the provenance visualization some features of the upcoming standard CSS3[5] are used whose most important features are already supported by modern web browsers. CSS3 provides among other things extended selectors and nice-looking transitions.

Scalable Vector Graphics (SVG)

According to the specification [al11b], SVG[6] is a markup "language for describing two-dimensional graphics in XML". SVG graphics can easily be embedded into HTML documents as both include a DOM and can optionally be styled using CSS. SVG facilitates three types of graphic objects: vector graphic shapes based on primitives (rectangles, circles, ellipses, straight lines, polylines, polygons), raster images and text. Those objects can be transformed to obtain more complex shapes, furthermore simple animations are supported.

[5] http://www.w3.org/Style/CSS/current-work

[6] http://www.w3.org/Graphics/SVG/

```
body {                                          type selector
  background: #aaa;
}

#title {                                        ID selector
  font-size: 2em;
  border: 2px solid red;
}

.info {                                         class selector
  text-decoration: underline;
}

.highlight {                                    class selector
  color: #ff0000;
}

h1:first-child + p {                            type, pseudo-class and adjacent selector
  font-weight: bold;
}
```

Listing 2.6: Sample style sheet for the HTML document of listing 2.5

An advantage of vector graphics over pixel-based raster formats such as JPEG or PNG is the ability of lossless resizing as they only store a description of objects, but no concrete pixel values.

Listing 2.7 shows a HTML document with an embedded SVG that describes a red circle with a 3 Pixel wide black border.

```
<!DOCTYPE html>
<html> [...]
  <body>
    <svg height="50" width="50">
      <circle cx="25" cy="25" r="20"
          stroke="black" stroke-width="3" fill="red" />
    </svg>
  </body>
</html>
```

Listing 2.7: Sample HTML document with inline SVG

Distinction to Canvas HTML5 introduced another technique for generating graphics in web browsers: the `canvas` element (specification [WW14] chapter 4.11.4). However, Canvas graphics do not scale as well as SVG graphics and do not support animations and mouse hovering effects without complexity (see [Mic14]). According to [Mic14], Canvas is better for complex real-time or high performance

scenarios, while SVG is more appropriate for (more or less) static high fidelity images. Nevertheless, almost any two-dimensional graphic can be drawn by using either of both technologies. For the provenance visualization the decision came to SVG as it seemed to be more pleasing for generating nice charts without large effort.

2.4 Tools, Libraries and Frameworks

For both server and client it was made use of several libraries and frameworks. While selecting it has been paid attention that these tools only cause a minimum amount of overhead and that they are likely to be further developed.

2.4.1 Server-side

Bottle.py

Bottle.py[7] is a lightweight web-framework for Python. It is used to deliver the web page and to build an Application Programming Interface (API) that allows clients to fetch results of the provenance analysis. The utilized build-in development server is sufficient for non-public servers, however, any server written to the Web Server Gateway Interface (WSGI) specification can be used as server backend. WSGI describes the communication between server and application, but it is not of importance for this work.

Psycopg

Psycopg[8] is a PostgreSQL database adapter for Python and is used for the server application. An existing PostgreSQL plugin[9] for Bottle.py making use of Psycopg did not work with object-oriented programming style and has been discarded for this reason.

2.4.2 Client-side

RequireJS

JavaScript lacks a statement as `import`, `include` or `require` to load required source code from other files. This may pose problems if the amount of source files is large as the order script files are loaded (on a web page) matters. Also the common

[7] `http://bottlepy.org/`
[8] `http://initd.org/psycopg/`
[9] `https://github.com/raisoblast/bottle-pgsql`

approach of concatenating all script files at the end of a development circle does not eliminate this difficulty.

RequireJS[10] is a module loader implementing the Asynchronous Module Definition (AMD). An AMD module is defined by an optional identifier, an optional list of dependencies and a factory function that instantiates the module or object. All this resolves the difficulties mentioned above. Furthermore, RequireJS includes an optimization tool for combining and minifying all scripts to reduce the number of required server requests and to decrease the file sizes.

jQuery

jQuery[11] is a popular JavaScript library to simplify among other things DOM manipulation, event handling and asynchronous data transfer. It is used throughout the implementation where performance is not critical and native functions are inconvenient.

jQuery UI A[12] set of user interface utilities built on top of jQuery that is required for making the table visualization resizable.

Handlebars.js

Handlebars.js[13] is a JavaScript template engine and used to generate the HTML document based on the results of the provenance analysis. It is an extension of the logic-less template engine Mustache[14] and provides some level of logic that can be further extended by custom helper functions. A plugin for RequireJS facilitates the handling and activates the optimizer to pre-compile the templates into JavaScript code in order to reduce the required runtime.

DataTables

Displaying tables on a web page does not seem to be difficult. Indeed it is not, at least if the amount of rows and columns is not that large. It turned out that even modern web browsers are not able to load huge tables satisfactorily as they tend to freeze or crash in such scenarios.

The jQuery library DataTables[15] together with the extension Scroller remedies because it allows to only load and draw the visible part of a scrollable table. On

[10] http://requirejs.org/
[11] https://jquery.com/
[12] http://jqueryui.com/
[13] http://handlebarsjs.com/
[14] https://mustache.github.io/
[15] https://www.datatables.net/

scrolling, the required data is fetched from the server asynchronously. Therefore, the size of a data set to display does not matter in principle.

Data-Driven-Documents (D3.js)

D3.js[16], introduced in [BOH11], is a JavaScript library for creating visualizations usually meaning SVG graphics. It is neither intended nor able to generate predefined convenient visualizations, but drives the construction of graphical forms as it facilitates creating new DOM nodes based on text-based data. Moreover, animated transitions that interpolate styles and attributes over time extend the possibilities already afforded by CSS3.

In this work, D3.js is used for the dynamic generation of histograms, scatter plots and line mosaic plots.

Functionality One of D3.js's core concepts is the binding of data to DOM elements and their creation based on data. Binding means attaching data to the DOM tree which allows to access it at a later time, even using other tools and libraries. Following the functional programming style, D3.js employs method chaining as operators usually return a selection object representing a set of DOM nodes.

Listing 2.8 shows the steps required for creating three paragraphs (`p` elements) each including the numbers 1, 2 or 3. First of all, the DOM element `body` and all included paragraphs are selected if any exist yet. Otherwise, an empty selection is returned. `data()` parses and counts the given data whereby everything past this point is executed once for each single datum. The method `enter()` creates a new data-bounded placeholder element respectively selection for each data element for which no corresponding DOM element was found. Only subsequently a, `p` element is actually appended to the DOM as child of the `body`. Finally, the `text()` method evaluates the given function passing the current datum, and sets its return value as the paragraph's text content.

```
d3.select('body').selectAll('p')
  .data([1, 2, 3])
  .enter().append('p')
  .text(function(d) { return d; });
```

Listing 2.8: Core concept of D3.js for creating DOM elements based on data

Discarded Frameworks

Multiple JavaScript libraries and frameworks have been examined at very early state of development, especially for forming a clean structure of the whole appli-

[16] `http://d3js.org/`

cation. The library Backbone.js[17] seemed to be a good choice as it is based on the Model–View–Controller pattern and because it is known for its flexibility. However, it became apparent that most of the provided features are not really required. Beside the framework AngularJS[18], the comparable Ember.js[19] has been interesting, especially as a plug-in[20] for (large) tables already exists. In the end, it has turned out that all of the tested tools do not quite fit the needs. They are certainly great for building feature-rich web applications that require frequent database access and real-time updates of dynamic views as soon as the underlying data changes. But all of this is not needed for a comparatively simple application as the provenance visualization is. Database access is limited to reading and the analysis results do not change while the visualization is running. Moreover, separation of models and views as well as a notification system can be easily implemented by hand. For a basic structure RequireJS is entirely sufficient, and DataTables seems to be more appropriate than the Ember.js plugin.

2.4.3 Development

Syntactically Awesome Stylesheets (Sass)

Sass[21] is a preprocessor for CSS that allows among other things the use of variables, nested rules and custom functions such as for color manipulation. Sass supports two different syntaxes whereby for this work the Sassy CSS (SCSS) syntax is used that merely extends the one of CSS.

JSDoc

JSDoc[22] is a markup language for annotating JavaScript source code similar as Javadoc does with Java code. Finally, a HTML documentation website describing the complete code can be generated.

JSHint

It is probably easier to write sloppy code in JavaScript than in Python or compiled languages as Java or C as the language itself is not that strict. JSHint is a configurable JavaScript syntax checker and validator that tries to detect potential problems to increase the code quality. Moreover, it supports complying with coding

[17] http://backbonejs.org/
[18] https://angularjs.org/
[19] http://emberjs.com/
[20] https://addepar.github.io/ember-table/
[21] http://sass-lang.com/
[22] http://usejsdoc.org/

guidelines as, for instance, line length limitations. It is a less rigorous community-driven fork of JSLint that in turn has been inspired by the C tool Lint. As it is a static code analysis tool, it is not able to determine whether a program is correct or free of memory leaks.

JSHint is applied to all JavaScript code written as part of this thesis.

Node.js

Node.js is a server-sided platform for running JavaScript that is available for all major operating systems. While it is possible to build complex network applications with Node.js, it merely has been used during development of this work to optimize the build process.

Grunt

Grunt[23] is a Node.js task runner simplifying the execution of repetitive operations as compiling and unit testing. The following task is performed to build the final application code of this work:

1. *JSHint* — apply JSHint to all but third-party JavaScript code and break if problems are detected
2. *RequireJS* — run the optimizer to generate one single minified JavaScript file without log statements used for debugging
3. *Sass* — compile all SCSS files to one single minified CSS file
4. *Copy* — copy files such as the server application to a folder containing all release files
5. *JSDoc* — generate a HTML documentation website based on JSDoc annotations

[23] `http://gruntjs.com/`

Chapter 3

Implementation

After a short introduction to the overall setup, the particular components server and client including all chart types are expounded. A final section on fine tuning arrangements closes this chapter.

3.1 Setup

The whole application is broken into two tiers as illustrated in figure 3.1. A server component delivers files and the results of the provenance analysis fetched from a database, while a client component represents the graphical interface running in a web browser.

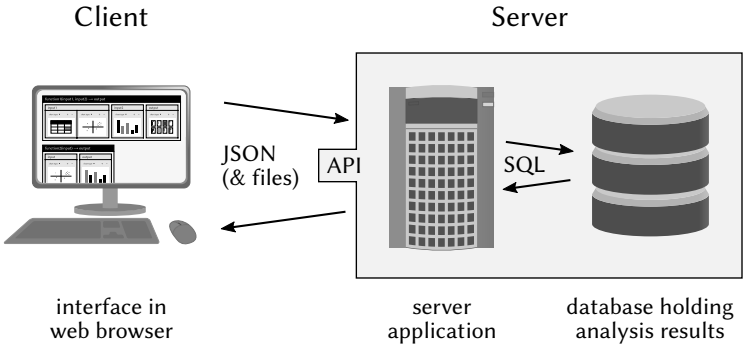

Figure 3.1: Overall client–server setup

The client demands the provenance data via an API that receives and returns JSON formatted strings. Static files are requested using HTTP request by the client's web browser. The server manipulates and retrieves data of the PostgreSQL database through SQL queries. Client and server application might run on the same computer system, but do not have to necessarily as the server might be accessible via the World Wide Web.

3.2 Server

The server's main purpose is to provide an API that allows the client component to retrieve the results of the provenance analysis. In this work, the server further delivers static files such as HTML, CSS, JavaScript and image files. However, latter task might be taken on by a more powerful web server in public environments. The application is based on Bottle.py and uses Psycopg to access the database.

Bottle.py allows to link an URL path called route to a callback function that generates the content of a response. The frameworks supports a wide range of content types and cares about a correct response header. Moreover, many Python data structures are automatically transformed into an equivalent JSON representation.

As serving static files is just a simple Bottle.py method call, only the implementation of the API, starting with database preparations, is further described.

3.2.1 Database Preparations

While all information required for generating the visualization is available in the database, it is not that easy to receive it. As the data is distributed over multiple tables, a lot of SQL joins are required to handle API requests. Moreover, recursive queries are essential to obtain all related values for a given one. Because those queries are expensive and run inconvenient long for large datasets, two additional tables `argumentvaluesRelations` and `returnvaluesRelations` are created and pre-calculated. They contain *complete* lists of related values (respectively their IDs) as one-dimensional arrays for each value that has any relationships. This means that not only the first related value is considered, but also its related ones and so on (in one direction). It is differentiated between backward relationships that indicate the provenance and forward relationships that exhibit the impact of a value. Furthermore, argument and return value IDs are always kept separate as their primary key value ranges overlap. So the additional relationship tables each contain four columns plus one column `id` referencing the respective value. Merely non-atomic related values are ignored to keep the results reasonable and the amount manageable. However, this is an optional optimization that can be readily deactivated in code.

Complex recursive queries fill the new tables whereby the fundamental principle is comparative simple. For forward relations the table `pmain` links argument values with return values, while `pcompo` does the opposite as illustrated in figure 3.2. Proper joins of those tables, visualized by informal join graphs in figure 3.3, can reveal all relationships in both directions.

So for instance, for retrieving all forward relationships of argument values the table to consider first is `pmain`. It gives the IDs of related return values (in column `returnvalueid`). Next, values related to those are essential. These are again argument values and can be found by performing a (left) join of `pmain` and `pcompo`

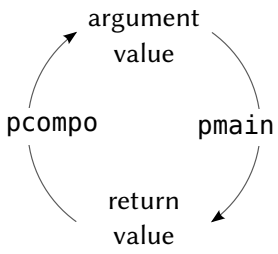

Figure 3.2: Forward relationships

under the condition that their `returnvalueids` match. The column `argumentvalueid` in `pcompo` now contains the desired IDs. After a further join of `pcompo` and `pmain` with concurrent `argumentvalueids`, the procedure starts from the beginning again. The process repeats itself as long as further relationships exist.

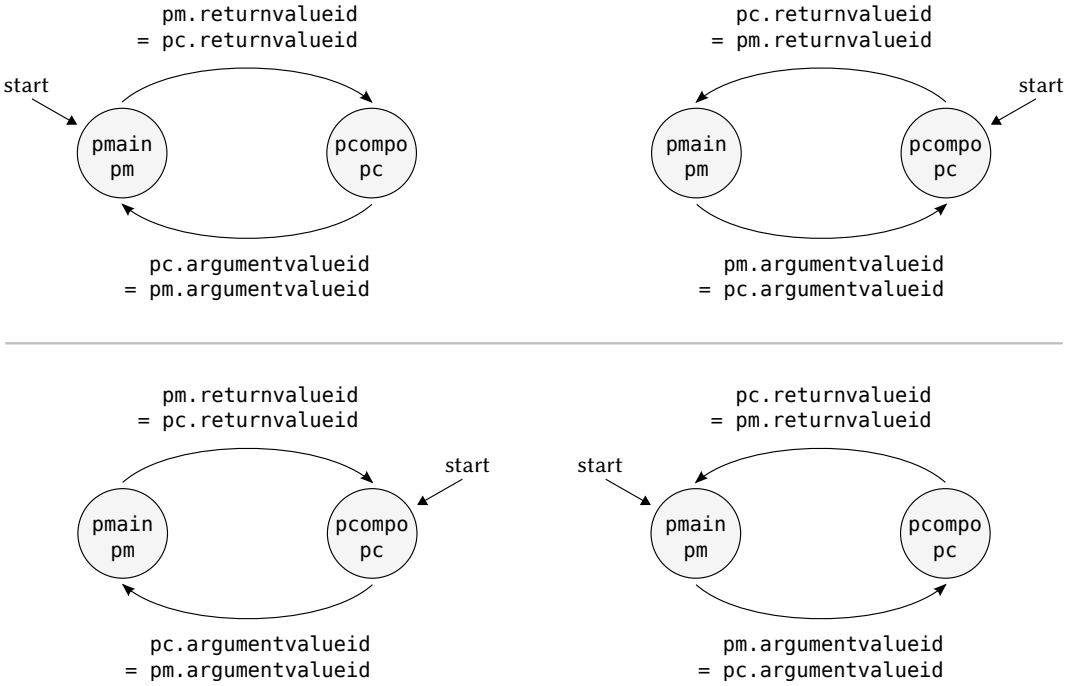

Figure 3.3: Informal join graphs for forward (left half) and backward (right) relations of argument values (top half) and return values (bottom)

Beyond that, the value tables `argumentvalues` and `returnvalues` make it hard to formulate queries that return values of a specific variable. So an additional column `varid` is created and filled by a straightforward recursive query that finds the ID of the respective root container.

The mentioned database preparation measures can be carried out subsequently to the provenance analysis or on server start triggered by the server application. Latter approach, supported by Psycopg, allows non-persistent modifications that only apply while the server is running. Especially because of an increased startup

time this strategy has been rejected.

3.2.2 Application Programming Interface (API)

The server provides an API with two resources that allow clients to fetch the results of PAT via HTTP. The resources are explained below, while exemplary JSON formatted responses, based on the analysis of the sample program in listing 2.1, are illustrated in listing 3.1 and 3.2.

```
{
  "calls": [
    {
      "id": 2,
      "name": "sum",
      "parameters": {
        "input": [
          {
            "id": 1,
            "name": "arg_data",
            "subscripts": [
              "a",
              "b",
              "c"
            ]
          }
        ],
        "output": [
          {
            "id": 1,
            "name": "res",
            "subscripts": [
              "v"
            ]
          }
        ]
      }
    }
  ]
}
```

Listing 3.1: Exemplary response to `/api/calls` ↑

```
{
  "requestid": 1,
  "data": [
    {
      "a": {
        "id": 3,
        "value": "1",
        "relations": {
          "forward": {
            "output": {
              "1": [ 3 ]
            }
          }
        }
      },
      "b": {
        "id": 5,
        "value": "2",
        "relations": {
          "forward": {
            "output": {
              "1": [ 3 ]
            }
          }
        }
      },
      "c": {
        "id": 4,
        "value": "3",
        "relations": {}
      },
    }
  ],
  "count": {
    "total": 1,
    "this": 1,  affected by limit*
    "filtered": 1 affected by ids*
  }
}                * if any
```

Listing 3.2: Exemplary response to `/api/values` →

`/api/calls` Returns meta data about function calls such as function names and associated parameters together with their subscripts. No additional parameters can be passed to the resource.

`/api/values` Delivers atomic values and IDs of related values grouped by parameter as well as meta data. The following parameters have to be passed:

- `type` — either `input` or `output` indicating the parameter type
- `subscripts` — comma-separated sequence of subscripts

These parameters are optional:

- `varid` — identifier of a parameter
- `requestid` — numeric identifier that is part of the response
- `offset` — number of containers to skip
- `limit` — maximal number of containers to return
- `ids` — comma-separated sequence of value IDs used for filtering

Originally, it was planned to formulate SQL queries so that their results can be returned as response without further processing for best performance. However, that turned out to be difficult as PostgreSQL does not really support nested query results. For instance, aggregate functions such as `array_agg()` only work for atomic values, but not for arrays. While multi-dimensional arrays are possible, it is not allowed to create arrays with varying dimensions (see figure 3.4, a). Other

a)	`ARRAY[ARRAY[1], ARRAY[1,2]]`	illegal
b)	`row(ARRAY[1], ARRAY[1,2])`	legal, but still unsatisfactory

Figure 3.4: Attempts to create multi-dimensional structures in SQL

attempts such as working with row constructors (b) are not satisfactory due to bad support by Psycopg that necessitates the use of regular expressions to obtain the individual parts of the result. After all, it seemed like there is no appropriate way to get along without post-processing the query results.

The progressed approach makes use of comparatively simple and straightforward SQL queries that benefit from the introduced tables `argumentvaluesRelations` respectively `returnvaluesRelations`. After the particular SQL query was executed, the Python application once iterates over the query results with cost in $O(n)$ to obtain the proper nested structure.

Some arrangements improve the server's performance and reduce the response time. Count statements are cached, so they have not to be executed for each request. The same applies for the parameter identifier `varid` that is cached for each `argumentvalue` and `returnvalue` on server start to save one expensive SQL join and to simplify the queries.

3.3 Client

As the client component is comprehensive, not all singe aspects, but only the main ones as the core model and the different charts are addressed in depth hereinafter.

3.3.1 Overall Structure

The client application running in web browsers is based on modules supported by RequireJS and follows the object-oriented programming style. Figure 3.5 shows an UML diagram of the most important classes and relationships. For clarity only the prime attributes and methods are listed.

The application's entry point is the file *main.js* that initializes a single `MainController` object. It receives the `chartLoader` module which simply wraps the constructor functions of the various chart types. Moreover, the controller requests the `/api/calls` resource to create corresponding `Call` and `Parameter` objects on the basis of its response. These objects are passed on a Handlebars template that is evaluated in order to provide the content of the web page. For each `Parameter` object the appropriate DOM node is located and stored in a (map-like) object. Furthermore, a `ChartsController` is created per parameter which allows the user to select a chart type and to add or remove charts. All concrete chart classes inherit from `BaseChart` with an abstract method `render()` that has to be implemented by subclasses such as `TableChart` or `ScatterplotChart`. The chart classes using D3.js for graphic generation also inherit from `D3Chart` and `ConfigurableD3Chart`, both providing appropriate helper methods.
Two observable model classes are required for interaction and correct highlighting of selected and related values in all charts. The main model `SelectionModel` is instantiated only once, and holds the IDs of currently selected as well as related values. Public methods allow selection or unselection whereby related values are always considered. Beyond that, the `AggregationModel` is additionally used apart by charts that aggregate values, which are `HistogramChart` and `LineMosaicPlot`. It also stores IDs of selected and related values, but now grouped by individual aggregates (e. g. bars of a histogram) in order to determine how many values of an aggregate are actually selected.
Beyond that, further classes exist like for the definition of custom data structures as `Set` and `ValueMeta`. The important observer pattern is backed by event classes and interfaces whereby latter only exist for documentary purposes as JavaScript does not support that type natively.

3.3.2 Selection Model

The `SelectionModel` is the application's most significant model and required for a correct highlighting of chart elements. Chart objects register as observers and

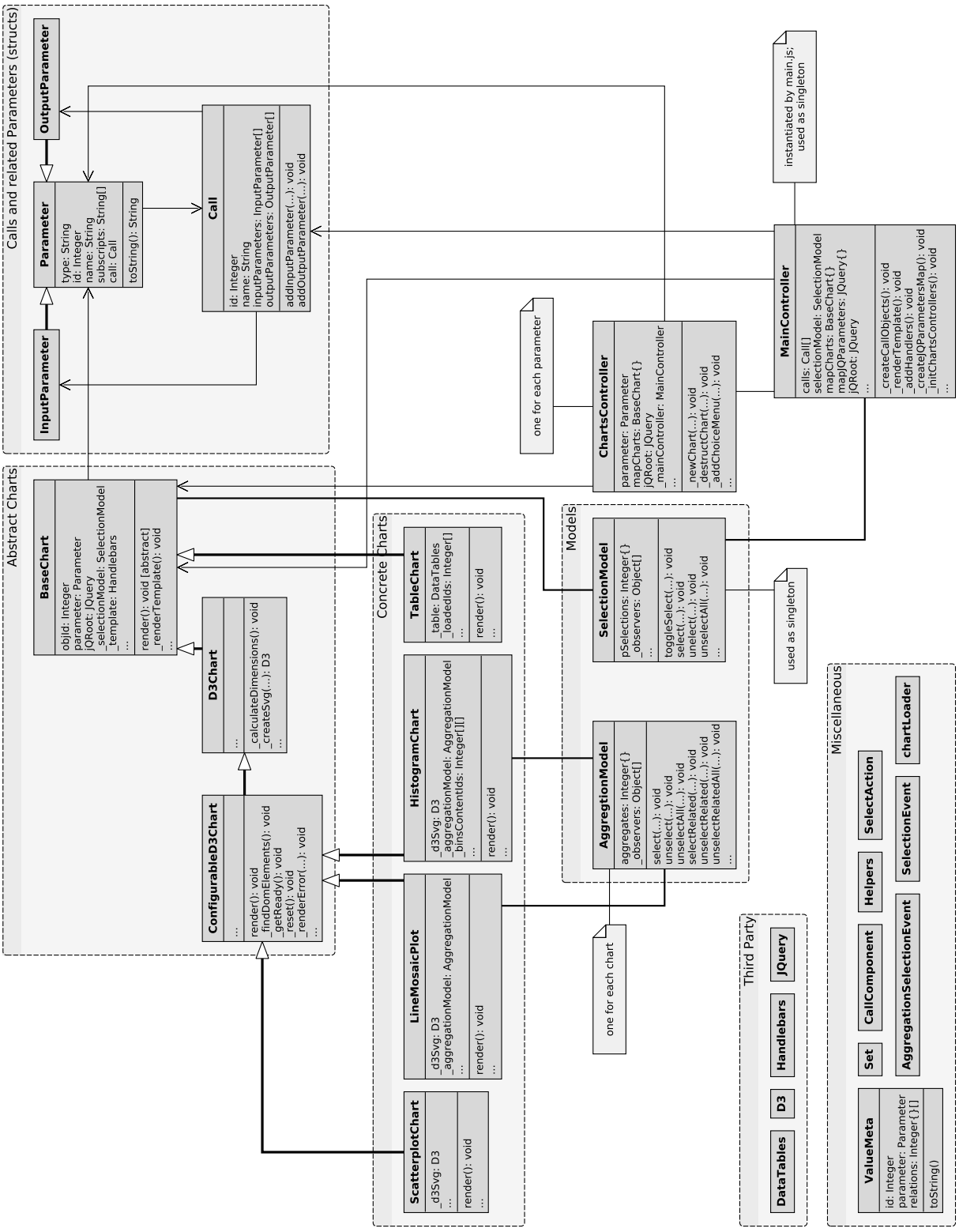

Figure 3.5: *Incomplete* UML diagram of the client application illustrating the most important classes and their prime attributes, methods and relationships

get notified once the selection state of associated values changes. Public methods such as `toggleSelect(...)`, `select(...)`, `unselect(...)` and `unselectAll(...)` allow the chart objects to change the selection state of passed values. It has been put emphasis on the performance of the model as it might have to process large amount of data.

When an user selects a value in a chart, it is not sufficient to just inform all chart objects containing related values about the event so that these values get highlighted. But it is also necessary to store the IDs of all selected and related values as the following example, illustrated in figure 3.6, shall justify:

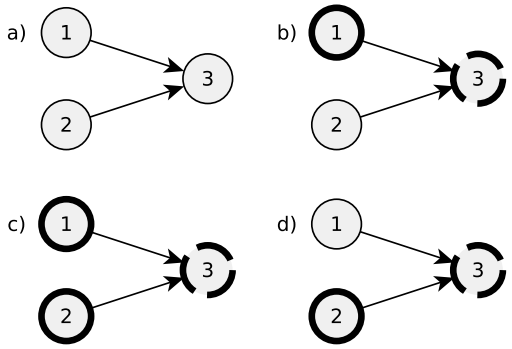

Figure 3.6: Exemplary relationships and selection states

a) Given three values 1, 2 and 3 where each of the first two is related to the last one.

b) When value 1 gets selected by the user, the related value 3 gets selected, too.

c) If value 2 gets selected afterwards, the likewise related value 3 does not change its state as it is already selected.

d) Once the first value gets unselected, it is not possible to just unselect all of its related values—that is value 3—because they might be bound by another selection—value 2 in this example.

a) Value 3 has to stay selected as long as one or more related values are selected. Not until value 2 gets unselected, the third one can revert to its initial state.

As usual, there exist many ways to accomplish this task, so multiple approaches have been considered.

1. Maintain a set of IDs of selected values and furthermore a set for *each* related value containing IDs of associated (selected) values. However, JavaScript does not have a set data type and an own implementation leads to a certain overhead. Moreover, a lot of set objects would be necessary.

2. Maintain two arrays, one for selected values and one for selected related ones. As arrays are dynamic in JavaScript and behave similar to list known from other programming languages, it is easy to add and remove IDs. Beyond that, arrays may contain duplicates which allows to insert related values multiple times reflecting the relationship to multiple values. In contrast to a custom set implementation native arrays benefit much more from internal optimizations by web browsers.

The previous approaches have got a disadvantage in common: as the IDs are only grouped by type (selected and selected related), all chart objects get informed about changes on every value regardless whether the value is associated to the parameter the particular chart visualizes or not. So a lot of data is unnecessarily processed by the chart objects which has a perceptible negative impact on the overall performance. For this reason, two arrays analogous to the second approach above are maintained, but now for each single parameter. The select method gets passed `ValueMeta` objects that contain, beside the numeric `id` and lists of `relations`, a reference to the `parameter` the value belongs to. The `relations` are already grouped by parameter due to an adjusted API. All this allows much more specific notifications of the chart objects that generally only register for one parameter. The observers get passed a `SelecionEvent` that contains, apart from a type indicator and a timestamp, a complete list of value IDs that are affected. This corresponds to the push approach of the observer pattern. The pull variant, by which the observers call themselves for the desired data, is unsuitable for requesting IDs of values that have been unselected as they do not exist in the model afterwards. Also a pursuant computation by the chart objects seems to be unreasonable.

The implemented approach based on the use of simple arrays also shows a weakness: operations that test the existence of a value in an array are expensive in comparison with sets. However, this merely has a negative impact on the rarely called `_unselectRelated(...)` method that needs to test whether an unselected related value is still referenced by another selected value. The usually more often executed methods `select(...)` and `unselectAll(...)` are unconcerned and profit from the simple structure. To sum up, approaches using sets perform much better on a rarely called method, but are slower on other more significant operations.

Further arrangements improve the runtime of the `SelectionModel`'s methods. Most jQuery methods that have been used in an early state of development, including `jQuery.each()` and `jQuery.inArray()`, are replaced by native equivalents which, admittedly, are not always supported by old browsers like the Internet Explorer 8. Some more changes listed in figure 3.7 improve the performance additionally.

Worse	Better
`a1 = a1.concat(a2)`	`Array.prototype.push.apply(a1, a2)`
`o1[o2.toString()]`	`o2[o1.type][o1.id]`

Figure 3.7: Quality of statements with regard to performance

3.4 Charts

The diverse chart types form the essential part of this work as they visualize data in a variety of ways for different purposes. All charts have in common that they illustrate relationships between values and that they look and behave similar in order to provide a consistent user experience.
Once the user clicks on a single value or an aggregation (such as a bar of a histogram), it gets highlighted together with all related values. Pressing the control key while clicking allows to select further values without loosing former selections. It also enables toggling which means that selected values get unselected and vice versa.

As mentioned, all charts inherit from a base class and share the same selection model. Furthermore, they each have one single parameter of the analyzed program associated. The chart construction always follows a similar pattern: first, a Handlebars.js template defining the basic DOM structure is evaluated and rendered, before the server API is requested and the actual chart is created. Selected values are highlighted in orange, while related ones are marked in red. This is usually done by setting a DOM class that applies a CSS rule.

In the following, the different chart types are described more elaborated putting emphasis on the characteristic features of each one. In regard to highlighting only the select operation of single values is explained as unselecting specific or all values works analogously or is not noteworthy.

3.4.1 Table

The table is the most basic visualization type of this work and supports both numbers and strings. It represents the plain data processed by the analyzed program without further manipulation. It copes best with large data sets and is suitable for getting a detailed insight. The table is resizable and unselected rows can be hidden. The implementation is based on the jQuery library DataTables whereby some artifices had been necessary to get it work as desired.

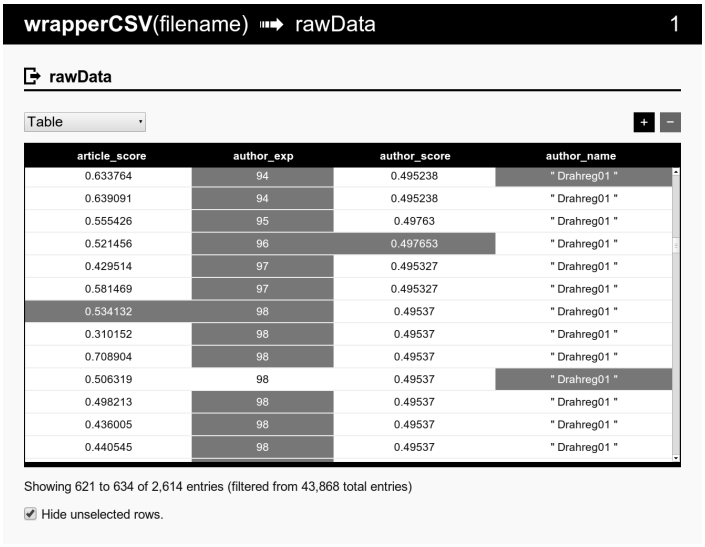

Figure 3.8: Table showing sample data with enabled hiding of unselected rows

Deferred rendering and performance

Backed by DataTables, only the visible part of the table plus a small offset on either side gets drawn which includes the creation of corresponding DOM elements. While scrolling the content gets updated with data received by further API requests. Due to this deferred rendering the client's overall performance stays approximately constant independently from the size of the data set.

Additional improvement of performance has been achieved by caching the IDs of currently drawn values respectively DOM nodes. This allows filtering of IDs when receiving an update by the selection model in order to prevent trials to access non-existing nodes for highlighting. As DOM operations are comparatively expensive, it is worthwhile to keep their amount as low as possible.

Data

Since the structure of the data returned be the API does not conform to the format required by DataTables, it gets transformed as shown in figure 3.9. Because DataTables does not support the annotation of meta data, such as IDs and relationships, for individual cells, those data is assigned to the respective row. Afterwards, a callback function executed for each cell attaches the meta data to the cell node using the jQuery method `data()`.

Hiding of unselected rows

Especially for large tables it might be hard to observe all relevant rows that currently contain selected values. So an option to hide all unselected rows, which can

```
{
  "0": { // first container respectively row
    "a": {
      "relations": { /* relations of a */ },
      "id": 3,
      "value": "1"
    },
    "b": {
      "relations": { /* relations of b */ },
      "id": 4,
      "value": "2"
    },
    "c": {
      "relations": { /* relations of c */ },
      "id": 5,
      "value": "3"
    }
  }
}
```

```
[
  { // first container / row
    "a": "1",
    "b": "2",
    "c": "3",
    "DT_C_relations": [
      { /* relations of a */ },
      { /* relations of b */ },
      { /* relations of c */ }
    ],
    "DT_C_ids": [
      3,
      4,
      5
    ]
  }
]
```

Figure 3.9: Part of the raw API response (left) and corresponding postprocessed data (right)

be activated by the user, has been implemented.

The IDs of all selected values offered by the selection model are passed to the API for filtering. As DataTables lacks a way to manually trigger a custom call to the server, its search feature has been assigned to this task. Special search keywords identify desired actions allowing to adapt the request parameters or to clear the table if no values are selected. This kind of workaround works properly, especially as the very search feature is not activated due to missing support by the server API.

Resizing

The user may change the height of the table by intuitive dragging, while the width is adapted to the content by DataTables. After the table's height has been changed with the help of a jQuery UI widget, DataTables properties get recalculated and adjusted correspondingly.

3.4.2 Scatter Plot

A scatter plot displays numeric value pairs in a two-dimensional Cartesian coordinate system resulting in a point cloud. Before the plot is rendered, the user may choose two subscripts defining the horizontal and vertical axis. Moreover, the point size and the overall dimensions of the scatter plot are configurable. The

points get drawn semi-transparent allowing to recognize accumulations which yield darker areas. It is not only possible to select single points by mouse click, but also using a so-called brush to select areas containing several points.
As a point represented by a DOM element is generated for each value pair and due to the large impact of the DOM tree's size on the overall performance of the application and web browser, the scatter plot is less suitable for very large datasets compared to the other chart types .
The SVG graphic representing the scatter plot is generated with the aid of D3.js.

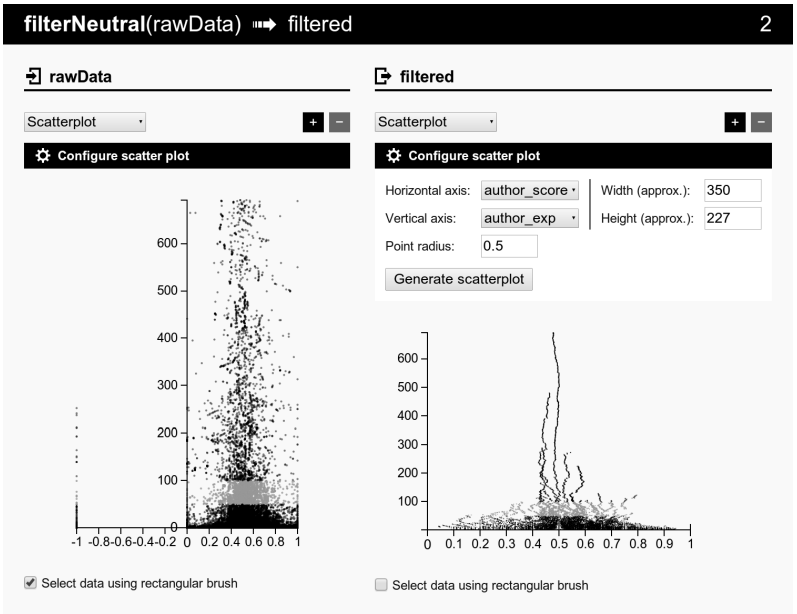

Figure 3.10: Scatter plots of two large datasets with each more than 25,000 points whereby the gray area contains a huge amount of value pairs selected by a brush

Overall construction

The creation of new DOM elements, such as the points, is done as described in the introduction of D3.js in sub section 2.4.2.
Understandably, the coordinates of the points correspond to the values of the dataset obtained by the server API. In order to map from the input domain defined by these values to the output range defined by the desired dimensions of the plot, D3.js provides scale functions. Furthermore, D3.js's axes component displays reference lines and takes care of human-readable tick values. After all, the library makes the construction really straightforward.

Filtering

Once the scatter plot objects receives a selection event, including the IDs of recent selected or unselected values, the point DOM elements are filtered accordingly. A filter method provided by D3.js is not used for this purpose as its runtime is insufficient for large datasets. Instead, a custom and more adjusted approach is applied which makes use of the knowledge that each value ID has only one associated DOM element. Moreover, converting the array of IDs to a set-like object notably speeds up tests whether a value is contained.

Bringing points to the front

Especially for large datasets points might overlap and hide each other. This is a problem as soon as a partly or completely covered point gets selected as it would not be visible for the user. For HTML documents the CSS property `z-index` helps as it specifies the z-order to determine which element covers the other. However, this property does not work on SVG graphics where only the order of elements in the DOM tree defines the z-order. The later an element appears in the DOM tree, the further up it gets positioned. So in order to bring an element of a SVG graphic to the front, it has to be moved to the end of the respective DOM sub tree. If hundreds or thousands of points get selected at once, the required reordering would be a too expensive operation. As reasonable compromise, elements only get brought to the front if doing so is not necessary for more then 50 elements. That ensures that for small selections with less than 51 points involved all relevant points get visible, whereas it is usually negligible for larger selections if some points are covered.

Brush selection

D3.js's brush control allows to span a rectangle in order to select underlying points. However, as the control only returns a value extent indicating the rectangle's cover in both horizontal and vertical direction, the points have to be filtered manually. Combining single point and brush selection so that a mouse click selects a single point, while mouse dragging spans the rectangle, is not easily possible due to the functional principle of the brush control. Instead, the brush selection can be toggled by a checkbox below the plot that is deactivated by default.

3.4.3 Histogram

A histogram visualizes the distribution of values by grouping them into bins. Bins are illustrated as rectangular bars whose size show the frequency of values over discrete intervals. The user has to select a variable first and might adjust the number of bars as well as the overall dimensions. Afterwards, the calculated histogram

appears animating the increase of each bar's height. The bars are colorized depending on how much of their contained values are selected: the more the darker. Furthermore, the bar size can be displayed optionally, and tool tips show further information about bins on hovering if desired.
This chart type also makes use of D3.js and requires an additional `AggregationModel` to determine the share of selected values per bin.

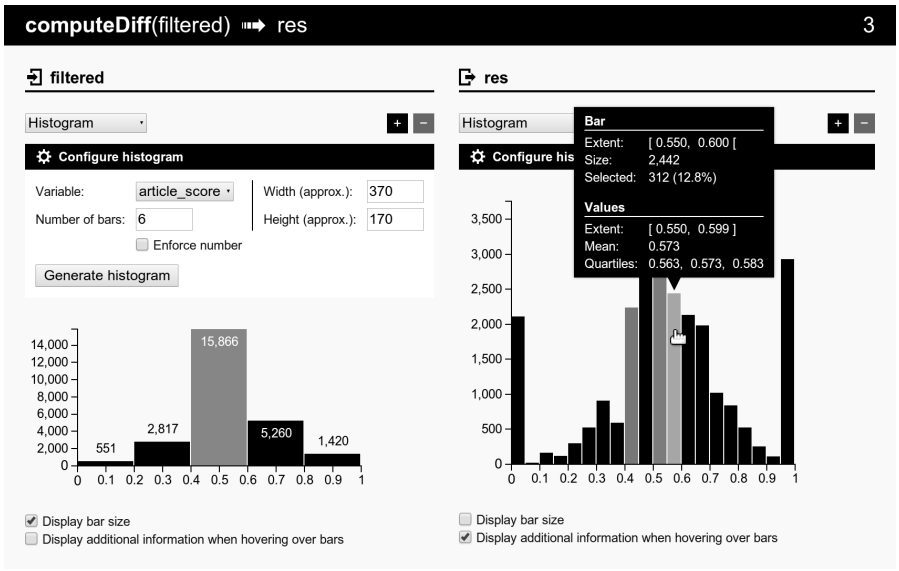

Figure 3.11: Two sample histograms demonstrating prime features of this chart type such as the bar size indicator, dependent highlighting and tool tips

Overall construction

Thanks to a so-called histogram layout by D3.js the construction is not that complicated as the library calculates the bins and associates appropriate values.

Number of bars The desired amount of bars respectively bins is usually merely regarded as a hint because D3.js uniformly spaces the value domain so that the interval values are human-readable. So for instance, a single bin might cover the interval $[10, 20]$ instead of $[8.284, 20.653]$. As a result, the resulting amount of bins may be less or more than set. Typically, the exact amount does not matter for the user, but if so, it can be enforced optionally. In that case, the intervals are set every $\frac{\text{value extent}}{\text{number of bins}}$ units yielding non-integer values in most cases.

Tool tips

Small info boxes, called tool tips, show additional information for each bin on mouse hovering. They are implemented using the plugin d3.tip[1] and a Handlebars template, and can toggled by clicking on a checkbox below the histogram. The tool tips display the following details:

- *Bin extent* — the bin's lower and upper bound
- *Bin size* — the bin's frequency respectively the number of contained values
- *Selected / related* — the number and share of contained values that are selected
- *Value extent* — the minimum and maximum contained value (has not to correspond to the bin's extent necessarily)
- *Mean* — the average of all contained values
- *Quartiles* — the 25, 50 and 75 percent quantiles indicating the value distribution inside the bin. For instance, the 25 percent quantile is the value such that 25 % of the values are smaller. The 50 percent quantile is called median.

Highlighting and performance

Once the histogram object is informed about new selections by the selection model, it has to find involved bins by grouping the recent selected values' IDs accordingly. An obvious approach applies two nested loops to iterate over the IDs s of selected values and over the IDs b of all values contained by bins. However, its quadratic complexity of $O(|s| \cdot |b|)$ is not satisfactorily. Instead, an improved method with a complexity of $O(|s| + |b|)$ has been implemented that makes use of a set-like data structure. An intersection operation finds all IDs that are contained by a bin, while a difference operation detects all IDs that still require an association. Intersection and difference are calculated in the same iteration.

When the user clicks on a bar, the contained values' IDs are passed to the selection model that in turn informs the histogram object about the selection. In that case, it is not necessary to search for associated bins as the single bin involved gets cached once the user selects it.

Once all involved bins are known, they are passed to the aggregation model together with associated value IDs. The model updates its state and notifies the histogram including the bin statuses, so they finally get highlighted.

[1] `http://labratrevenge.com/d3-tip/`

3.4.4 Line Mosaic Plot

The line mosaic plot, based on the article [Huh04b], is the most sophisticated and complex chart type of this work and allows to recognize relationships among categorical variables. Just like conventional mosaic plots, it visualizes so-called contingency tables where each cell contains the frequency respectively count of a distinct value combination. In traditional plots each cell is displayed as a rectangle whose dimension corresponds to the respective frequency. However, this kind of plot might be misleading and confusing as it is difficult to compare rectangle sizes, especially if they vary in aspect ratio. Moreover, missing alignment of columns and rows in conventional plots makes an interpretation even harder. These disadvantages do not apply to the line mosaic plot as it consists of lines split inside uniform boxes, referred as cells, instead of different-dimensioned rectangles. The line lengths correspond to each cell's frequency and are adjusted to the cell with maximum count. If a variable has been chosen as "target", each cell contains different-colored lines for each value (if any) of the target variable.

The user may select one or more variables, optionally including one target, and can set the overall dimensions as well as the lines' width and spacing. If desired, tool tips show additional information.

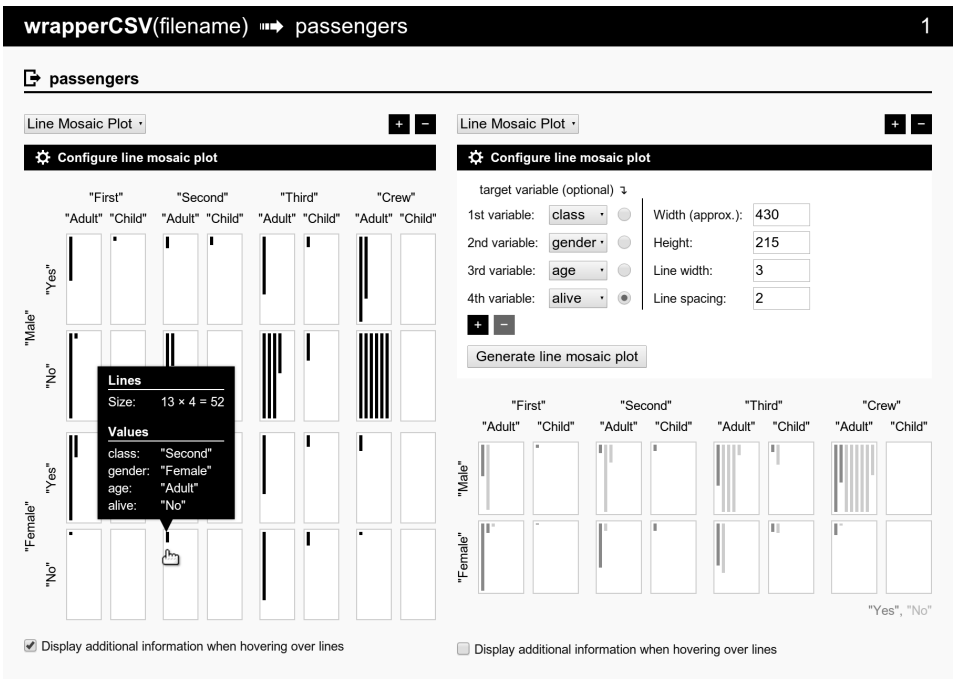

Figure 3.12: Two line mosaic plots visualizing the often-cited dataset of the passengers on the sunk Titanic grouped by *class (first, second, third, crew)*, *gender (male, female)*, *age (adult, child)* and whether *alive (yes, no)*. On the right plot, the variable *alive* has been selected as target. Among other things, it is recognizable that the chance of survival depends on the class.

Functional principle and implementation, including some optimizations, correspond to those of histograms to a great extent. The computation of the contingency table is adopted from [Huh04b] with small modifications, whereas own partly complex calculations determine positions and sizes of the plot elements. The fact that an arbitrary amount of variables can be used proves the powerfulness of all algorithms and formulas. Again, D3.js is used to draw the resulting SVG graphic.

Overall construction

The basis for the construction is the calculation of the contingency table as described in [Huh04b]. Unfortunately, some of the presented formulas are a bit defective and had to be corrected first. For this purpose, a concrete example in the underlying article as well as the inspection of the source code of the visualization tool eDAVIS[2], described in [Huh04a], has proved beneficial. Figure 3.13 shows the revised formulas without further explanation. Besides, the article gives a formula to calculate the size of gaps between cells in both horizontal and vertical direction. They vary in order to simplify perception.

page 4:

$$I = \sum_{i=1}^{\lfloor p/2 \rfloor} (v_{2i} - 1) \prod_{j=i+1}^{\lfloor p/2 \rfloor} n_{2j} + \cancel{v_{2\lfloor p/2 \rfloor}}^{1}$$

$$J = \sum_{i=0}^{\lfloor (p-1)/2 \rfloor} (v_{2i+1} - 1) \prod_{j=i+1}^{\lfloor (p-1)/2 \rfloor} n_{2j+1} + \cancel{v_{2\lfloor (p-1)/2 \rfloor + 1}}^{1}$$

page 5:

$$Mod(x, y) = x - \overbrace{\cancel{x}}^{y} \cdot \left\lfloor \frac{x}{y} \right\rfloor$$

$$v_i = \begin{cases} 1 + \left\lfloor \dfrac{\overbrace{I}^{+1}}{\prod_{j=\lfloor i/2 \rfloor + 1}^{\lfloor p/2 \rfloor} n_{2j}} \right\rfloor, & i = 2, 4, \ldots, 2\lfloor p/2 \rfloor - 1 \\ [\ldots], & [\ldots] \end{cases}$$

and apply $Mod(v_i, n_i)$

Figure 3.13: Revised formulas concerning [Huh04b]

Dimensions After the contingency table and the size of gaps are calculated as mentioned, further dimensions need to be determined, beginning with the width

[2] `http://stat.skku.ac.kr/myhuh/DAVIS.html`

of single cells wrapping the lines. As recognizable in figure 3.14, the cell width can be estimated (ignoring labels) by:

$$\frac{\text{total width} - \text{total horizontal gaps}}{\text{number of columns}}$$

However, this might lead to non-integer values which is not suitable for a clean rendering. Moreover, this simple approach does not ensure that all lines fit perfectly into the cells with given line width and spacing.
Therefore, the number of lines that entirely fit into a cell, wide as calculated above, is determined. If a target variable has been defined, however, the amount of required lines might increase as cells are usually not tightly packed. Finally, the cell width is adopted to the maximum need of lines. The overall plot width gets updated accordingly, while all other user-defined dimensions remain unaffected.
Further straightforward computations determine heights and label dimensions.

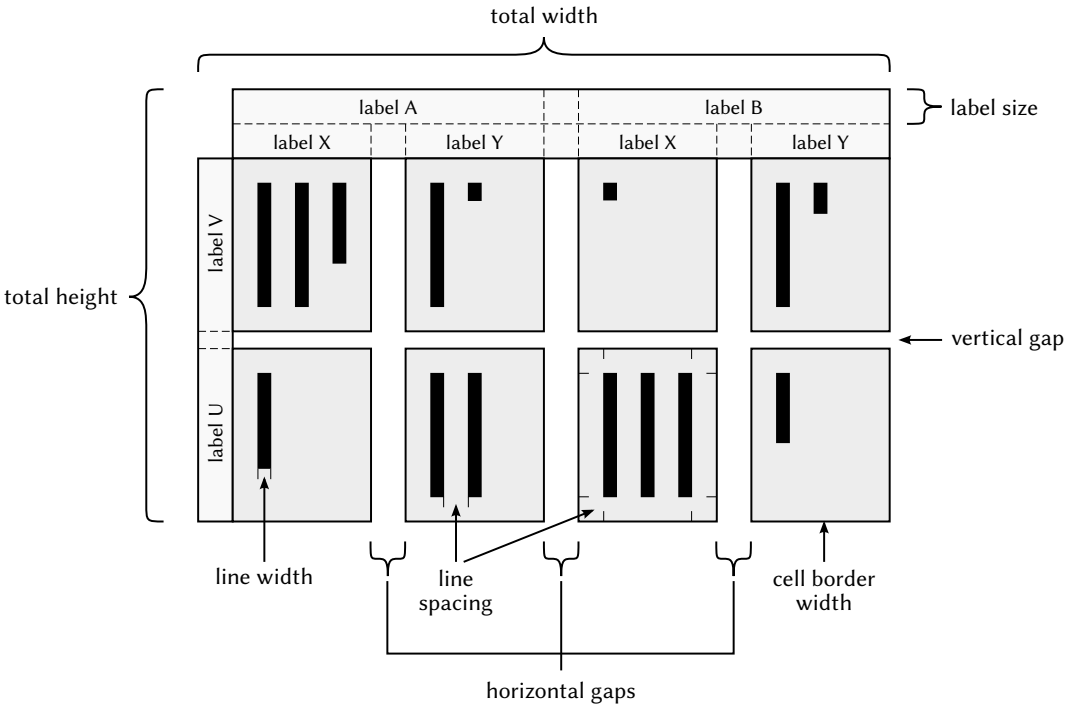

Figure 3.14: Construction scheme of a line mosaic plot

Lables The cell labels denote a variable value for one or more columns respectively rows, as discernible in figure 3.14. What makes computation complex is the circumstance that the size of gaps between cells varies and that the amount of labels depends on the values. The essence of the implemented algorithm is that it determines how much cells and gaps each label spans which allows to shift it to the proper position. The spanned area is equal for each label corresponding to a

value of the same variable. Figure 3.15 shows a highly simplified extract of the iterative algorithm in pseudo code. It calculates a coordinate of the label's center in either horizontal or vertical direction.

```
cumulatedGaps ← calculate cumulated sum for each gap;
foreach variable v do
    spannedCells ← get total size of cells that each label spans;
    spannedGaps ← get toal size of gaps that each label spans;
    spannedTotal ← spannedCells + spannedGaps;
    foreach label of v with index i do
        leftGapIndex ← calculate index of gap left to the label;
        coordinate ← (i + 0.5) · spannedTotal
                     + (cumulatedGaps[leftGapIndex] or 0)
                     − i · spannedGaps;
        // do something with the coordinate, e. g. render the label text
    end
end
```

Figure 3.15: Calculation of the coordinate defining a label's center whereby the algorithm works in both horizontal and vertical direction

Tool tips

The tool tips work analogous to those of histograms and show the size and selection state for each cell respectively line group. Additionally, the associated values are listed. As those do not arise while the contingency table is calculated, they need to be determined by a further algorithm given in the underlaying article.

3.5 Fine-tuning

In order to increase the quality of the developed software, it has been analyzed and optimized.

3.5.1 Memory Leaks

The responsible use of memory is an important aspect, especially if dealing with a large amount of data and complex charts. In contrast to most low-level programming languages, such as C, JavaScript does not allow to explicitly allocate and free memory. Instead, memory is allocated when objects are created and freed if they are not needed respectively used anymore. This is done by a garbage collector that usually considers an object as not needed if it is not referenced by another

object. Memory is called leaking when the garbage collector is not able to release memory of actually unused objects. This might happen if not all references have been removed and particularly if multiple objects contain cross references to each other.

In this work, partly serious memory leaks occurred when a chart was removed by the user. Although the sole reference to the chart object has been deleted and all corresponding DOM nodes have been removed, not all memory previously required by the chart was freed. The developer tools of the web browser Google Chrome showed that most of the chart's DOM nodes were still part of a detached DOM tree and therefore kept in the JavaScript heap. Even though not being able to reconstruct all causes for this leaks, they have been resolved. Before the reference to a chart object gets deleted, not only the chart's root node gets removed. Prior to this, large DOM subtrees are removed explicitly and some object properties are set to `null`.
It seems likely that internal caching mechanisms of jQuery have a share in the memory leaks. Definitely some messages logged to a browser console for debugging cause leaking which is why all logging statements get removed by RequireJS for the final code.

Figure 3.16 shows the result of this effort. First, a chart—a line mosaic plot in this example—is created causing the use of memory by the JavaScript heap and the number of DOM nodes to increase. Once the user removes the chart, it gets removed immediately from the browser's view. However, the use of memory does not change a lot, but only when the garbage collector is executed. The fact that the heap size and amount of nodes are approximately the same before and after the chart's existence demonstrates that no or less memory is leaked. If a memory leak existed, the consumption of memory would not decrease that much once garbage is collected.

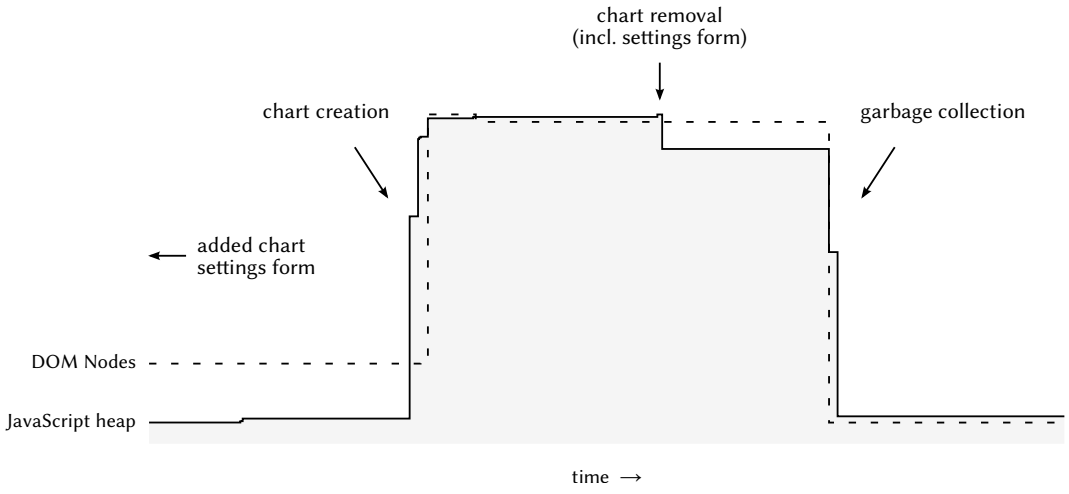

Figure 3.16: Timeline showing the heap size and number of DOM nodes in memory while a chart gets created and removed (recorded with Google Chrome 37)

3.5.2 Validation

While the JavaScript code has been checked by JSHint, the markup of the resulting website including the charts has been validated by the World Wide Web Consortium's (W3C) validation service[3]. Merely the use of a HTML5 `data` attribute on SVG elements, required for highlighting of histogram and line mosaic plot elements, is faulted. As it works on all modern browsers nevertheless and because an alternative approach would be disproportionately effortful regarding implementation and runtime, this kind of flaw is accepted.

[3] `http://validator.w3.org/`

Chapter 4

Conclusion and Outlook

4.1 Conclusion

The implementation of a browser-based interface for provenance analysis provided by the given analysis tool has been successful. The resulting implementation works well and allows the user to choose from four different chart types, each having merits of its own. Thanks to numerous optimizations in nearly every component the size of datasets that can be processed has been increased. Moreover, the application is extensible by further chart types due to a well-conceived structure and an universal server API.

The fact that the charts are generated on client-side in a web browser that furthermore has to handle relationships and highlighting, however, sets certain limits in regard to the amount of processible data. The interface has successfully been tested with a dataset of about 44,000 tuples, but a much higher amount would probably overstrain the efficiency of web browsers on common computers.

4.2 Future Work

Scatter plot efficiency Displaying single value pairs the scatter plot might lead to the creation of a large amount of DOM elements representing the points. This has a great influence on the overall performance of the whole interface and involves that the scatter plot copes worse than any other chart type with mass of data. In order to massively reduce the number of DOM elements, points that cover or are located next to each other could be combined respectively accumulated.

Waiting indicator When an user clicks on a chart element, such as a bar of a histogram, in order to select values it might take some moments until all highlighting is calculated and displayed. Indicating that the application is still working,

for instance by changing the mouse cursor to a waiting symbol, would probably satisfy the user.

Communication layer In the current implementation the client application processes the API responses directly what implies that all client components such as the several chart classes are adjusted to that specific interface. Therefore, modifications of the API's parameters or of the structure of its responses would require an adaption of many different parts of the client software. The introduction of an additional communication layer that requests the API and that passes the maybe post-processed response to other components would bring more flexibility. If the API was changed or replaced, merely that layer would require an adaption.

API extension The library DataTables, used for the implementation of the table, allows to sort columns by clicking on the respective column heading. Moreover, a search field simplifies tracing of specific values. Currently, the API does not support sorting and searching which is why these DataTables features are not enabled. By extending the API an enhancement of usability and functionality would be achieved.

4.3 Outlook

Currently, the analysis tool by the Database Research Group at the University of Tübingen computes data provenance concerning Python programs. This is considered as a first step towards the actual aim of analyzing SQL queries that conceivably get initially translated into equivalent Python programs. The graphical interface elaborated in this work might get extended and adapted to this altered purpose.

Bibliography

[Wei81] Mark Weiser. "Program Slicing". In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.

[KL88] B. Korel and J. Laski. "Dynamic Program Slicing". In: *Information Processing Letters* 29.3 (Oct. 1988), pp. 155–163.

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. "Why and Where: A Characterization of Data Provenance". In: *Proceedings of the 8th International Conference on Database Theory*. ICDT '01. London, UK, UK: Springer-Verlag, 2001, pp. 316–330.

[Huh04a] Moon Yul Huh. "eDAVIS: An Enhanced Data Visualization System". 2004.

[Huh04b] Moon Yul Huh. "Line Mosaic Plot: Algorithm and Implementation". In: *COMPSTAT 2004 — Proceedings in Computational Statistics*. Ed. by Jaromir Antoch. Physica-Verlag HD, 2004, pp. 277–285.

[al11a] World Wide Web Consortium (W3C) et al. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification (W3C Recommendation)*. June 7, 2011. URL: http://www.w3.org/TR/2011/REC-CSS2-20110607/ (visited on 08/05/2014).

[al11b] World Wide Web Consortium (W3C) et al. *Scalable Vector Graphics (SVG) 1.1 (Second Edition) (W3C Recommendation)*. Aug. 16, 2011. URL: http://www.w3.org/TR/2011/REC-SVG11-20110816/ (visited on 08/06/2014).

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "D3: Data-Driven Documents". In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pp. 2301–2309.

[Int14] T. Bray from Internet Engineering Task Force. *The JavaScript Object Notation (JSON) Data Interchange Format*. Mar. 2014. URL: https://tools.ietf.org/html/rfc7159 (visited on 08/05/2014). Archived by WebCite®at http://www.webcitation.org/6RbJiw80d.

[Mic14] Microsoft. *SVG vs canvas: how to choose*. 2014. URL: http://msdn.microsoft.com/en-us/library/ie/gg193983.aspx (visited on 08/06/2014). Archived by WebCite® at http://www.webcitation.org/6RcvwTtT2.

[WW14] World Wide Web Consortium (W3C) and Web Hypertext Application Technology Working Group (WHATWG). *HTML5 (W3C Candidate Recommendation)*. July 31, 2014. URL: http://www.w3.org/TR/2014/CR-html5-20140731/single-page.html (visited on 08/05/2014).

List of Figures

List of Code Listings

Appendix A

CD-ROM Content

The CD-ROM attached to this thesis contains especially the following folders and files. Not all deep levels of the directory structure are listed.

- *application* . . . . . developed application
  - *release/* . . . . . esp. final code for release
    - *assets/* . . . . . compressed main client application files
    - *config.ini* . . . . . database credentials & server settings
    - *index.html* . . . . . main HTML file
    - *provis_server.py* . . . main server component
    - *README.md* . . . . . setup and startup instructions
  - *development/* . . . . . esp. raw code files
    - *assets/* . . . . . main client application files
      - *img/* . . . . . graphic files
      - *js/* . . . . . JavaScripts
      - *styles/* . . . . . SCSS style files
      - *templates/* . . . . Handlebars.js templates
    - *doc/* . . . . . JSDoc documentation website
    - *vendor/* . . . . . third-party JavaScripts
    - *Gruntfile.js* . . . . . Grunt tasks
    - *package.json* . . . . . Node.js dependencies & meta data
    - *prepare-database.sql* . PostgreSQL database preparation script
    - *setup-build.sh* . . . . Node.js setup assistance
    - *[…]* . . . . . further files analogous to *../release/*
- *thesis-provis-bettinger.pdf* . . . this thesis document

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Tübingen, 30. September 2014

Janek Bettinger