Ilija Šimić, BSc

VISUALIZER
An Extensible Dashboard for Personalized Visual Data Exploration

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Veas, Eduardo Enrique, Dr.techn. MSc

Institute of Interactive Systems and Data Science

Sabol, Vedran, Dipl.-Ing. Dr.techn.
Mutlu, Belgin, Dipl.-Ing.

Graz, January 2017

Kurzfassung

Das Analysieren von Datensätzen in Tabellenform gestaltet sich als sehr schwierig, sobald die Anzahl der Reihen und Spalten in einem Datensatz zu steigen beginnt. Um einen gewissen Datenverlauf erkennen zu können, werden typischerweise andere Methoden benutzt, als die Tabelle Zeile für Zeile zu durchsuchen. Eine solche Methode ist das Visualisieren einer aggregierten Teilmenge eines Datensatzes welcher für den User von Interesse ist. Es gibt bereits Programme auf dem Markt, welche Lösungen für dieses Problem bieten. Diese sind jedoch entweder einschüchternd komplex für neue User, bieten unzureichende Möglichkeiten für Programmerweiterungen, oder sind einfach zu teuer.
Deswegen wurde *Visualizer* entwickelt. Es handelt sich dabei um eine Webanwendung für clientseitige Datenvisualisierung. Bei der Entwicklung wurden unerfahrene und erfahrene User gleichermaßen berücksichtigt. Die Anwendung bietet dem User die Möglichkeit die Plattform anzupassen und zu erweitern. Zusätzlich kann Visualizer in bestehende Webseiten integriert werden und ist komplett steuerbar über die URL.

Schlüsselwörter: Visualizer, Visualisierungen, Datenvisualisierung, Webanwendung, Erweiterung, Anpassung, Personalisierung, Integration, Aggregation

Abstract

Analyzing datasets in tabular form quickly becomes difficult with a growing number of rows and columns in a dataset. In order to detect certain patterns in such datasets, going through the table line by line proves to be inefficient, which is why visual methods are often used for this purpose. One such method is the aggregation of a subset of interest of the dataset and its visualization in a chart. Off-the-shelf tools already offer the possibility to open datasets and display the contents for visual exploration, but they are either intimidating to new users, limited to a small number of visualizations and not extensible, or simply too expensive.
Visualizer was developed as a web application for client side data visualization which can be used as a stand-alone service or integrated into existing web-pages. It was created with non-expert and expert users in mind, and offers the options to extend the platform with new configurable charts and to fully customize and control it via the URL. For example, an expert user can create a layout consisting of different charts representing aspects of a dataset. This layout opens in other users‘ profiles to visualize their data.

Keywords: Visualizer, visualizations, data visualization, web application, extension, customization, personalization, integration, aggregation

Acknowledgment

First I would like to thank my co-supervisor Belgin Mutlu who was my first contact person regarding the thesis and who offered me great support and guidance whenever I was stuck somewhere and pushed me to finish this thesis much faster, than I would have without her. Furthermore, I would like to thank my supervisor Eduardo Veas and my other co-supervisor Vedran Sabol for their great ideas and time, which helped make *Visualizer* the way it is.
I am also thankful to my sister Marija and my brother in law Krešo, my brother Filip and my sister in law Tina for supporting me throughout my studies. In particular I express my gratitude to Marija and Krešo for helping me to settle in when I came to Graz. They were always there for me whenever I needed something and I am incredibly thankful for that. I would also like to thank all of my friends for all the fun we had together in the past years and for helping me relax when I needed it the most.
A special thanks goes to my fiancée Michaela, who supported me patiently all this long time it took me to finish my degree, and who made the great effort to proofread this long thesis.
My biggest gratitude goes to my mother Jela and to my father Ivo, who have always given me everything they have. Without them I would not have been able to study in Graz and write this thesis. Therefore, I would like to dedicate it to them.

Graz, January 18, 2018 Ilija Šimić

Za mamu i tatu

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is the aim of this thesis to introduce, describe and analyze *Visualizer*, an extensible web application for personalized visual data exploration. Being a web application, it is meant to be platform independent and accessible from everywhere. As a target group, non-experts and experts were considered equally. For the first group, features for assisted data visualization were integrated which only allow the user to select visualizations that are compatible with their fields of interest. For the second group, the experts, powerful features for data manipulation were added, and an application programming interface for the visualizations was defined, which allows the extension of the platform itself to suit the users' needs.
Analyzing data usually takes several steps of operations on the data in addition to creating visualizations. Therefore, once a user is satisfied with the "view" on the dataset showing certain characteristics, it is possible to save it in a URL. Using this URL with different datasets, which have the same structure as the one for which the URL was generated will apply all operations performed on the original dataset and create the same visualizations.

1.1 Motivation

With data collection becoming ubiquitous, the amount of data which can be analyzed skyrockets. However, data is not only being automatically generated, it is becoming also more common that regular users keep track of certain activities by writing them down into spreadsheets. Therefore, data analysis has become a task, which is not exclusive to data analysts anymore. For instance, a teacher has to analyze data if he wants to keep track of the progress of his students. When trying to analyze how the students performed in the previous semester, or how the whole class performed, it becomes difficult to discover patterns in tables. Visualizations can help with this problem, but many of the

available tools on the market require a particular skill set and expect certain knowledge from the user to be efficiently used. One often has to be partially an expert in data analysis and partially an expert in data visualization to create meaningful charts. In addition to the knowledge barrier, the tools either cannot be adjusted to the user, or lack the option to be extended with custom visualizations.
Therefore, *Visualizer* was developed as a free open-source web application, which assists beginners in visual data exploration and allows experts to fine tune their visualizations.

1.2 Focus Points

While developing *Visualizer*, the following aspects where in focus:

- **Wide target group**
  The tool should have all the functionalities to satisfy power users, yet have a mild learning curve and assisting features to guide users without knowledge in this area. A tool for data analysis should not exclusively target professionals who are ready to spend hours and hours learning to work with it and discover every possible functionality which the tool might offer, nor force users to go to lengths as accepting features which are disruptive, just because the tool might prove useful in other aspects.
- **Platform support**
  With the wide variety of today's operating systems and platforms, it is expected that anything runs on everything. Companies often have certain operating systems which users are not allowed to change; therefore, cross-platform support becomes even more important.
- **Focus on usability and guidance without disruption**
  The tool should be easily usable, with the access to all features, guide users if they desire so, and at the same time it must not be obnoxious for users who want to explore themselves. This issue is tightly coupled with the range of users.
- **Integration and extensibility**
  It should be possible to integrate the platform in another platform for usage, ideally with the option to control it. On the other hand, it should also be possible to extend the tool for custom functionalities.
- **Personalization**
  Every tool should be custom tailored to the user using it.

- **Price**
  An aspect often neglected, yet of utmost importance. If possible the tool should be easily available and free.

1.3 Structure of Work

In the second chapter a look will be taken into all the related scientific work regarding all the aspects of the application. This will include tables, dashboards, visualizations, interaction aspects, data provenance, visualization recommenders and guidelines for extension support. Additionally, an analysis will be done of scientific tools that influenced this thesis. Furthermore, the state of the art tools which are used in this domain of expertise will be investigated. The third chapter will, at first, give an overview of *Visualizer* and subsequently every implemented feature will be presented and thoroughly analyzed. The fourth chapter will cover the major applications for the created tool. In chapter five the performance and the limitations of the platform will be analyzed through a set of performed benchmarks. Chapter six will summarize the most important aspects and give an outlook for the next possible steps.

Chapter 2

Related Works

Similar to the structure of *Visualizer*, the related works section is also split into two theoretical parts. After the theory behind the dashboard has been covered, the current state of technology and the tools that influenced the thesis will be reviewed. The first subsection covers the topic about the *Dataset Table* in *Visualizer*. It is a means of initial presentation of the loaded data, a place to get a quick overview and transform the data. Therefore, this subsection will give more information about tables in general. The second subsection gives an overview on the background for the key components of *Visualizer*. It will touch on the topics of: dashboards, visualizations, interactions, information provenance, visualization recommenders and extensions. There, an overview will be given over different key components of *Visualizer*. The third subsection will focus on existing tools in this domain. The first part of the third subsection will cover scientific tools for data visualization which were taken either as a basis, or as metric for this thesis. The industry leaders in this field will then be presented and analyzed.

2.1 Data Tables

In his *Handbook of Biological Statistics* McDonald generally suggests, when given a choice, the usage of graphs for the purpose of displaying information and quick comparison of values in different categories [McDonald and of Delaware, 2009]. On the other hand, graphs obscure and aggregate data, thus when wanting to explore data in the greatest detail, a table is of better use.
When thinking about digital tables, the first thing which comes to most people's minds is Microsoft's Excel[1]. While being a great tool for tabular calcula-

[1]`https://products.office.com/de-at/excel`

tions and the creation of complex tables with its numerous different calculation and linking options, it is not designed for a simple and quick preview and exploration of big datasets. Features like filtering, data type detection, sorting columns, fast page based browsing, and simple column statistics are not easily accessible, or need multiple steps to be created. This is no criticism of the tool itself, because it is not designed for this purpose. It is merely an identification of the missing aspects of a table with the emphasis on fast data exploration. Google tried another approach with its table editor in Google Sheets[2]. They tried a platform independent approach, where the user creates and edits tables in the browser, keeping the features limited to the most important ones. Yet it still lacks the ability for a quick overview, review and simple manipulation of the data, with an emphasis on speed.
A different approach was taken by Chen et al. [Chen and Chung, 2004]. They presented the *TPS*, a **T**able **P**resentation **S**ystem, which allows the automatic generation of reports from databases and visual table transformation. In their work, Chen et al. focused mostly on the visual transformations of the tables, after the database has been queried. They defined a set of primitive and derived table operations. The primitive operations cannot be split into multiple operations. The derived operations on the other hand are just a fixed sequence of primitive operations. Chen et al. defined the following primitive operations:

- *Select:* returns a table with the given rows and columns
- *Merge:* merges a table into the given coordinates of the other table. Overlapping entries are aggregated
- *Contract:* merges a fixed number of rows or columns in a single table together
- *Expand:* expands a table by splitting entries into multiple cells
- *Match:* returns the rows or columns in a given order

The derived operations can be reproduced with the following sequences of primitive operations:

- *Flip:* Select operation with reverse selected rows and columns
- *Rotate:* Contract, Expand, Flip
- *Transpose:* Flip, Rotate
- *Resize:* Contract, Contract, Expand, Expand

To perform manipulations on a table a connection to a database is needed first. Afterwards, it is possible to select the tables of interest and perform the listed

[2]`https://docs.google.com/spreadsheets`

operations on them. Almost every operation has its specific GUI window fitted for the intended purpose. Furthermore, when done with the alterations on the tables, the user can create reports in multiple different formats for displaying or saving. With the *TPS*, Chen et al. showed that it is possible to do rather complicated table manipulations visually and without knowledge about querying the underlying database.
Visualizer's Dataset Table also allows multiple operations to be performed on the dataset with a cleaner UI and additionally offers powerful data aggregation methods and fast navigation inside the table. Furthermore, additional statistical information is computed about every column, which can be easily retrieved, to gain more insights into a column.

2.2 Visualization Dashboards

2.2.1 Design Considerations for Dashboards

According to Stephen Few, "a dashboard is a visual display of the most important information needed to achieve one or more objectives; consolidated and arranged on a single screen so the information can be monitored at a glance". In his book *Information Dashboard Design* [Few, 2006] he analyzed many visualization dashboards available at the time, for the purpose of categorization and the creation of recommendations for good design principles for dashboards. At first he pointed out the most critical aspects of dashboards.

- Dashboards display information needed to achieve specific objectives
- A dashboard fits on a single computer screen
- Currently the best medium for a dashboard is a web browser
- It should be possible to observe the information at a glance
- Dashboards have small, concise, clear and intuitive display mechanisms
- Dashboards are customizable

In contrast to his most critical aspects, Few also pointed out the thirteen greatest mistakes a designer can make when creating dashboards. Those thirteen mistakes can be easily compressed into the following ones:

- overabundance of information and content
- missing key features
- using multiple screens
- colorful dashboards elements and meaningless decoration

- not highlighting data

All of Few's most critical aspects mentioned were taken into consideration as much as possible while creating *Visualizer's* dashboard. While designing the dashboard component, simplicity and usability were the main focus. With its simplistic look, *Visualizer* has no abundance of information or color, yet every explorative or analytical feature is present and reachable with one click. All visualizations can be placed on a single screen and in case it becomes difficult to order them, there is an option of automatic ordering, which puts them all back into sight. Data is evidently presented in the visualizations, and important bits can be additionally highlighted. Furthermore, if the user is not satisfied with the visual appeal of the dashboard, the colors and the fonts can be freely customized to his liking.

2.2.2 Background Information on Visualizations

Jacques Bertin was the first to describe the components of what is known today as visualizations in his book *Semiology of Graphics* [Bertin, 2011]. Carpendale [Carpendale, 2003] took Bertin's work as a foundation and applied it to modern visualizations. When Bertin described the building blocks of graphics, his views derived from the field of cartography, taking into consideration that the graphics will be printed on paper and be mostly black and white. Carpendale on the other hand took Bertin's defined components and built on them for the purpose of displaying visualizations on a computer screen.
As a basic unit for displaying information, Carpendale used a mark which was previously defined by Bertin, representing a symbol in the visualization. The modifications and alterations which can be applied to a mark were called visual variables. Marks can be of the following types: Points (zero-dimensional), Lines (one-dimensional), Areas (two-dimensional), Surfaces (two-dimensional in three dimensional space) and Volumes (three-dimensional). As for visual variables, Bertin defined the following: Position, Size, Shape, Value (color value in the HSV color space), Color, Orientation and Texture. Carpendale added also motion to the visual variables, because on a computer, a mark does not have to be static like on a paper, but it can interactively be moved around. For the purpose of selecting the correct visual variables for the data, Bertin defined a list of characteristics which visual variables might possess. Thus, a visual variable can be:

- Selective - Changing a mark in this visual variable makes it more notable
- Associative - Changing marks in this visual variable can make them appear as a group

- Quantitative - Changing this visual variable of a mark, changes the perception of the value of the mark
- Order - changes the perception of order (greater, smaller)
- Length - defines how many changes in the visual mark are noticeable

With those definitions, it is much simpler to dissect visualizations into their constituent parts, create complete new visualizations tailored to data or choose the best representations for the selected data. Furthermore, a vocabulary was created, simplifying discussions between data analysts.

2.2.3 Identifying interactions

When talking about interactions in visualizations, it might prove as useful to have a categorization of the countless different possible interactions. Yi et al [Yi et al., 2007] focused on the interaction component of information visualization (Infovis) systems. They tried to analyze the different interaction methods in Infovis systems, find common aspects of them and categorize them. Firstly, they adopted the definition of interaction from Foley et al, which defined an interaction technique as a way of using a physical input/output device to perform a generic task in a human-computer dialogue [Foley, 1995]. Yi et al adapted this definition by saying that interaction techniques in Infovis systems were more designed for changing and adjusting visual representation than entering data into systems.
After analyzing existing taxonomies of Infovis systems and going through many commercial tools and papers regarding Infovis techniques, they identified 311 different interaction techniques, which they then categorized into seven different categories based on the user's intent to accomplish a certain task with a specific interaction technique. Those seven categories are as follows:

- *Select:* Highlight certain items. Rearrangement of highlighted items in the visualization keeps them in focus
- *Explore:* Show more items, based on a certain selection. Clicking on a node in a graph loads more items connected with this node.
- *Reconfigure:* Rearrange the items in the current visualization. For example: Sort by name
- *Encode:* Change the representation of the current selected data. For example: Change the visualization type
- *Abstract / Elaborate:* Display more or fewer items.
- *Filter:* Exclude certain items in the visualization

- *Connect:* Highlight the same item in another visualization (Brushing)

However, the authors did not succeed in assigning every collected Infovis technique to one of their defined categories, either because the interaction technique is used as an aid to compare data, or the interaction technique fits into multiple categories at once. They also considered adding additional techniques, like a compare technique, but decided against it, because they were higher level end goals of the visualization process and not directly interaction techniques, based on simple user intents.
Yi et. al's categorization based on user intent presents a suitable way of assigning the actions, which can be performed by a user in *Visualizer*, to categories, which makes it immediately clear, what the expected outcome from the user's perspective is.

2.2.4 Information Provenance

As *Visualizer* became functional, it became also clear that in order to arrive to a dashboard a non-trivial sequence of steps had to be carried out. It also became evident that most users would want to go straight to a view on the data, and even expert users would not be keen on finding out those steps or repeating them every time. A way to save the state of the dashboard was necessary. When thinking about saving the current state, for the most part, there are two ways to approach this task. Either save the altered version of the dataset, which completely removes all the steps applied to get from the original dataset to the transformed one, or you can only save the interactions done in the original dataset and apply those interactions the next time the dataset gets loaded. Saving the altered version of the data is an easy task, but saving the actions required to get from the original state to the desired required research. This process is not only about transforming the dataset, but going beyond it and saving interactions on the dashboard and the state of the visualizations. This research often leads to the terms information provenance or data provenance. Groth et al. presented a model for information provenance and annotation in visualization systems [Groth and Streefkerk, 2006]. This model was designed to simplify the way information about interactions in the discovery process of new insights is recorded. Furthermore, this model was designed as a directed graph. Each node of the graph represented a state of the visualization system and each edge an interaction which resulted in a transition from one state to another. Adding annotations to the graph was also made simple, by simply binding them to a state in this graph. The graph's structure also makes collaborations easy, because the graph can be forked. This model makes it possible to see how a user goes from an initial dataset to conclusions step by step. Another system for managing information provenance was presented

by Silva et al., namely VisTrails [Silva et al., 2007]. VisTrails uses an action based provenance model to track the user's interaction with the visualization system without the user noticing. Beyond simple collection of interactions on a visualization system, it additionally provides parameter exploration in the visualizations, which lets the user explore, group and display a set of parameters. What makes VisTrails stand out, is that the recorded workflow is not bound to a single dataset. It is also possible in VisTrails to compare different workflows. The main advantage of an action based provenance system is that only the actions have to be saved and not the whole states of the system.
According to Ragan et al. who categorized and described provenance models and their purpose [D. Ragan et al., 2015], the relevant provenance information for this work were Data and Visualization provenance. Data provenance describes the changes of the underlying dataset, the transformations which led to the state of data before the visualization was realized. Visualization provenance, on the other hand, saves the displayed visualizations and their state. In *Visualizer's* case this would be the sorting of the data in the visualization and the active brush. As for the purpose of provenance, according to the model of Ragan et al. replication, collaborative communication and presentation were relevant for *Visualizer*. Replication allows to save the current state of work and continue the work. Presentation, similar to replication, but with the purpose of showing the discovery. Furthermore, collaborative communication to share the insights with other users.

2.2.5 Recommending Visualizations

Since *Visualizer* does not contain one, but two different recommenders, it is important to give an overview over the different types of visualization recommendation systems. The recently released paper of Kaur et al. [Kaur and Owonibi, 2017] deals exactly with this topic. Kaur et al. classifies the different visualization recommenders into four categories.

- *Data Characteristics Oriented:* Visualizations are recommended based on the used data
- *Task Oriented:* Recommendations are based on user's expected outcome
- *Domain Knowledge Oriented:* More for support. Share knowledge about recommended visualizations
- *User Preferences Oriented:* User actions are tracked and visualizations recommended upon comparison with users who did similar actions

The data characteristics oriented and the user preferences oriented approach will be explained in more detail, since the two visualization recommenders

used in *Visualizer* are based on them.
According to Kaur et al., the focus of data characteristics oriented recommenders lies on "the definition of new data dimensions or attributes, the formalization of the process of visual mapping from data attributes to visual marks, and the introduction of new techniques for visual mapping" [Kaur and Owonibi, 2017, p. 267]. Basically, the recommenders in this category react based on a ruleset. If the given selected fields fit the conditions of a visualization, the visualization gets recommended. The most prominent recommender in this category is the *Show Me* module from Tableau, which inspired the *VisPicker* in *Visualizer* and which will be covered separately in 3.5.1.3. User preferences oriented recommenders recommend the visualizations based on the perceived intentions of a user. These types of visualization recommenders are very scarce, because it is a relatively new research topic. There are different methods for this, of which one would be machine and probabilistic learning. The other, more relevant for this thesis, would be, monitoring the actions of a user, comparing them to other user recorded actions with similar behavior and based on their outcome, recommending certain visualizations. Mutlu et. al [Mutlu et al., 2016] proposed such a recommender, which is based on selected fields and user added tags. This recommender was also used as a service in *Visualizer* and will also be explained in more detail in subsection 3.5.2.4.

ShowMe

The *Show Me* [Mackinlay et al., 2007] module in Tableau was the inspiration which lead to the creation of the *VisPicker* component in *Visualizer* and which greatly influenced the interaction process for the creation of visualizations in *Visualizer*. When creating *Show Me*, Mackinlay et al. wanted to lower the entry barrier for new users in Tableau, and improve the speed of data analysis for skilled users. The idea was to let the user decide what he wants to show and that he does not necessarily need to know how. *Show Me* heavily relies on VizQL [Hanrahan, 2006] for decision making, which is the specification language for the visualizations created for Polaris that later on became Tableau. It is based on Bertin's definition of components of visualizations and an algebra used in APT [Mackinlay, 1986], which was '**A** **P**resentation **T**ool' for the automatic generation of visualizations. The algebra from APT was extended in VizQL and it proved to be an effective tool for describing visualizations. *Show Me* focuses mostly on the fields which are dropped on the rows and columns in Tableau, because they get assigned to the axes and headers of the table pane. The table panes in Tableau represent nested axes. Nested axes are used in Tableau to display multiple views within one visualization. For example a bar chart displaying the yearly income of markets gets split into twelve displays of monthly incomes in the same visualization. As a navigation element, *Show Me* uses a small window with all the supported visualizations, which can be seen in Figure 2.1.

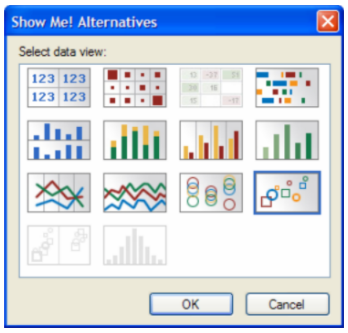


Figure 2.1: Show Me! Alternatives dialog [Mackinlay et al., 2007, p. 6]

Show Me actually supports two variations for suggestions. One is when a visualization gets automatically created from the preselected fields and the other gives recommendations for visualizations compatible with the selection. When selecting multiple fields, *Show Me* allows only compatible charts to be generated. The charts displayed in *Show Me* also have a ranking, which depends on the selected fields. For example, when selecting only one categorical field a table will be suggested at first. Selecting another quantitative field will always favor a bar chart in comparison to the other visualizations. Thus, automatically creating a visualization, will always create the one ranked the highest. This can be seen in the *Show Me* alternatives dialog. The ranking of the visualizations is following a ruleset determined by Mackinlay et al. by monitoring experts on their usage of visualizations with fields of specific types. *Show Me* has one additional functionality, which is called *Add to sheet* and which automatically updates an already existing visualization, when an additional field has been selected.
To this day, *Show Me* is still a part of Tableau, which means that it proved to be a useful tool, thus the incentive to create something based on this became even stronger.

VizRec

With *VizRec*, [Mutlu et al., 2016] Mutlu et al. presented a recommender system for visualizations. *VizRec* expects a query in a free-form text as an input and responds with a ranked list of visualizations. The generation procedure of personalized visualizations is split into three steps. First, the data passed to *VizRec* is preprocessed in a common data format. The preprocessing includes metadata extraction, data type categorization, semantic extraction and semantic enrichment. Second, the preprocessed data goes through the process of visual mapping. According to Mutlu et al., this can be seen as a schema matching problem [Rahm and Bernstein, 2001]. Expressed in simple terms, the visual mapping process tries to find matches of data fields to compatible visual channels of a visualization, thus generating a list of visualizations which can be used with this data. The third step for generating personalized visualizations is the filtering of those compatible visualizations. Mutlu et al. used three approaches to tackle this problem:

1. Collaborative Filtering
2. Content Based Filtering
3. Hybrid Filtering

The collaborative filtering in *VizRec* is based on user preferences expressed as a rating. Users rate visualizations according to the satisfaction of the outcome of a used visualization. When a recommendation for a new user has to be generated, the user is compared to a similar user and based on the ratings of similar users the recommended visualizations are ranked. Content Based Filtering in contrast to Collaborative Filtering, needs information on the user's interest before the recommendations come. The user enters a number of tags, which represent his interest in certain topics, and gets recommendations based on those tags. The third approach, Hybrid Filtering, utilizes both filtering methods. The tags are entered before the recommendations are given and the recommended visualizations are rated, which further improves the recommender.
In *Visualizer*, *VizRec* was integrated as a service. The first two steps in the generation of personalized visualizations is covered by *Visualizer*, and when called, *VizRec* gets passed the mapping of the fields to the channels of the visualizations and additionally is fed with the tags and ratings of the visualizations, returning a ranked list of recommendations.

2.2.6 Extension guidelines

Incorporating an extension interface in applications has become the norm today. When looking at frequently used programs, many of them offer a way to extend their functionalities. There are many names for them, plug-ins, add-ins, add-ons or modules, but all of them extend the core software with new features. The web browser Firefox supports extensions since its version 0.1

in 2004[3] and Chrome since its release in 2008[4]. Even Microsoft's Office suite allows the addition of extensions and the email management program thunderbird too. Integrated Development Environments like Eclipse, Visual Studio or IntelliJ have also means to extend their basic functionalities. Offering an extension interface is a good way to offer the users who like the software to add additional features if they are not satisfied with the current ones. As can be seen in *Show Me*, extensions can even be added to the core software. Aly et al. [Aly et al., 2012] notes that extensions can be integrated into software on multiple layers. For example, an extension which adds changes to the GUI does not necessarily have to do changes to the data. Therefore, extensions should have some restrictions. Aly et al. presented a list of requirements which should be followed for better extension integration.

- *Controlled Visibility:* To what degree have the developers access to the codebase
- *Controlled Extensibility:* Limit extensions to a certain scope of the application.
- *Stable Contract:* The extensions are written separately from the core software
- *Support for extensions on multiple layers:* It should be possible to add extensions for every aspect of the application
- *Composition Approach:* It should be possible to integrate extensions into the main application
- *Invasiveness:* It should not be necessary to make changes to the main application to guarantee that a single extension works
- *Multiple extensions:* It should be possible to add multiple extensions to the application
- *Simplified consumption of the extensibility model:* Make it easy to write and integrate extensions

Following the rules given in this list of requirements it becomes much simpler to design an appropriate extension interface for user created visualizations.

[3]`http://website-archive.mozilla.org/www.mozilla.org/firefox_releasenotes/en-US/firefox/releases/0.1.html`

[4]`https://chromereleases.googleblog.com/2008/09/`

2.3 Data Visualization and Recommendation Tools

The analyzed tools were split into two categories. The first one are scientific tools which had the greatest impact and influence in the design process of *Visualizer* or which proposed ideas which can be incorporated into future work. The second category analyzes the leaders in the commercial world.

2.3.1 Scientific

In the following subsection a look will be taken at some notable data exploration and visualization tools created as parts of scientific projects. The Visualisation Wizard [Mutlu et al., 2013] was selected because it was the basis of the *Visualizer* project. SeeDB was selected because of its approach in selecting which visualizations might be considered as interesting. Lastly, Polaris was analyzed because it was the predecessor of Tableau.

CODE - Visualisation Wizard

The Visualisation Wizard (Vis Wizard) [Mutlu et al., 2013] as part of the CODE platform[5] is a visualization recommendation and data exploration tool and can be seen in figure 2.2. As an input, the platform expects data in a Linked Open Data (LOD) format described with the RDF[6] data cube vocabulary. The data is then analyzed and compatible visualizations are suggested. Multiple visualizations can be created and the data further explored. As mentioned, the Vis Wizard encompasses two main functionalities, which will be discussed in more detail.

[5]`https://code.know-center.tugraz.at`

[6]Resource Description Framework

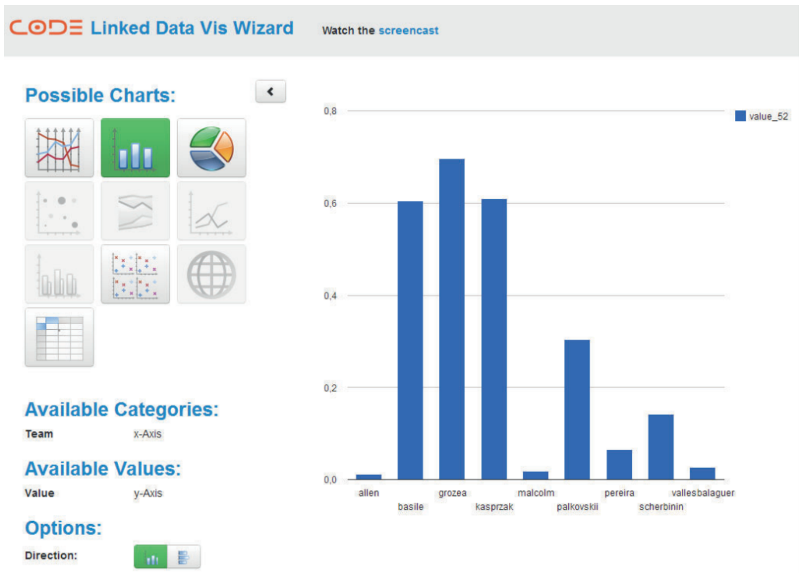


Figure 2.2: CODE - Vis Wizard [Mutlu et al., 2013, p. 43, Fig. 1.]

Visualization Recommendation

In order to suggest data, first a description of the data and the visualizations is needed. For this, Mutlu. et al. created a Visual Analytics Vocabulary (VA Vocabulary) representing descriptions of visualizations, which contain only meaningful attributes for the creation of valid visualizations from the given data. The VA Vocabulary is a semantic description of the visualizations defined with an OWL[7] ontology. The most important part of this description is that each visualization was split into the respective visual channels as defined by Bertin. Additionally, each visual channel has been assigned properties which have to be fulfilled as a requirement for data to be mappable unto it. Those properties are: (i) Occurrence - meaning how often data can be mapped to a channel, (ii) Persistence - meaning if the channel has to have data mapped unto it (an optional channel) and (iii) compatible datatypes. For the datatype Mutlu et al. first split the data into categorical and numerical data. Categorical data represents data on which no mathematical operations can be performed. Numerical data on the other hand supports computations. Furthermore, categorical data was split into *String*, *Date* and *Location* and numerical data into *Integer* and *Number* (float). For example, a bar chart consists of two visual channels: one x-axis and one y-axis. The data has to be mapped on both of those channels so that the bar chart can be created. However, it is not possible to map any kind of data to any channel. The x-axis supports only categorical data and the y-axis has to be numerical. Other

[7]Web Ontology Language

visualizations can be described in a similar fashion.
In order make suggestions, the fields in the RDF data are categorized into categorical and numerical fields. Now for all fields, it is checked, if a combination exists which can be mapped correctly on the visual channels of a visualization. If such a combination exists, the visualization is activated in the user interface.

Data Exploration
After the data has been checked and the visualization suggested, the data can be explored. Multiple visualizations can be created from one dataset. In order to assist the user in the exploration, the data can be aggregated in different ways before a visualization is displayed. For example, when creating a bar chart, one can choose to average the numerical values. Additionally to data aggregation, the visualizations are connected with each other i.e. selecting certain data points in one visualization will highlight the appropriate data points in other visualizations.

By defining the VA Vocabulary and the rule based suggestions of visualizations for data which gets analyzed as a whole, Mutlu et al. presented a new way of data exploration without the need of knowledge about the structure of visualizations. The user only needs to load a dataset into the Vis Wizard and as a result will be presented with valid visualizations, which can then be explored and analyzed.

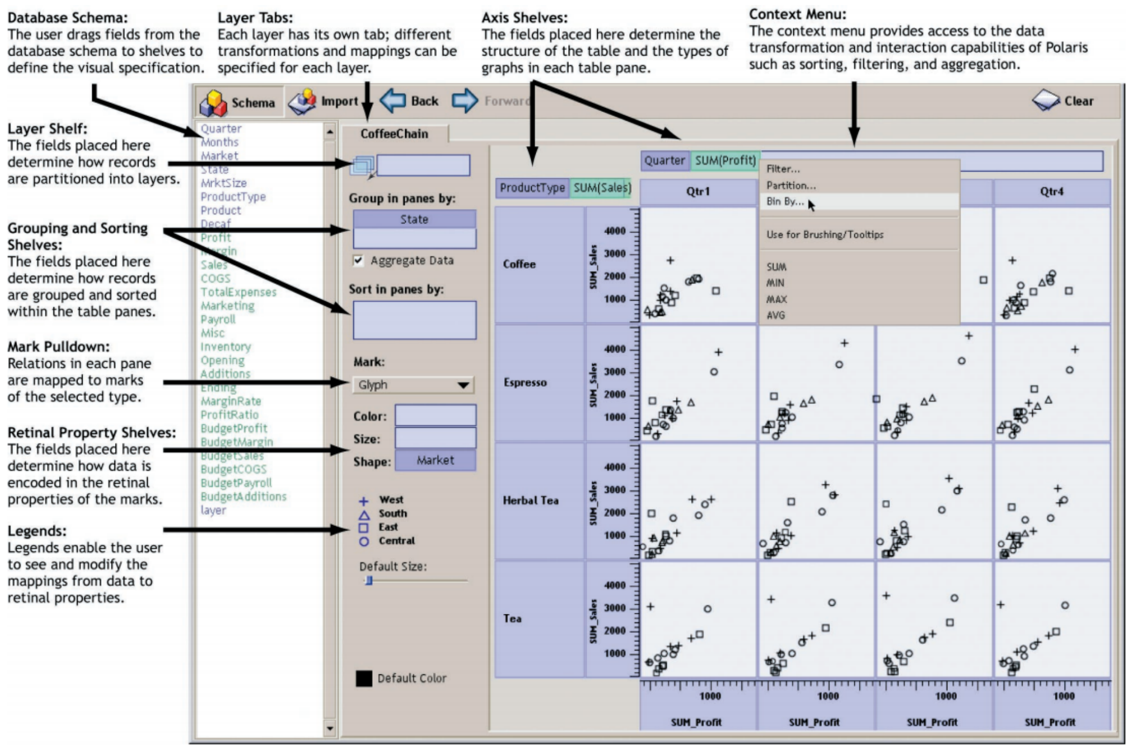
SeeDB

With SeeDB, Vartak et al. [Vartak et al., 2014] presented a tool for the automatic generation and suggestion of visualizations, given an input query to a database. First, they split the typical workflow of a data analyst into the following 4 steps: (i) use a query to get data from the database, (ii) generate different visualizations with the retrieved data, (iii) select “interesting” visualizations from all the generated ones, (iv) share results or start anew. Subsequently, they tried to automate the most time intensive steps, namely steps (ii) and (iii). In order to achieve this, they created SeeDB, which first finds all possible visualizations that can be generated by the results of the query and then compares how much each visualization deviates from the visualization generated from the whole dataset. Vartak et al. consider visualizations with a higher deviation as more "interesting", because that shows an abnormality in the data. SeeDB does not operate as a standalone application. Instead, it functions as a wrapper over an existing database. Thus, the application is split in two components: a frontend and a backend. The frontend functions as a thin client, whose only tasks are: (i) taking the queries and transferring

them to the backend and (ii) presenting the results. The backend receives the query, processes it and generates multiple optimized queries, which are run on the database. The backend then generates views from the results of those optimized queries. Views are not visualizations, they are only descriptions of a visualization. Those views are then compared to the view of the entire dataset. After the inspection, the top k views with the greatest deviation are sent back to the frontend, which generates and displays visualizations from them.
With its interest based filtering of visualizations SeeDB presents a very interesting and novel approach to assist the user in data exploration by recommending the most interesting visualizations for the selected fields.

Polaris

Stolte et al. [Stolte and Hanrahan, 2000] showed with Polaris a tool designed for the visual exploration of data in databases. The data in the databases is organized in tables. In Polaris, the rows of the tables are referenced as tuples, and the columns as fields. Stolte et. al characterized the fields in a table as ordinal and quantitative and additionally partitioned the fields into dimensions and measures. So ordinal fields get treated as dimensions, and quantitative fields as measures. Stolte et al. tried to address the following three points with Polaris: (i) Show huge amounts of information from the database on the display (ii) Display the information in different ways (iii) Allow exploration of the displayed data. A detailed description of the user interface can be seen in figure 2.3.



Database Schema: The user drags fields from the database schema to shelves to define the visual specification.

Layer Tabs: Each layer has its own tab; different transformations and mappings can be specified for each layer.

Axis Shelves: The fields placed here determine the structure of the table and the types of graphs in each table pane.

Context Menu: The context menu provides access to the data transformation and interaction capabilities of Polaris such as sorting, filtering, and aggregation.

Layer Shelf: The fields placed here determine how records are partitioned into layers.

Grouping and Sorting Shelves: The fields placed here determine how records are grouped and sorted within the table panes.

Mark Pulldown: Relations in each pane are mapped to marks of the selected type.

Retinal Property Shelves: The fields placed here determine how data is encoded in the retinal properties of the marks.

Legends: Legends enable the user to see and modify the mappings from data to retinal properties.

Figure 2.3: Polaris: User interface description [Stolte and Hanrahan, 2000, p. 54, Fig. 1.]

The user interface of Polaris displays only the field names, which allows the user to create visualizations by dragging and dropping the fields on the shelves of the visualization window. The state of the currently dropped fields on the different shelves is called a visual specification. The following actions are recorded in a visual specification:

- Data sources mapped to the layer
- Number of rows, columns and layers, and their order
- The current selection of records from the database
- Grouping and computations on the data
- Selected visualization
- The mapping of the fields to the retinal properties

The fields can be directly assigned to one of the available retinal variables (shape, size, orientation, color) as defined by Bertin, with the exception of texture. Additionally, each visual specification consists of the following three

parts: (i) the recorded actions (ii) the used graphics, and (iii) the visual encoding. Furthermore, Polaris provides methods for data transformation. Those methods include: aggregation, counting of ordinal dimensions, discrete partitioning (binning and partitioning), ad hoc grouping and threshold aggregation. Moreover, the data can also be sorted and filtered. While using Polaris, the user does not have to be afraid of making mistakes while exploring the dataset, because the integrated undo and redo functionalities allow a simple correction of the mistakes. Polaris also supports brushing.
Polaris allows the user to freely explore the dataset and to even perform transformation directly on the data, which can give even more insights into the dataset. With Polaris Stolte et al. clearly succeeded in creating a great tool for data exploration, as later on Polaris became the top market leader with the new name Tableau.

2.3.2 Commercial

According to Gartner's Magic Quadrant for Business Intelligence and Analytics Platforms[8], these are the three most popular and successful visual analytics tools:

- Tableau
- Microsoft
- QLIK

There exist different versions of all these three tools, each of which is tailored for a specific group of users. These versions differ from each other in terms of allowing, creating and sharing visualizations, collaboration between users, and if they are cloud-based or just directly installed on a machine. Thus, these products also have different prices.
For my analysis, I focused on those which focus on the creation of dashboards and data exploration, while keeping the costs as low as possible. Therefore, the visualization platforms I will analyze in the following section are:

- Tableau Desktop (Professional Edition)
- Microsoft Power BI Desktop
- QLIK Sense

Each of the tools were analyzed with the following aspects in mind:

- Price

[8]https://www.gartner.com/doc/reprints?id=1-3TYE0CD&ct=170221&st=sb

- Supported platforms
- Supported data sources
- Data manipulation
- Personalization
- Extensibility
- Usability

Tableau Desktop - Professional Edition

Tableau, the market leader, offers a rich variety of business intelligence solutions. For instance, Tableau Desktop is a powerful tool for creation of interactive dashboards, defining stories for visual data exploration and presentation of prepared visualizations. Moreover, Tableau Desktop is a prerequisite for adding new data sources to the dashboards in other Tableau versions. In Tableau Server the administrator is required to add a data source to the system so that other users might use it in their dashboards.
Tableau Desktop (Professional Edition) supports a great variety of file types (Access, Excel, text files, statistical files, etc.), databases (PostgreSQL, MySQL, Oracle, etc.), servers and services (Amazon Redshift, Microsoft SQL Server, Google Analytics, etc.) from which it can extract data. Once the data is extracted from a data source it can be transformed into a specific format which can then be applied to the visualizations. In the dashboard the individual data attributes can be filtered, sorted, merged, split or even completely deleted. Each data attribute is categorized into two groups: Dimensions and Measures. Dimensions represent categorical data we cannot perform arithmetic operations on. Measures, however, represent numerical values we can aggregate, measure etc.
In its explorative task, Tableau Desktop supports the user with a friendly user interface, which can be seen in figure 2.4. It has an intuitive way of structuring data with abundant drag and drop support and samples, and tutorials for the absolute beginners. It is a compelling tool with a wide assortment of options for modifications of the presented data.

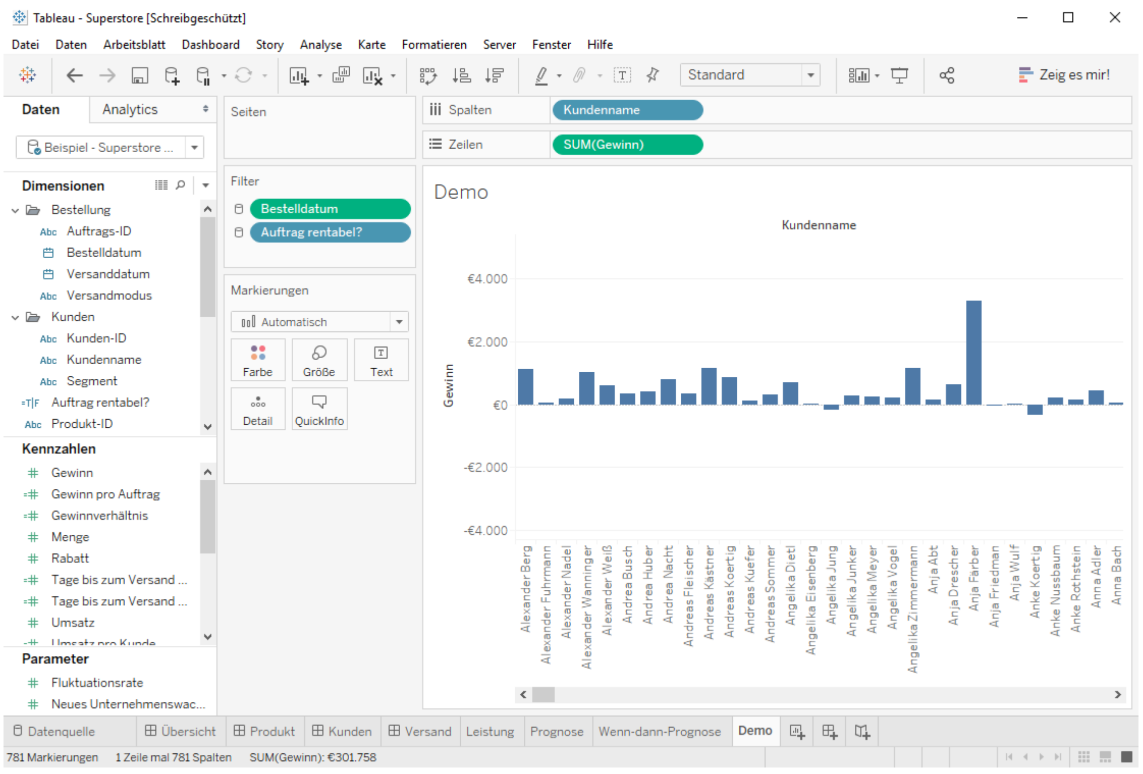


Figure 2.4: Tableau Desktop: Creating a visualization

One of Tableau's greatest strengths lies in its Show Me feature. It supports the user in the selection of visualizations which are compatible with the selected attributes. It also enables changing the visualization to another one which takes the same attributes. An alternative way of creating a visualization is to directly drag and drop the attributes of interest and map them onto the visual channels of the visualizations. This requires a certain amount of knowledge about visualizations. The created visualizations can be then edited in different ways until the user is happy with them. This includes changing the colors, selecting subsets of data in the visualization, sorting, renaming labels and resizing the visualizations.

Another useful feature for the data analysis aspect of Tableau is the possibility to enrich the visualizations with trendlines, forecasts or to summarize the data via a boxplot, which makes it easier to detect certain patterns in the data. Another interesting feature of Tableau is the coordinated view: Interactive changes made in one visualization are automatically reflected in the other ones.

For presentation purposes, Tableau includes a story mode allowing the user to create a presentation with multiple dashboards. Furthermore, each dash-

board can be annotated to highlight notable points in the data.
To sum up, Tableau is a well-designed program for visual analytics and data exploration. It is fast, snappy, easy to use. However, it also has some drawbacks. For instance, when the visualizations become more complicated, it becomes difficult for the user to keep track of the attributes which were selected before. Moreover, Tableau is only available for Windows and Mac. The biggest drawback of Tableau, however, is that it lacks extensibility meaning that it is unable to include new visualizations in an easy and intuitive way. Moreover, Tableau does not fulfill the most important requirement for dashboard design according to Few, namely, that a dashboard has to fit in a single screen. This is not the case in Tableau, because it assigns each visualization to its respective screen, and only when creating a special dashboard called a story, it can show a combination of all.

Microsoft Power BI Desktop

Power BI is Microsoft's tool for data visualization and data exploration. It offers a familiar design with its top ribbon navigation elements for users who are already used to working with Microsoft Windows and Microsoft's Office suite. There are three different versions of Power BI: Power BI Desktop, Power BI Pro and Power BI Premium. Power BI Desktop, which can be seen in figure 2.5, was chosen for this analysis because Power BI Pro and Power BI Premium, in addition to the features of Power BI Desktop, focus more on cloud based data storage and user collaboration. Furthermore, Power BI Desktop is completely free in contrast with the other two.

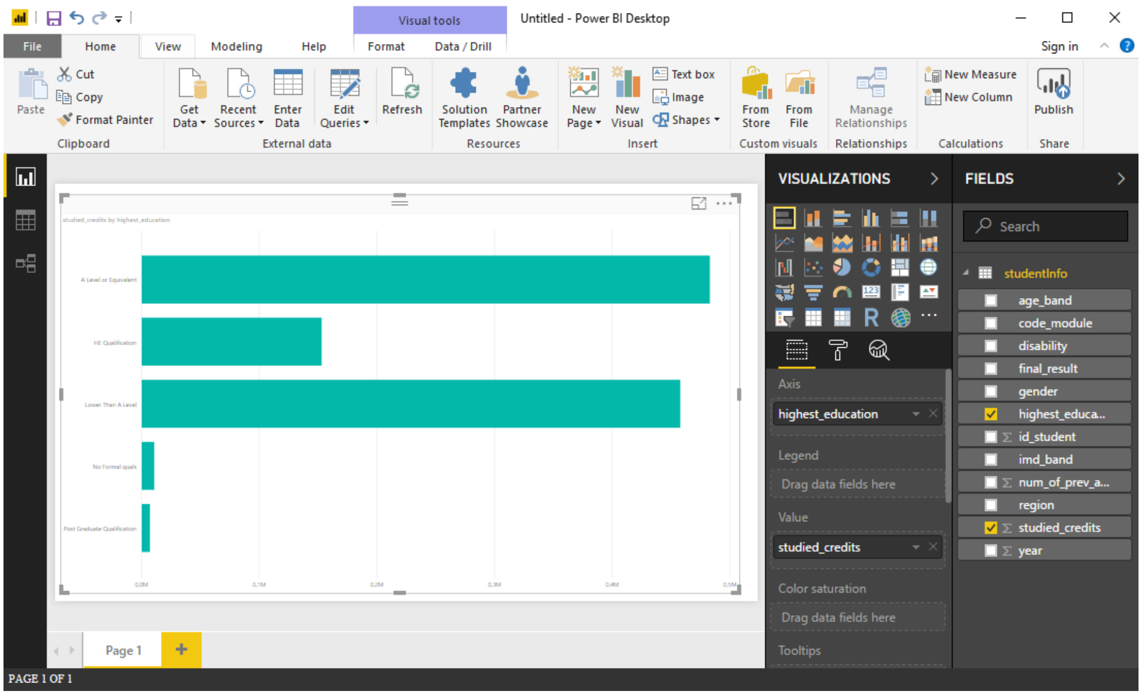


Figure 2.5: Power BI Desktop: Creating a visualization

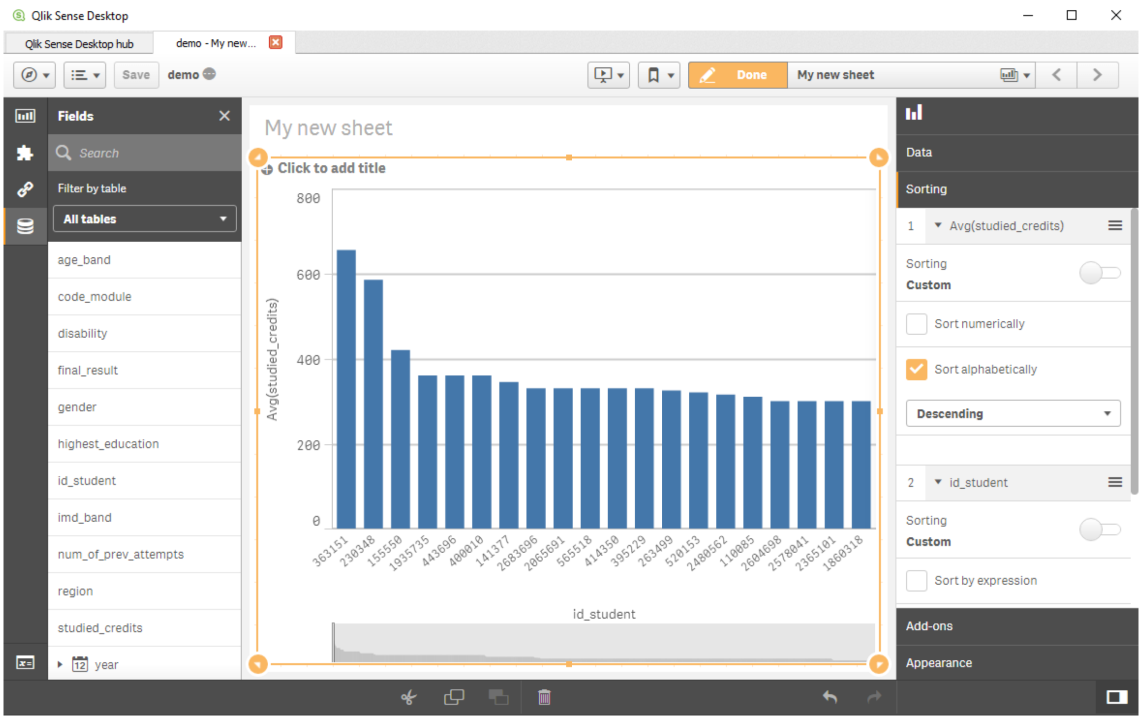
Power BI does not support many static files for data import, but covers all of the mostly used ones (Excel, CSV, XML, JSON). Otherwise, it supports many databases (MySQL, Oracle, Access DB, PostgreSQL, etc.), online services (Azure Enterprise, Google Analytics, GitHub, Facebook, etc.) and other data sources (HDFS, Spark, ODBC, R-Scripts, etc.).
After the data has been loaded, it can be prepared before the data visualization is created. Power BI offers a very powerful editor. The most important features of the editor are that (i) columns can be renamed, removed, merged or split, (ii) rows removed, (iii) data types automatically detected, (iv) values grouped (v) statistics about the data calculated (vi) values filtered (vii) values in columns replaced (viii) r-scripts run on the data (ix) new tables created (x) and features undone. Therefore, Power BI offers a solution even when the data is not properly loaded.
In the dashboard, all the attributes are displayed in alphabetical order next to the buttons for the creation of the visualizations. There are basically two ways to create a visualization: You can select the attributes of interest and then click a visualization button, or you can create an empty visualization by clicking the button and then drag the attributes to the desired channels. The main disadvantage of Power BI here is that it will not suggest different visualizations for different types of attributes, but instead it will take the last used

visualization (in case of the first visualization it will be a table) and will try to fit in all the attributes even if the created visualization does not make sense. However, after a visualization has been created, it is very customizable. Almost every aspect of the created visualization can be changed separately, like label size, font, color, toggling axis, background, border or line widths. The visualization buttons themselves have no descriptions of their requirements, so existing knowledge about how each visualization is structured and how they work is required and therefore, Power BI is not well suited for non-expert users.
Linking and Brushing is only applied at single data points and not regions, therefore it is not possible to highlight one area in a visualization and get a mapping to the same data points in another.
Power BI offers its own developer tools, which, coupled with NodeJS, offer the possibility for users to write their own customized visualizations.
Overall, Power BI is a powerful tool for data analysts, which offers many aspects expert users might desire, such as a wide variety of supported data sources, running R-scripts, customizations and extensibility. On the other hand, its user interface with its crude visualization windows, makes it difficult to find properties and the lack of user support renders it hard for non-experts to use the tool. Furthermore, it is only available for Microsoft Windows.

Qlik Sense (Desktop)

Qlik also offers multiple versions of its visualization tool. There are cloud and desktop based versions for single users, which are free, and then there are the same versions with additional support for enterprises and businesses which are paid per user/license. Qlik Sense Desktop was chosen because it is more powerful than the cloud version concerning the manipulation of the data and the customization of visualizations and unlike business solutions, it is free.
In order to be able to use Qlik Sense the user has to first create an account on the Qlik website and when starting the desktop application, a login is required.
Even though Qlik offers most of the common connections to data sources (ODBC, Microsoft SQL Server, PostgreSQL, Oracle), it does not support out of the box certain online services which are also used widely (Amazon Redshift, Google Analytics). However, it is still possible to use those services, after additionally installing the appropriate connectors. Regarding static file types, Qlik covers all of the mostly used table files (excel, csv, xml, etc.) and even very specific ones (kml, fix, dat, etc.)
When editing the loaded data, Qlik offers a very weak data type detection. It

only distinguishes between 'general', 'date', 'timestamp' and 'geo data'. Dimensions and measures have to be split manually when creating the visualizations. However, a very useful feature, while editing the loaded data, are the column statistics. When selecting a column in the table, Qlik will display a bar chart with the number of occurrences of each value. For columns with numerical values it is additionally possible to display an axis with marked values for the minimum, maximum, average and median. Again, the user has to change the column manually to a measure to see those statistics.
The possibilities of editing the table are very limited. It is possible to replace values, and split columns, but merging 2 columns is cumbersome. The user has to create a new column and define with a function, how the 2 columns should be merged. Filtering and removing rows is not possible either in the data editor, but has to be done with another tool, prior to importing the file in Qlik Sense Desktop.
In order to create a visualization Qlik Desktop uses an approach, which has separate modes for inspecting the visualizations and creating them. The dashboards are split into sheets and opening a sheet displays the contents in the viewing mode of the dashboard. Enabling the editing mode, seen in figure 2.6, Qlik offers the possibility to create visualizations on the dashboard.



1 Avg(studied_credits)
Sorting: Custom
Sort numerically ☐
Sort alphabetically ✓
Descending
2 id_student
Sorting: Custom
Sort by expression ☐

Figure 2.6: Qlik Sense: Edit mode

Creating a visualization is very simple; the desired visualization just needs to be dragged to the dashboard and the needed attributes have to be added. However, if the attributes are not correctly selected, the visualization will still be created, but wrongly. For example, it is possible to add two categorical attributes to a bar chart which just produces an empty bar chart. That means the user has to have knowledge about the structure of the visualizations and data. However, with a small amount of knowledge it is possible to produce an entire dashboard with visualizations very fast. A significant drawback of the tool is that it is not possible to select some attributes and get suggestions of which visualizations they can be used with. Therefore, it is not well suited for non-expert users.
The produced visualizations look appealing and are fully customizable concerning colors, label, orientation etc. Furthermore, the created visualizations are linked with each other; thus a selection in one visualization is also highlighted in others.
In conclusion, Qlik Sense is an impressive tool for the creation of visualization dashboards. However, due to the lack of suggestions it is not suitable for entry level users, which restricts its usability. The design of Qlik sense can be adapted to a user's preferences via themes and the option to write one's own extensions (visualizations) gives developers the opportunity to include visualizations tailored to their needs. Additionally, Qlik Sense is free, which increases the likelihood of users trying it out.

2.4 Summary

At first this chapter gave an overview over existing works regarding every component of *Visualizer*. After the analysis of the key components and explanations of the vocabulary, different visualization tools were investigated. Scientific tools - which had novel features at the time when they were developed - were analyzed and following that, the most popular commercial tools were examined. Having covered all of those topics in this chapter, the next chapter will deal with the main aspect of this thesis.

Chapter 3

Visualizer

In this chapter the functionalities and implementation details of this project will be presented in detail. In the first subsection, an overview over the whole web application will be given, explaining what it is and what it does. After that the used technologies and the structure of the web application will be presented. In this section, the architecture of the web application will be outlined, and the responsibilities of each component explained. Furthermore, the back end functionalities will be analyzed in more detail. Since the focus of *Visualizer* lies on the front end, a whole subsection will be dedicated to this aspect, dividing it into two individual components, one for pure data manipulation and the other for data presentation. Furthermore, the data presentation section will be divided into two parts, one investigating the features which are available for all guest users and the second one outlining the additional personalized functionalities of logged in users. The dataset used for all the visualizations in this section is the dataset for real estate transactions in the Sacramento area[1]. This dataset was chosen, because it covers almost all data types supported by *Visualizer*, and it has a great variation in the column entries.

3.1 Overview

Visualizer is a customizable, extendable and personalized web application for data manipulation and interactive visual data exploration. It runs in a web browser, hence it is platform independent. The data which can be loaded in *Visualizer* is expected to be imported as a Comma-Separated Values file, also called CSV file. The CSV file can be loaded either locally (from the machine *Visualizer* is run) or remotely (from an arbitrary web server). Figure 3.1 depicts the starting screen of *Visualizer* with the two selection options.

[1] `http://samplecsvs.s3.amazonaws.com/Sacramentorealestatetransactions.csv`

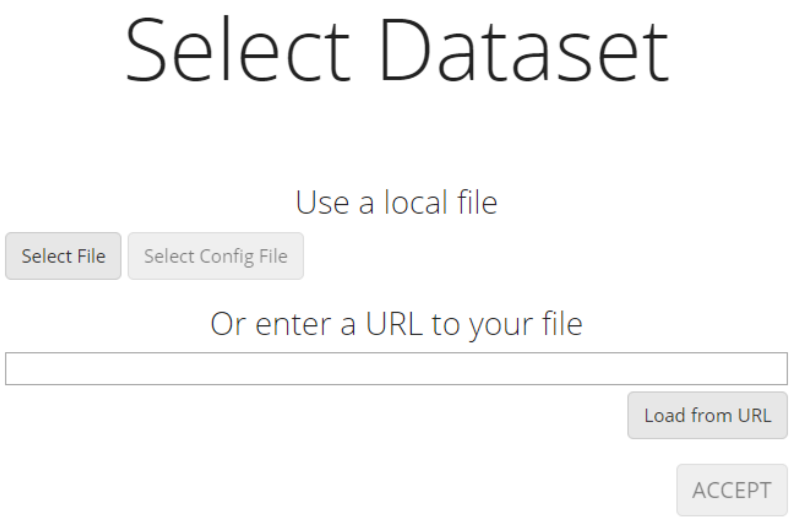
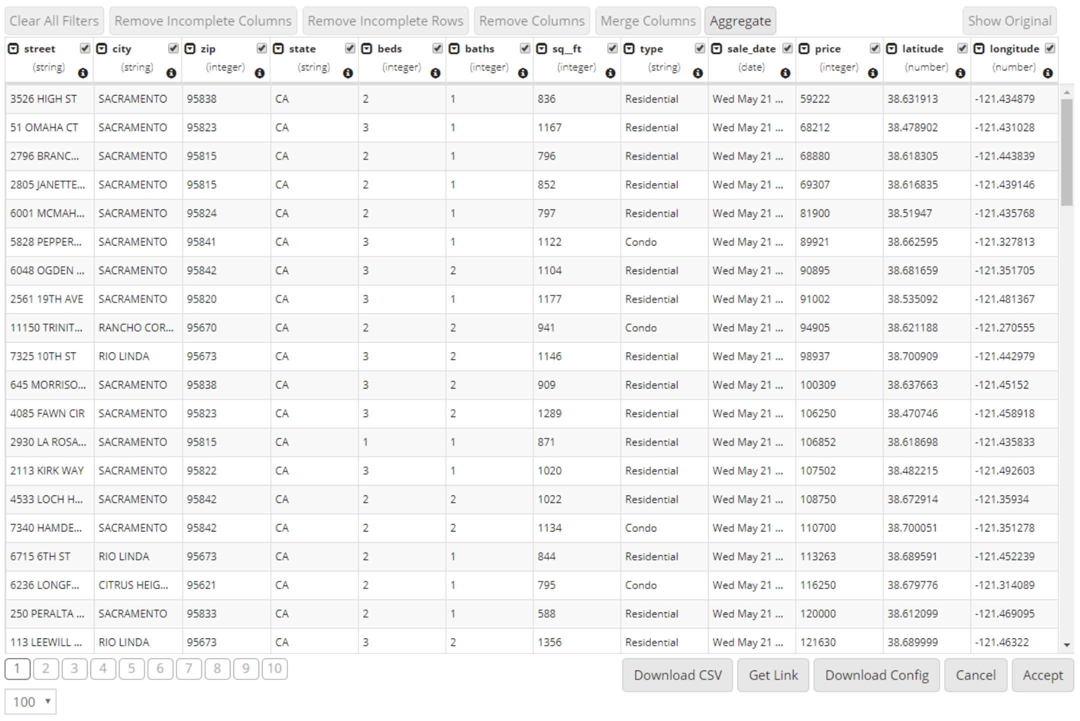


Figure 3.1: File selection screen of *Visualizer*

As soon as the data has been imported into *Visualizer*, the user is presented with the *Dataset Table*. This is the first of the two major components. The *Dataset Table* displays the data from the CSV file as a table split into multiple pages in order to ensure loading speed and it offers multiple options to manipulate the data before going further with the visual data exploration. The data manipulation includes filtering, sorting, aggregation, quick replacement of values with user defined entries, merging and removing columns as can be seen in figure 3.2. Filtering is the only data manipulation which can be reverted without affecting any other manipulation. All the other transformations are permanent for the current session and can only be reverted by reloading the original document. After the data has been transformed, and the user is satisfied with the outcome, two options are available before proceeding to the *Visualization Dashboard*. In order avoid having to do the same manipulations every time on the same dataset before proceeding to the dashboard, it is possible to either download the manipulated CSV file for later use, or alternatively, save the applied data transformations in a configuration link or configuration file. Both the configuration link and the configuration file contain the same information; the only difference is in the usage. The configuration file has to be uploaded in addition to the local file or added with the URL to the remote file. However, the configuration link has to be pasted directly into the address bar of a web browser and is automatically applied to every loaded dataset. In order to ensure that the transformations have the desired outcome, the different datasets used with the same configuration have to have the same structure, otherwise unforeseeable outcomes are possible.



street (string)	city (string)	zip (integer)	state (string)	beds (integer)	baths (integer)	sq_ft (integer)	type (string)	sale_date (date)	price (integer)	latitude (number)	longitude (number)
3526 HIGH ST	SACRAMENTO	95838	CA	2	1	836	Residential	Wed May 21 ...	59222	38.631913	-121.434879
51 OMAHA CT	SACRAMENTO	95823	CA	3	1	1167	Residential	Wed May 21 ...	68212	38.478902	-121.431028
2796 BRANC...	SACRAMENTO	95815	CA	2	1	796	Residential	Wed May 21 ...	68880	38.618305	-121.443839
2805 JANETTE...	SACRAMENTO	95815	CA	2	1	852	Residential	Wed May 21 ...	69307	38.616835	-121.439146
6001 MCMAH...	SACRAMENTO	95824	CA	2	1	797	Residential	Wed May 21 ...	81900	38.51947	-121.435768
5828 PEPPER...	SACRAMENTO	95841	CA	3	1	1122	Condo	Wed May 21 ...	89921	38.662595	-121.327813
6048 OGDEN ...	SACRAMENTO	95842	CA	3	2	1104	Residential	Wed May 21 ...	90895	38.681659	-121.351705
2561 19TH AVE	SACRAMENTO	95820	CA	3	1	1177	Residential	Wed May 21 ...	91002	38.535092	-121.481367
11150 TRINIT...	RANCHO COR...	95670	CA	2	2	941	Condo	Wed May 21 ...	94905	38.621188	-121.270555
7325 10TH ST	RIO LINDA	95673	CA	3	2	1146	Residential	Wed May 21 ...	98937	38.700909	-121.442979
645 MORRISO...	SACRAMENTO	95838	CA	3	2	909	Residential	Wed May 21 ...	100309	38.637663	-121.45152
4085 FAWN CIR	SACRAMENTO	95823	CA	3	2	1289	Residential	Wed May 21 ...	106250	38.470746	-121.458918
2930 LA ROSA...	SACRAMENTO	95815	CA	1	1	871	Residential	Wed May 21 ...	106852	38.618698	-121.435833
2113 KIRK WAY	SACRAMENTO	95822	CA	3	1	1020	Residential	Wed May 21 ...	107502	38.482215	-121.492603
4533 LOCH H...	SACRAMENTO	95842	CA	2	2	1022	Residential	Wed May 21 ...	108750	38.672914	-121.35934
7340 HAMDE...	SACRAMENTO	95842	CA	2	2	1134	Condo	Wed May 21 ...	110700	38.700051	-121.351278
6715 6TH ST	RIO LINDA	95673	CA	2	1	844	Residential	Wed May 21 ...	113263	38.689591	-121.452239
6236 LONGF...	CITRUS HEIG...	95621	CA	2	1	795	Condo	Wed May 21 ...	116250	38.679776	-121.314089
250 PERALTA ...	SACRAMENTO	95833	CA	2	1	588	Residential	Wed May 21 ...	120000	38.612099	-121.469095
113 LEEWILL ...	RIO LINDA	95673	CA	3	2	1356	Residential	Wed May 21 ...	121630	38.689999	-121.46322

Figure 3.2: The *Dataset Table* with loaded data

After the user has completed the data transformations on the loaded dataset, he can proceed to the *Visualization Dashboard*. While the *Dataset Table* is a place to get an overview of the loaded data and to perform alterations on it, the *Visualization Dashboard* is the place, where the user does the actual visual data exploration and data analysis. The dashboard consists of four parts:

1. *Visualization Space:* The created visualizations are placed here upon generation
2. *Field Selection Area:* It contains all the data fields of the *Dataset Table*
3. *VisPicker:* This is tool for the selection of compatible visualizations
4. *Toolbar:* It offers navigation, view, tools, download, upload options and user login

The dashboard and its individual parts can be seen in figure 3.3. The fields from the *Dataset Table*, will be displayed in the Field Selection Area on the left in the separate windows based on their data type. The dashboard is designed in such a way that users with preknowledge in data visualization will be instantly familiar with its design and that users without any knowledge in this area will be supported in generating visualizations. The data visualization process is as follows: the user can select the fields of interest in the left sidebar and in the *VisPicker* compatible visualizations will be displayed. When clicking on a visualization icon in the *VisPicker* the user will first be asked to select how the numerical values should be aggregated and then create a new window in the visualization space with the visualization previously chosen. This window can be freely moved in the visualization space, resized, minimized, maximized or closed. When multiple visualizations are created, they may also overlap each other. Every created visualization supports dedicated filtering and sorting which only affects the contents of the visualization. Furthermore, certain visualizations support channel remapping if more than one field to visual channel mapping exists for this particular visualization.

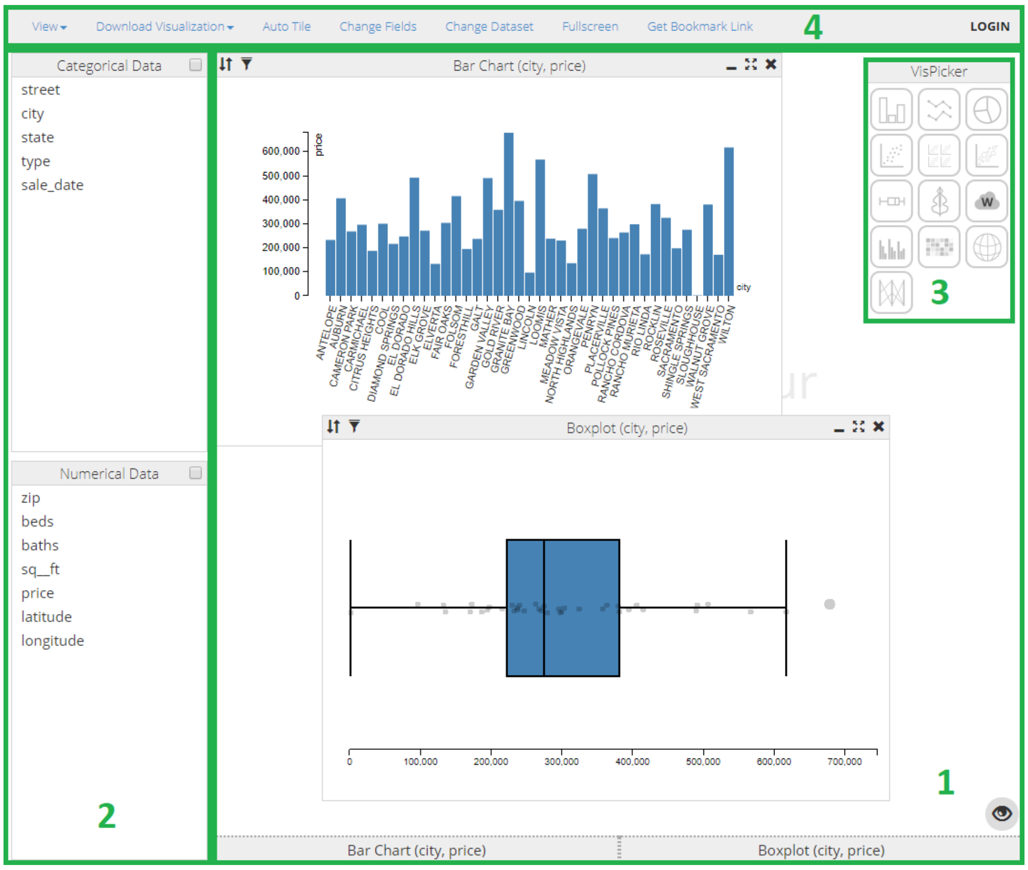


Figure 3.3: The *Visualization Dashboard* and its components: (1) Visualization Space, (2) Field Selection Area, (3) *VisPicker* and (4) Toolbar

All of the created visualizations are linked with each other, thus performing a selection in one visualization will highlight the same data points in another visualization. Besides saving the configuration of the manipulated data in the *Dataset Table*, the dashboard also has its own functionality of saving a session, namely generating a bookmark link. This bookmark link contains the information of the current state of the dashboard with the displayed visualizations, the position and size of each visualization and the performed aggregations. As mentioned, the dashboard is personalized and extensible. Those functionalities are only present if the user is logged in, because the data is saved to the user's profile. Personalization of the dashboard includes the customization of the dashboard, which can be the change of color of almost every aspect of the dashboard and changing the font types and sizes of the different windows. Additionally, a personalized recommender for visualizations is also part of *Visualizer*, which can be accessed only by logged in users. If the user is not satisfied with the visualizations already included in the dashboard, there is a possibility to upload and integrate his own visualizations. In order to create a visualization a very simple interface has to be implemented which is only responsible for drawing the passed data. After the visualization has been implemented it can be uploaded after describing the visualization and the needed data types. In the next sections all of the mentioned functionalities will be examined in much more detail. Firstly, a look will be taken at the structure of the web application, the used tools and libraries, and how everything works together in order to understand the technological background behind it. Thereafter, the *Dataset Table* and the *Visualization Dashboard* will be examined and every functionality explained.

3.2 Architecture and Used Technologies

This subsection will first outline the architecture of the application and all the decisions which had to be made concerning this aspect. It will also serve as background information for the detailed explanation of the smaller components of the *Dataset Table* and *Visualization Dashboard*. While covering each component of the architecture, the used tools and libraries will also be explained. Figure 3.4 shows the web architecture of *Visualizer*, it is divided into a front end, a back end and external services, which is very common for web architecture.

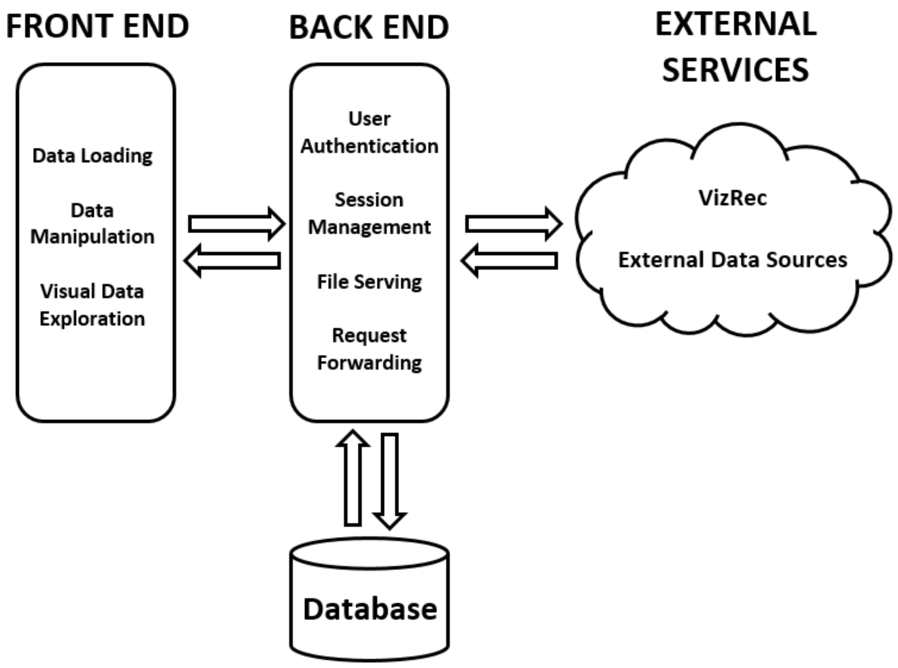


Figure 3.4: Web Architecture of *Visualizer*: Split into front end, back end and external resources

3.2.1 Back End

The back end in *Visualizer*s case is very thin. It is not used for any complex computations and is only for tasks which cannot be performed on the front end. The back end consists of an Apache[2] web server, which executes the desired Common Gateway Interface scripts, also called CGI[3] scripts, written in Python 3[4]. This combination was chosen because the programmer is familiar with it, the setup was simple and all the needed functionality is included out of the box without the need for additional libraries or tools. As a database SQLite 3[5] was chosen, again because of the familiarity and the simplicity of the use case. The database is only needed for user creation, authentication and session management and therefore consists of only one table with the schema, which can be seen in table 3.1
The back end has the following functionalities:
User creation - When a user fills in the registration form on the client and sends it to the server, a new user has to be created in the database and a user directory has to be generated. For this purpose, first a check is performed

[2] https://httpd.apache.org
[3] https://tools.ietf.org/html/rfc3875
[4] https://www.python.org/download/releases/3.0/
[5] https://www.sqlite.org

users		
Fieldname	*Type*	*Description*
uid	integer autoincrement primary key	A unique identification number which serves as the primary key for the table
username	text not null	A unique username. In case a user tries to take an already reserved username, a message will be displayed to choose another
passhash	text not null	The hash value of the user password. It is used for authentication
userdirectory	text not null	The path to the directory where the user style sheets and upload visualizations are
email	text not null	The email address of the user
lastrequest	integer	The timestamp of the last request. It is used for session timeout
activesessionid	text	The id of the active session

Table 3.1: The schema of the `users` table and the description of the fields

if the username has already been taken. The username has to be unique, because alongside the password it serves for the user's authentication. However, the password is not stored as plain text, but as a hash of the password entered by the user. This is done for security reasons. In case of a breach, no user password is revealed. If the username is valid, a new entry will be created in the database for this user and a directory in the filesystem will be created and initialized with the default structure. The newly created user folder contains only two folders upon creation: (i) style and (ii) charts. The style folder contains the initial files for the customized stylesheet. The charts folder on the other hand contains two user mapping files which describe the uploaded visualizations. One file is a JavaScript file and the other a JSON file. The JavaScript file is served to the user and the JSON file is used for easier parsing of the values on the server. Upon creation they only contain empty mapping objects, because upon creation no visualizations uploaded by the user are present. The mappings.js file, utilized by every user, and containing all the descriptions for the default visualizations can additionally be inspected in Appendix B. After the successful entry into the database and the creation of the user folder in the filesystem, the newly registered user can log in into his account.
User authentication and presentation of the correct start page - *Visualizer* can be used without the need to create an account, but without an

account, certain features will be missing. Those features include customized stylesheets, visualizations created by the user and personalized recommendation of visualizations. Therefore, one of the responsibilities of the back end is also the authentication of the user and providing of the correct starting page and files.
After the user has created an account, the back end is responsible to check whether the login was successful or not, whenever a user tries to log in. When doing this he has to enter his username and password. First the username is checked if it exists in the database and the appropriate user row in the table is selected. Then a hash is generated from the password which was entered by the user and compared to the saved hash. If it matches, a new session id and a timestamp are generated of the login time and stored in the database. Furthermore, the web page gets reloaded and the user is presented with his personal customized dashboard.
Session management - In the user authentication description it was mentioned that the back end stores a session id and a timestamp of the last request. This is done to guarantee that only the real user has access to his personal files (stylesheets and visualizations). When the user logs in, a cookie is set with the session id of the user. This cookie is valid for 12 hours, i.e. the session will expire after that and a new login will be required. The cookie is sent to the server on every user request, which could be reloading the page, accessing visualizations or loading new data. The timestamp is saved on the server to ensure that the cookie is not manipulated on the client's side so that the session actually expires if no request has been sent for 12 hours. On every request, this timestamp is checked and if the session is still valid, the timestamp is updated.
Managing user style sheets - When a user saves the modifications to the visual style of the dashboard, a script on the server generates a new stylesheet with the customized values, which is saved in the style directory of the user's directory. When the user logs out and logs in again, this stylesheet will be served to him.
Managing user uploaded visualizations - The charts directory in the user directory contains all of the needed files to integrate and display the files uploaded by the user. As mentioned in the user creation subsection, at first this folder only contains a file called usermapping.js and usermapping.json, which contain only empty visualization mapping objects. Every time a new visualization is uploaded, a new entry is created in those files which describes the properties of the new visualization. In addition to those two files, which describe the visualizations, each visualization is put into its own respective directory. The visualization directories always have the same basic structure, as it can be seen for the directory exampleChart in the userfolder:

```
userfolder/
+-- style/
+-- charts/
+-- +-- usermapping.js
+-- +-- usermapping.json
+-- +-- exampleChart/
+-- +-- +-- css/
+-- +-- +-- fonts/
+-- +-- +-- libs/
+-- +-- +-- mainfile.js
+-- +-- +-- icon.png
```

It is expected from the user, when uploading new visualizations, to follow this directory scheme for new visualizations, otherwise the visualizations will not work properly in *Visualizer*. After the visualization has been uploaded and successfully added to the user's directory it can be used in *Visualizer*. Like the user stylesheets, those visualizations are only served to the logged in user who uploaded them.

Fetching datasets from external data sources - Trying to fetch datasets from other servers from the client side is not possible, because in every web browser Cross-Origin Resource Sharing [6] is not allowed per default due to security reasons. Consequently, the back end becomes responsible to fetch data from external resources. When a URL to a dataset is entered on the client's side, this URL gets passed to the back end. If it is a dataset located on *Visualizer's* server (demo data), it gets served immediately. In the future this mechanism could also allow users to store their own datasets in their respective user directories. In case the dataset actually comes from a different server, the dataset is fetched and temporarily saved in a directory. The content of this dataset is then passed to the user for inspection.

Communicating to the external recommender - Since the visualization recommender is an external service and not part of *Visualizer's* back end, the same problem as with the external datasets occurs. Hence, the back end has to pass the data needed for the recommender to the recommender and then return it to the user. The visualization mappings, tags and ratings for the recommender are completely prepared for the recommender in the format with which the recommender can work on the client. The only information which the back end adds to this data before passing it to the recommender is the user id. This is needed so that the recommender can learn more about the specific user and select the personalized recommendations for this user. When a response comes back from the recommender, the server sends this

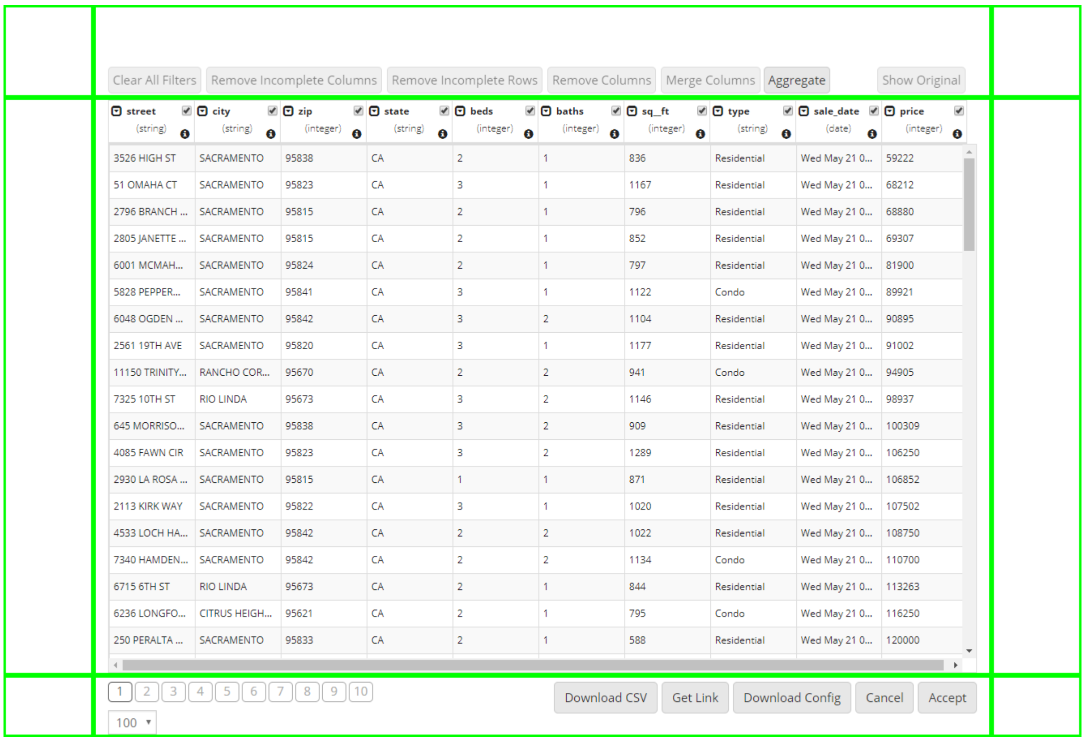
[6] `https://tools.ietf.org/html/rfc6454`

response to the user. The requests from the client to the server and then to the recommender are all non-blocking, i.e. when a user performs an action related to the recommender, he can continue to work without having to wait for a response.

3.2.2 Front End

Considering that in the overview subsection the basic functionality of the front end was already explained, in this subsection the focus will be brought on the technological aspects and used libraries for the creation and functionalities of the front end. As the fundament for the web application, the newest versions of HTML and CSS were used, respectively HTML5 and CSS3. Additionally JavaScript was used with the ECMAScript 6[7] functionalities. For extended programming functionalities the well known JavaScript library jQuery[8] was used. This library made the selection and handling of DOM elements much simpler, the source code cleaner and ensured faster programming. In addition to jQuery, the jQuery UI[9] library was used to add easy mechanisms for resizing and dragging elements. This library is mostly used for the visualization windows and the *VisPicker*, allowing them to be moved freely in the visualization space and additionally the visualization windows can be resized. Instead of having to implement all of the click, drag and drop handlers individually, a single call to the library is needed in order to enable an element with this functionality. For the responsive structure of the layout, the consistent style of all elements and all the icons except the visualization icons, the bootstrap[10] library was used. This library offers a way to structure a web page in a grid system which is divided into 12 columns, in which an arbitrary number of rows can be added. It is the developer's decision to define in which way the columns should be ordered. Figure 3.5 shows how the *Dataset Table* is structured with the help of Bootstraps grid system.
Furthermore, Bootstrap has a class system for various elements to ensure a consistent layout. Thus, adding the class "btn" to an element will apply the appropriate style so that all buttons look and behave the same. This includes highlighting the buttons and changing the cursor to a pointer when hovering over a clickable button. An additional feature of bootstrap which was widely used in *Visualizer* were the included icons. Bootstrap includes a variety of default icons called glyphicons, covering most of the icon needs for a web page. Those icons are integrated in a font and since they are vector graphics they

[7]`http://es6-features.org`
[8]`https://jquery.com`
[9]`https://jqueryui.com`
[10]`http://getbootstrap.com`



street (string)	city (string)	zip (integer)	state (string)	beds (integer)	baths (integer)	sq_ft (integer)	type (string)	sale_date (date)	price (integer)
3526 HIGH ST	SACRAMENTO	95838	CA	2	1	836	Residential	Wed May 21 0...	59222
51 OMAHA CT	SACRAMENTO	95823	CA	3	1	1167	Residential	Wed May 21 0...	68212
2796 BRANCH ...	SACRAMENTO	95815	CA	2	1	796	Residential	Wed May 21 0...	68880
2805 JANETTE ...	SACRAMENTO	95815	CA	2	1	852	Residential	Wed May 21 0...	69307
6001 MCMAH...	SACRAMENTO	95824	CA	2	1	797	Residential	Wed May 21 0...	81900
5828 PEPPER...	SACRAMENTO	95841	CA	3	1	1122	Condo	Wed May 21 0...	89921
6048 OGDEN ...	SACRAMENTO	95842	CA	3	2	1104	Residential	Wed May 21 0...	90895
2561 19TH AVE	SACRAMENTO	95820	CA	3	1	1177	Residential	Wed May 21 0...	91002
11150 TRINITY...	RANCHO COR...	95670	CA	2	2	941	Condo	Wed May 21 0...	94905
7325 10TH ST	RIO LINDA	95673	CA	3	2	1146	Residential	Wed May 21 0...	98937
645 MORRISO...	SACRAMENTO	95838	CA	3	2	909	Residential	Wed May 21 0...	100309
4085 FAWN CIR	SACRAMENTO	95823	CA	3	2	1289	Residential	Wed May 21 0...	106250
2930 LA ROSA ...	SACRAMENTO	95815	CA	1	1	871	Residential	Wed May 21 0...	106852
2113 KIRK WAY	SACRAMENTO	95822	CA	3	1	1020	Residential	Wed May 21 0...	107502
4533 LOCH HA...	SACRAMENTO	95842	CA	2	2	1022	Residential	Wed May 21 0...	108750
7340 HAMDEN...	SACRAMENTO	95842	CA	2	2	1134	Condo	Wed May 21 0...	110700
6715 6TH ST	RIO LINDA	95673	CA	2	1	844	Residential	Wed May 21 0...	113263
6236 LONGFO...	CITRUS HEIGH...	95621	CA	2	1	795	Condo	Wed May 21 0...	116250
250 PERALTA ...	SACRAMENTO	95833	CA	2	1	588	Residential	Wed May 21 0...	120000

Figure 3.5: Layout structure of *Dataset Table* with Bootstrap

always look sharp no matter what size the icon is.
For data extraction and conversion the library PapaParse[11] was used. This library is used for two very important functionalities. The first is for reading CSV files and converting them to JSON, which can then be parsed with ease in the browser. The second important functionality is creating a CSV file from the manipulated table, which then can be downloaded. PapaParse reads either a local file, or a file from a URL. In case the URL is not from the same domain, the URL is passed to the server, which then returns a URL with the temporary file on the server. This URL can be then used by PapaParse.
Another very useful library lodash[12] was also included in *Visualizer*. This library is used for array manipulation and data aggregation. It has built in functionalities for operations on sets, like intersections and unions, and also the most used aggregation operations on arrays with categorical or numerical values. Moreover, it makes it unnecessary to reimplement those extensively used utility functionalities.
In addition to those extensively used libraries named before, there are multiple libraries worth mentioning for smaller functionalities which were included

[11]`http://papaparse.com`
[12]`https://lodash.com`

in *Visualizer*.

- *Moment.js*[13] was used for easier date manipulation
- *Masonry.js*[14] for the auto tile feature, which organizes the visualization windows in a manner that they do not overlap
- *Spin.js*[15] was used for the dynamic creation of a loading/computing spinner
- *Canvg*[16] was used for rendering SVG elements into a canvas and enabling a direct download of visualization images from the client

3.2.3 External Services

At the time of writing, the only external service used in *Visualizer* is the visualization recommender which can only be used when a user is logged in. The visualization recommender expects two things: (i) a list of tags, describing the interests of the user (ii) a list of possible configurations of mappings from selected fields unto visual channels of a visualization. When the list gets generated it is passed to the back end with the added tags, which then calls the recommender with the user id. The recommender sends as a response a ranked list of recommended visualizations with set channel mappings. The back end redirects the recommender response to the front end, which then displays the ranked list of visualizations. Furthermore, the user has the option to rate and tag the visualizations suggested by the recommender as feedback. Those tags and ratings are then used by the recommender to improve and suggest better personalized recommendations of visualizations for the user giving the feedback and for other users with similar needs.

3.3 Dataset Table

This subsection will cover every functionality of the *Dataset Table* in detail. At first, the processes will be explained that happen immediately after a dataset has been loaded, the needed computations and preparations for the data to be presented in the table. Afterwards, it will be explained how the preprocessed data is used to draw the table and what the links between the dataset object and the table are. Furthermore, the table will be split into multiple smaller components which will then be analyzed.

[13]`https://momentjs.com`
[14]`https://masonry.desandro.com`
[15]`http://spin.js.org`
[16]`https://github.com/canvg/canvg`

3.3.1 Data Preprocessing

After the selection of a local or remote dataset, the data has to be preprocessed before it is displayed in the *Dataset Table*. With PapaParse it is possible to extract the raw data from the CSV file and convert it to JSON, but for the purposes of *Visualizer* the data has to be further enriched. For this enrichment a dataset object gets created. This dataset object contains a fields array. Every entry in this array comprises the raw data from a column of the dataset and additional computed information about the field. In table 3.2 the structure of an object in the fields array is displayed.

Fields object		
Type	*Attribute name*	*Description*
array	data	Each entry in this array is one cell from the column in the dataset
integer	datapoints	Total number of different values in this column
object	filterList	Contains the filtering constraints for this column. Detailed description in table 3.3
boolean	isActive	Should the field be displayed in the dashboard or not
boolean	isIncomplete	Does this column have empty cells
string	name	The name of the column
string	type	The detected data type of the values in the column
array	uniqueData	The unique values of the data array

Table 3.2: Structure of an object in the `fields` array

When creating the `fields` objects, the `data` attribute of a `fields` object is assigned the raw data of one column of the dataset. After the data has been assigned, the real dataset has been loaded into the `dataset` object. The `name` attribute of the `fields` object is set to the name of the column in the original dataset, and the `isActive` attribute is per default set to `true`. Subsequently, the data type of each object has to be detected. For this, a maximum of 100 first entries are checked for their data type in each column to determine the data type of the whole column. *Visualizer* differentiates between five data types. (i) integer, (ii) number, (iii) date, (iv) location and (v) string. The checks are performed in the following order: If two entries in a column have different data types, the detected column type will be set to string. The exception is if an integer and number are detected as a data type for a column. In this case, number is set as the detected data type for the column, and the check

1. integer: Check if the entry contains only the values between 0 and 9
2. number: In addition to the integer check, check if ',' or '.' are present
3. date: Parse entry as a JavaScript date object and check if return value is a number
4. location: Compare entries to an array of supported locations
5. string: If all tests fail, the entry is detected as a string

continues. The data type check finishes, after a maximum of 100 entries in a column have been checked, or as soon as the detected data type is determined as string.
In the next step, the unique data entries have to be detected. This task becomes trivial, thanks to lodash's uniq function. This information is needed to detect if in a column certain values repeat themselves very often so that they can be handled as categories. An example would be a column with the days of a week. Even though there could be a thousand entries in a column, only seven will repeat. After assigning an array of unique elements from the data, the number of different entries in the column are set to the `datapoints` attribute of the `fields` object. This is done by taking the length of the `uniqueData` array. Additionally, after having the `uniqueData` array, it can be determined if the column has missing entries. This is done by checking the `uniqueData` array for an empty string. If an empty string can be found, the `isIncomplete` attribute of the `fields` object is set to be true.
The `filterList` attribute is not actually a list, but an object containing constraints. The name stayed even though an adoption to the data structure was performed. This object will later contain all the constraints for its column according to which the dataset should be filtered. In table 3.3 the structure of the `filterList` object can be seen.

FilterList object		
Type	*Attribute name*	*Description*
integer	from	lowest value that can be used
array	keywords	entries that can be displayed
integer	to	highest value that can be used

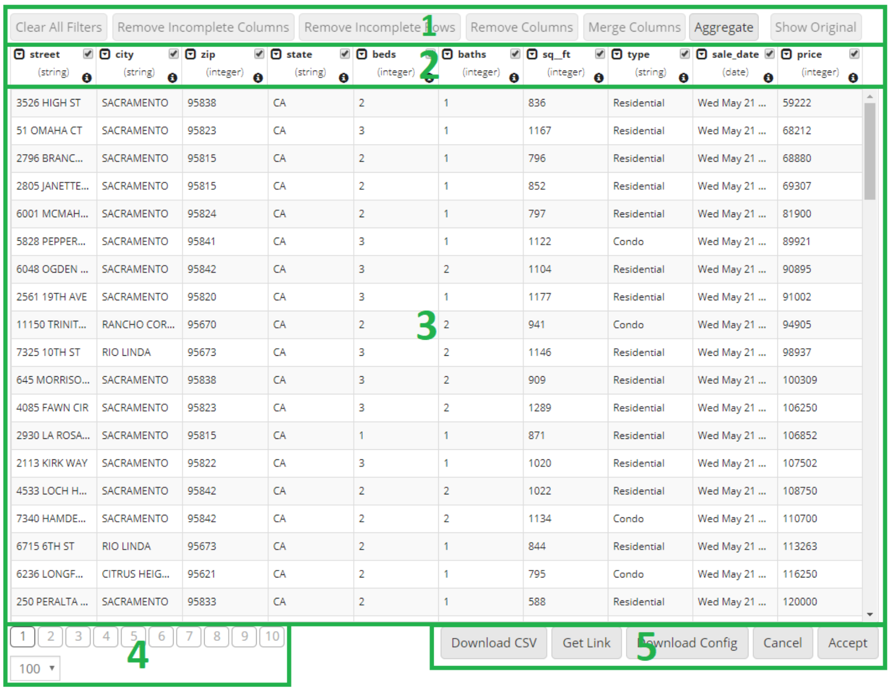
Table 3.3: Structure of a `FilterList` object

3.3.2 Displaying the Table

The `dataset` object is the actual loaded dataset. The table and the header only display the current contents of this object. When creating the table of contents, the data arrays in the `fields` objects of the respective column are walked through and the contents displayed. The header, like the contents of the table, is also a visual representation of certain fields in the `fields` object. The displayed name is saved in the `name` attribute of the `fields` object and the data type in the `datatype` attribute. If a functionality is present in the table to alter certain values, they will be changed in the `dataset` object and the table refreshed. It is also notable that not the whole dataset is displayed in the table of contents at the start, but only the first 100 entries.

3.3.3 Components

As can be seen in figure 3.6, the *Dataset Table* can be split into five regions categorized by functionality. Region 1 is for manipulation of the whole table. Region 2 is the table header, with the information about each column. Region 3 is the table with the contents of the dataset. In Region 4 are the navigation elements. Region 5 is for accepting and/or saving the manipulated table.



street (string)	city (string)	zip (integer)	state (string)	beds (integer)	baths (integer)	sq_ft (integer)	type (string)	sale_date (date)	price (integer)
3526 HIGH ST	SACRAMENTO	95838	CA	2	1	836	Residential	Wed May 21 ...	59222
51 OMAHA CT	SACRAMENTO	95823	CA	3	1	1167	Residential	Wed May 21 ...	68212
2796 BRANC...	SACRAMENTO	95815	CA	2	1	796	Residential	Wed May 21 ...	68880
2805 JANETTE...	SACRAMENTO	95815	CA	2	1	852	Residential	Wed May 21 ...	69307
6001 MCMAH...	SACRAMENTO	95824	CA	2	1	797	Residential	Wed May 21 ...	81900
5828 PEPPER...	SACRAMENTO	95841	CA	3	1	1122	Condo	Wed May 21 ...	89921
6048 OGDEN ...	SACRAMENTO	95842	CA	3	2	1104	Residential	Wed May 21 ...	90895
2561 19TH AVE	SACRAMENTO	95820	CA	3	1	1177	Residential	Wed May 21 ...	91002
11150 TRINIT...	RANCHO COR...	95670	CA	2	2	941	Condo	Wed May 21 ...	94905
7325 10TH ST	RIO LINDA	95673	CA	3	2	1146	Residential	Wed May 21 ...	98937
645 MORRISO...	SACRAMENTO	95838	CA	3	2	909	Residential	Wed May 21 ...	100309
4085 FAWN CIR	SACRAMENTO	95823	CA	3	2	1289	Residential	Wed May 21 ...	106250
2930 LA ROSA...	SACRAMENTO	95815	CA	1	1	871	Residential	Wed May 21 ...	106852
2113 KIRK WAY	SACRAMENTO	95822	CA	3	1	1020	Residential	Wed May 21 ...	107502
4533 LOCH H...	SACRAMENTO	95842	CA	2	2	1022	Residential	Wed May 21 ...	108750
7340 HAMDE...	SACRAMENTO	95842	CA	2	2	1134	Condo	Wed May 21 ...	110700
6715 6TH ST	RIO LINDA	95673	CA	2	1	844	Residential	Wed May 21 ...	113263
6236 LONGF...	CITRUS HEIG...	95621	CA	2	1	795	Condo	Wed May 21 ...	116250
250 PERALTA ...	SACRAMENTO	95833	CA	2	1	588	Residential	Wed May 21 ...	120000

Figure 3.6: Regions of the *Dataset Table*: (1) Table manipulation (2) Table information header (3) Dataset contents table (4) Table navigation elements (5) Save and accept

Table Manipulation

The table manipulation buttons can be used to manipulate the table as a whole as can be seen in figure 3.7.

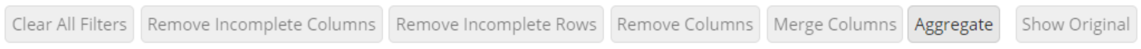


Figure 3.7: Table manipulation buttons

Almost all of the functionalities in this part of the *Dataset Table* are self-explanatory or very simple, except the “Aggregate” and the “Merge Columns” buttons, which will be explained in more detail. The buttons have the following functionalities:

- *Clear All Filters*: Reverts the filterList object to its original state for every field object
- *Remove Incomplete Columns*: Removes all fields objects with isIncomplete set to true, then refreshes the table
- *Remove Incomplete Rows*: For every empty entry in a data array of a field object, removes the entry at the same index in the data arrays of other field objects. Afterwards the table is refreshed.
- *Remove Columns Removes*: the field objects from the dataset object which were selected in the table
- *Merge Columns*: Merges the field objects of the dataset object which were selected in the table
- *Aggregate*: Aggregates the values in the dataset object
- *Show Original*: Drops all changes and displays the original dataset

Merge Columns - Columns can be selected by clicking on the header while holding the CTRL key pressed. If two or more columns are selected, the merge columns button will become active. A merge of columns means for the `dataset` object that a new `fields` object is generated with the merged values, and the selected `fields` objects are removed. This is done by first generating the new data array for the `fields` object. All entries in the data array of a selected `fields` object are iterated and merged with the entries at the same index of the other data arrays of the selected fields. Merging two or more entries is different, depending on the data type of the selected fields. As long as not all selected fields are numerical, the entries will be simply concatenated with an ", " in between. However, if all of the selected fields are of a numerical type, a dialog will be presented to ask for an aggregation method for those entries.

Selecting for example “Sum” in the dialog, will add all entries under an index. The aggregation dialog can be seen in figure 3.8.

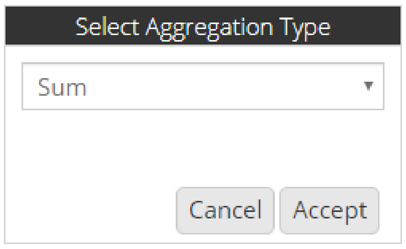


Figure 3.8: Aggregation dialog for merging numerical columns

After the new merged data array has been created, it can be assigned to the data array of the new `fields` object. The `name` attribute is set to a concatenation of all field names. The rest of the attributes in the `fields` object are then computed the same way, when the data gets loaded first. Following the creation of the new `fields` object and the removal of the selected ones, the dataset contents table and the header are refreshed.
Aggregate - Clicking on the “Aggregate” button will open the aggregation dialog as seen in figure 3.9

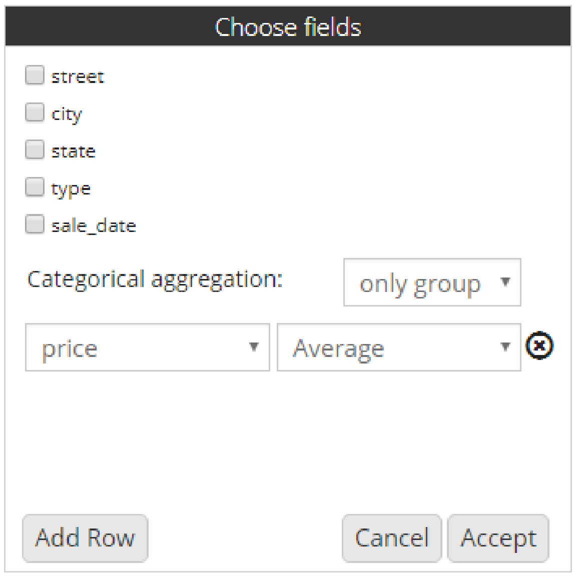


Figure 3.9: Aggregation dialog for aggregating whole dataset

The aggregation dialog is split into actions for categorical data and actions for numerical data. The checkboxes beside the categorical fields indicate what fields should be taken when performing the aggregation. Below the checkboxes with the field names is a select box, where it is possible to select if an additional aggregation should be performed on the categorical fields. Those optional aggregations are *count* and *ratio*. *Count* counts the occurrences of a unique tuple of fields with the same index in the data array of the field objects, or simply put, the unique rows in the dataset. *Ratio* displays with a percentage how often a unique row occurs compared to the total number of occurrences. The numerical values are in the select boxes below that. In each row, a numerical field can be selected and the type of aggregation to be performed on it. The possible aggregations are: minimum, maximum, sum and average. After clicking on "Accept", the original `dataset` object will be swapped with the aggregated one and the table refreshed.

Information Header

The information header serves the purpose of displaying compact information about each column. In figure 3.10 an isolated cell of the information header is displayed.

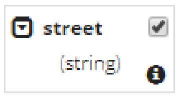


Figure 3.10: Isolated information header cell

The information header cell consists of five elements. From top-left to bottom-right: (i) context-menu button, (ii) field name, (iii) activity toggle, (iv) data type, (v) information element. The context-menu button opens a context menu in which the entries of the column can be filtered, sorted, or a default value added if the column misses it. Depending on the data type of the field, the context menu can look differently as can be seen in figure 3.11. Additionally, if the field can be categorized, the categories can be directly selected.

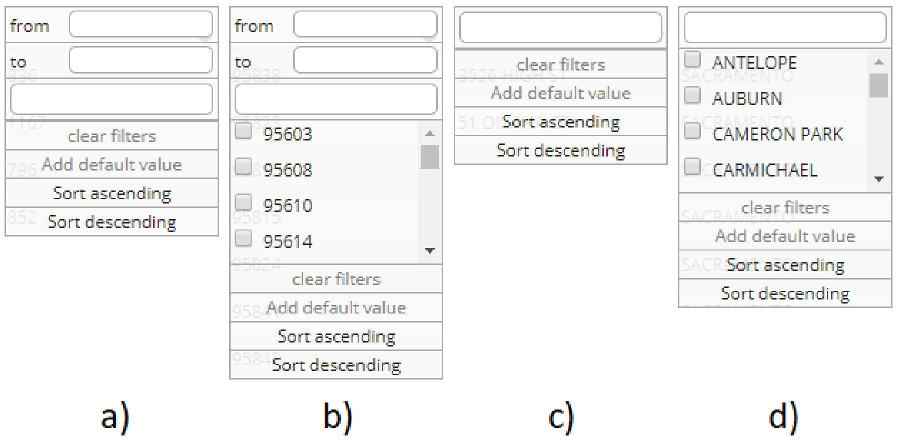
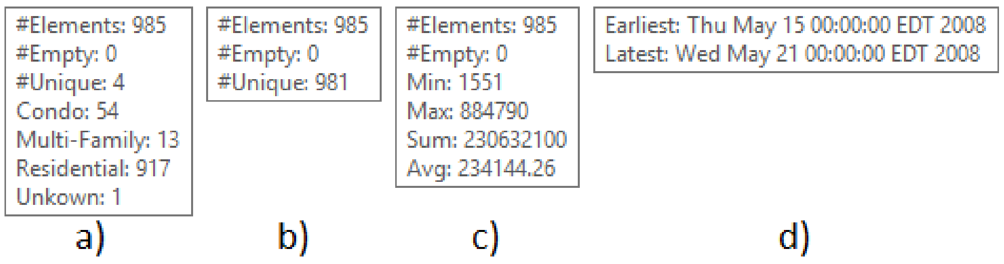


Figure 3.11: Information header context-menus: a) numerical without categorization b) numerical with categorization c) categorical without categorization d) categorical with categorization

Numerical data types are integer and number, and categorical types are date, location and string. Numerical values have the additional “from” and “to” fields. When entering values in the filter fields, the `filterList` object in the respective `fields` object is updated. The structure of the `filterList` object can be seen in table 3.3 in the data preprocessing subsection. Thus, entering values in the “from” and “to” field for numerical values adds them to the same called fields in the `filterList` object. The third field adds values directly to the keywords array of the `filterList` object. Depending, if the data can be categorized or not, this field behaves slightly differently. For data which can be categorized, entering text in this field will filter the categories below it, which can then be toggled. If the field cannot be categorized, hitting enter after the text has been entered will add the value directly to the keywords array. The filtering process of the whole table is applied every time a value has been added to a `filterList` object. The filtering mechanisms work as follows: For every `filterList` object which is not empty, the entries in the data array of this field are checked if they satisfy the constraints. If it is true, the indexes of those entries are saved. For the next field which does not have an empty `filterList` object, only the entries with the already limited indexes from the previous filtering are checked, thus limiting the indexes further. After all filters have been checked, a list of indexes is created which contains only the data that satisfies all of the constraints. Those entries are then displayed in the table.
The field name, as it suggests, is just the name of this field. The activity toggle (per default on) can disable the field, i.e. it will not be displayed in the dashboard.

The displayed data type is the same data type as in the field object representing this column in the dataset. It can be clicked and the data type changed. The data types can be only changed to a superset of the detected data type, otherwise the option will not be available. An integer can be changed to a number or a string, but not the other way around. The same goes for date to string and location to string. An integer can also be changed to a date, but this is the risk of the user. The reason is that a number can be also interpreted as a year.
The information element displays calculated information about the data in this column when hovering over it. Depending on the data type of the field, and again, depending on whether it can be categorized or not, different entries are displayed. Figure 3.12 displays the different information tooltips for fields of different data types.



```
#Elements: 985
#Empty: 0
#Unique: 4
Condo: 54
Multi-Family: 13
Residential: 917
Unkown: 1
```

```
#Elements: 985
#Empty: 0
#Unique: 981
```

```
#Elements: 985
#Empty: 0
Min: 1551
Max: 884790
Sum: 230632100
Avg: 234144.26
```

```
Earliest: Thu May 15 00:00:00 EDT 2008
Latest: Wed May 21 00:00:00 EDT 2008
```

Figure 3.12: Information tooltips: a) categorical with categorization b) categorical without categorization c) numerical d) date

Strings and location always display the number of entries, how many are empty and how many are unique. If the data can be categorized, the categories get displayed with their number of occurrences in this column. Numerical fields display also the number of entries and how many are empty, but in addition they also show additional information about the values. Date fields show the range from the starting to the ending date.

Dataset Contents Table

The dataset contents table is the third region of figure 3.6 It is also the region which takes up the most space. The contents of the table are the contents of the data array of each field object, with the data array representing one column in the table. For performance reasons only the first 100 entries in the data array are written. This is due to the fact that the dataset contents table is meant as a tool to preview the data, to see what is in the dataset. The other pages of the data can be loaded on demand by using the table navigation elements which will be covered in the next subsection. The functionalities in the dataset contents table are limited, but not without importance. The first and lesser important functionality is the column resizing. The columns in the table can be freely resized to better analyze the contents. The second, very important feature of the dataset contents table is the replace functionality. It is possible to change the content of every cell. There are two options for replacement: (i) replacing the value of a single cell and (ii) replacing all selected values in the column. Replacing a single value in a cell is very simple. Upon double clicking a cell it will convert to a text field which can be edited. If the user is satisfied with the contents, he can just press enter and the new value will be accepted. This value will automatically be present in the field statistics and categories. The other option is to replace all occurrences of the contents of a cell in a column. This is done by right-clicking a cell and clicking on 're-place all' in the context menu. A dialog will open, as can be seen in figure 3.13, which only allows entries that are of the same type as the other entries in the column.

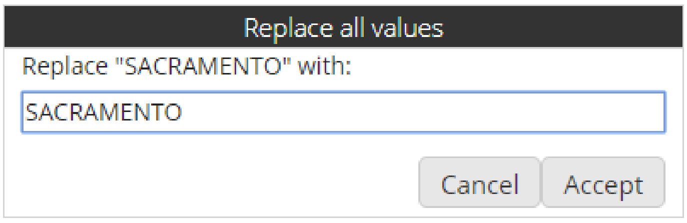


Figure 3.13: Replace all values dialog prefilled with the selected entry

Table Navigation Elements

At the bottom, on the left, underneath the dataset contents table, the navigational elements reside. Figure 3.14 shows the navigational elements up close.

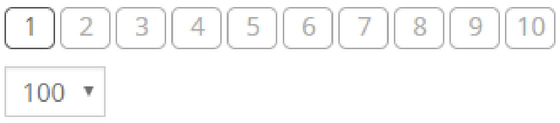


Figure 3.14: Navigational elements

The page numbers allow a quick navigation between the multiple pages of the dataset. In the selectbox below the page numbers it is possible to select the number of rows displayed per page. The default is set to 100 for performance reasons, since more entries can slow down the interaction with the page. Only the first 100 entries of the dataset are displayed when the dataset is loaded. The other pages are only written to the DOM when they are requested by clicking on a page number.

Finalize and Save

The last part of the *Dataset Table* is used either to discard all changes on the dataset or to save them in one way or the other. Figure 3.15 shows the buttons offering different functionalities for saving the data.

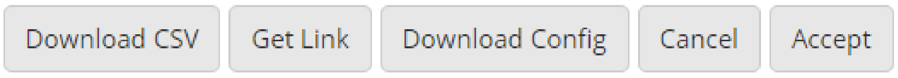


Figure 3.15: Finalize and save commands

Clicking the “Accept” button only transfers the displayed content from the *Dataset Table* to the *Visualization Dashboard*. All the transformations are already performed on the `dataset` object and clicking the “Accept” button leaves the `dataset` object untouched. The “Cancel” button, on the other hand, discards all the changes performed on the dataset and brings the user back to the page where he can upload a new file. “Download CSV”, “Get Link” and “Download Config” allow the user to save the changes performed on the dataset. “Download CSV” will start a download of a CSV file, which contains the manipulated *Dataset Table*. In this way it is possible to save or share an altered version of a dataset. The generation of the CSV file is done by going through the `dataset` object and generating a JSON object with the contents of valid entries. Valid entries in this case means entries of active fields, which also satisfy the constraints given by the filters. The unparse method of PapaParse is then called with this JSON which returns a well formatted CSV file that can be downloaded. “Get Link” and “Download Config” both save the current configuration of the dataset. This means that only the steps to get the dataset in the currently presented form are saved and the dataset itself stays untouched. This is very useful to apply the same transformations on different datasets with the same structure with different content. It can also be used to share a certain view of the dataset with someone, without altering the actual file. The difference between the “Get Link” and “Download Config” buttons lies in the returned output. “Get Link” will create a configuration URL and add it to the clipboard that can be pasted into the address bar of a web browser. When loading a dataset under this URL, the saved transformations will be automatically applied to a dataset. “Download Config” will download a configuration file which has to be added to *Visualizer* when uploading a local dataset.
In order to explain the generation of the configuration file another very important data structure which is frequently used in *Visualizer* needs to be mentioned. This data structure is the `actionStack`. The `actionStack` is an array, which contains all actions which modify the dataset. Every time the dataset is

altered, such as by sorting columns or replacing values, an action describing the performed interaction is pushed on the `actionStack`. This allows *Visualizer* to recreate the current state, starting from the freshly loaded dataset. The following actions are pushed onto the `actionStack` in the *Dataset Table*: (i) remove columns, (ii) merge columns, (iii) rename columns, (iv) aggregate dataset (v) filter (vi) add default value (vii) disabled columns (viii) change type (ix) sort (x) replace (xi) replace all. All of the mentioned actions, except filter and disabled columns are pushed onto the action stack as soon as they have been completed on the *Dataset Table*. The filters and the disabled columns are only pushed onto the `actionStack` when a configuration should be created. This is done to not needlessly fill the `actionStack` with actions which are not required to be in order. The listed actions are only the actions saved for the *Dataset Table*. There are also additional actions for the *Visualization Dashboard* which will be covered in their appropriate subsections later on. For a complete explanation on how every action is structured see Appendix A. When clicking "Get Link" or "Download Config", the `actionStack` is first finalized, i.e. the filters and disabled fields are pushed to the action stack. Then, in the case of "Download Config" the `actionStack` gets stringified before being downloaded, and in the instance of "Get Link" the stringified `actionStack` gets additionally encoded, so that it can be used in a URL. Thus, the contents of the `actionStack` will be in the config parameter of the newly created link. The URL generated by clicking on the "getLink" button consists of three parts:

- `host`: the URL where *Visualizer* is hosted
- `dataurl` (optional): the URL which points to a .csv file on a server (in case the .csv file is located remotely)
- `config`: a stringified JSON object which contains all the information about the performed transformations on the dataset. This is a stringified JSON array of the actionStack

Thus, a URL with all parameters set would have the following structure:

```
host + "?dataurl=" + urltocsv + "&config=" + conf
```

Loading a file into *Visualizer* with a selected configuration file or under a configuration link will automatically start the execution of actions in the configuration, after the file has been loaded. For that, the configuration file or parameter in the URL will be converted to a JSON object. After that the old `actionStack` will be extracted, and all of the saved actions on the action stack will be applied to the freshly loaded dataset in the same order, as they have been previously.

3.4 Visualization Dashboard

This subsection will cover the *Visualization Dashboard* with its two different views: (i) the *Guest Dashboard* and (ii) the *Personalized Dashboard*. The *Personalized Dashboard* has, in addition to the features of the *Guest Dashboard*, personalized recommendations for visualizations and options to customize the dashboard with configurable styles and visualizations, written by the user. The *Visualization Dashboard* with its highlighted areas can be seen in figure 3.16. Each of the highlighted areas will be covered separately in their own subsection. First the visualization space as a central part of the dashboard will be explained. Thereafter, the areas will be covered in the order in which they are usually used. First the field selection area will be explained, because the process of visual exploration starts by selecting fields of interest which should be visualized. After that the *VisPicker* and the process of recommending compatible visualizations will be described. Following the *VisPicker*, the created visualization windows will be examined, and at last the toolbar. After all of the features of the *Guest Dashboard* have been explained, the additional features of the *Personalized Dashboard* will be examined. Those include the user defined customization of the dashboard, the user uploaded visualizations and the personalized visualization recommender.

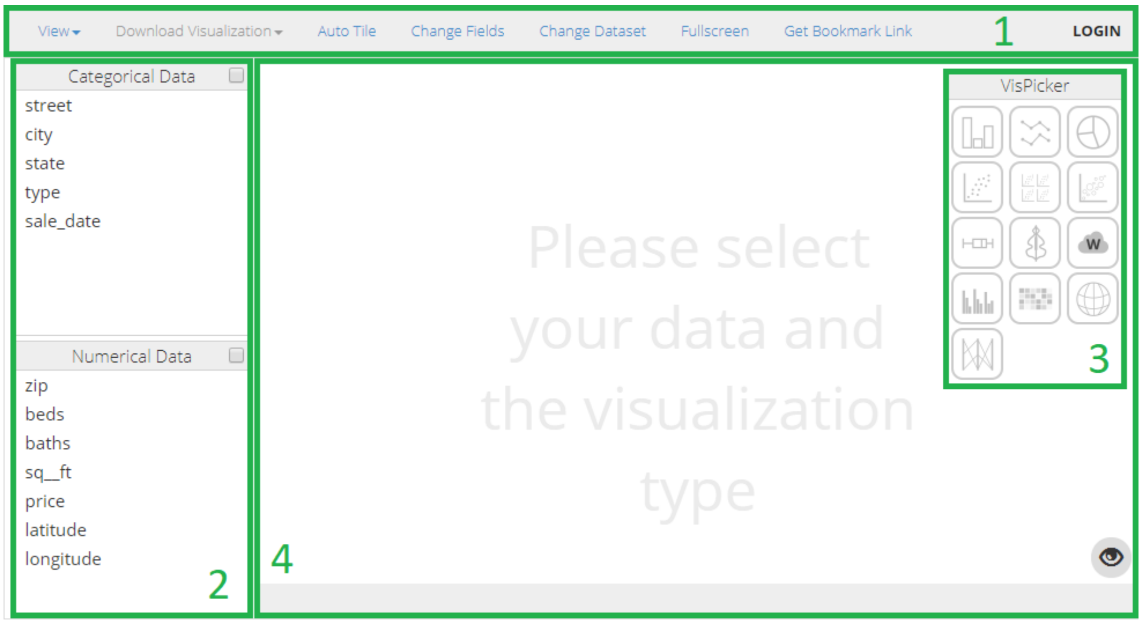


Figure 3.16: The *Visualization Dashboard* with the highlighted regions: 1. Visualization Space 2. Field Selection Area 3. *VisPicker* 4. Toolbar

3.4.1 Guest Dashboard

Visualization Space

The visualization space is the biggest region in the *Visualization Dashboard*. At the beginning, it hosts only the *VisPicker*, the viewing mode toggle in the lower right and an empty visualization bar. Every visualization that is created will be put into this space. The visualization windows and the *VisPicker* can be freely moved in the whole visualization space. The visualization bar, as can be seen in figure 3.16, displays all the created visualizations. An additional functionality of the visualization bar is that, with a click on a visualization tab, the visualization window is minimized to the bar. This is especially useful if a visualization is currently in the way, but is needed in the future. The viewing mode toggle in the lower right corner serves the purpose of getting a better view at the visualizations, for a better view. Everything but the visualization windows is disabled and the visualization windows are locked into place, thus no moving or resizing is possible anymore, as can be seen in figure 3.17. The icon changes from an eye to a pencil to indicate that clicking on the button will change the view back to editing.

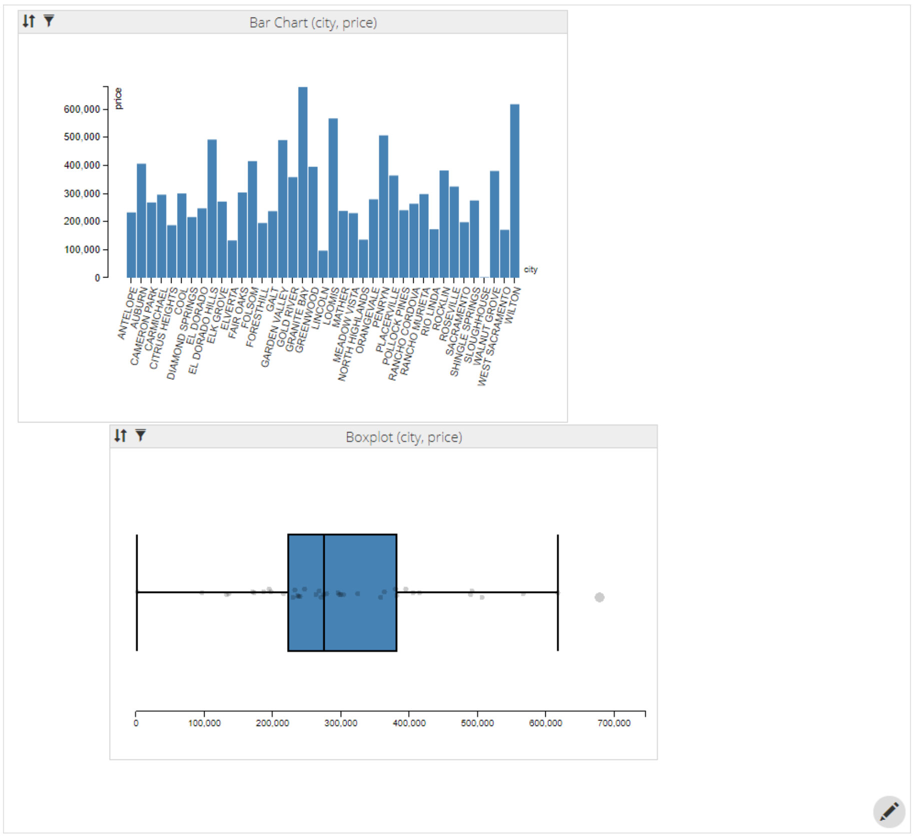


Figure 3.17: Preview mode of the *Visualization Dashboard*

Field Selection Area

The field selection area on the left side of the *Visualization Dashboard* contains all the active field names of the *Dataset Table*. The fields in the field selection are divided into two main categories: categorical data and numerical data. In the categorical data window are all fields which cannot be aggregated. Those fields are the fields with the selected data types of string, location and date. In the numerical data window are the fields which are aggregatable, meaning integers and numbers. The fields are split into those two categories, because all of the visualization distinguish basically only between numerical and non-numerical data, so it presents an easy way to fulfill the requirements of the visualizations, yet it is still possible to have fine categorization for special purpose charts, such as the line chart, which is basically a time series and needs a date type for the x-axis. In order to reveal the actual data type of a field in the categorical or numerical data windows, it is sufficient to hover over the field.
Every time a field is selected the visualizations are checked if they are compatible with the current selection. The process of determining the compatibility of fields for a visualization and mapping the fields correctly on the visual channels of the visualization will be explained in detail in the following section about the *VisPicker*. After every click, the compatible visualizations will be highlighted in the *VisPicker* as active and a visualization can be created.
Additionally, every time a categorical field is selected, a computed field is displayed in the numerical data window. This computed field is a concatenation of the "#" sign and the names of the selected categorical fields. This computed field contains the number of occurrences of each unique tuple of entries for the selected categorical fields.
The main purpose of the field selection area is to find visualizations suited for the fields in which the user is interested, simply by clicking on the fields of interest.

VisPicker

The *VisPicker* is the central piece for the creation of visualizations. It gives the user recommendations for compatible visualizations based on the currently selected fields. The *VisPicker* hosts thirteen different visualizations at the time of writing. At the beginning, when no fields are selected, all of the visualizations are disabled. Starting to select fields in the data selection part, will activate compatible visualizations. This can be seen in figure 3.18. On the left, the *VisPicker* is shown with all of its visualizations disabled. On the right, visualizations are activated which are compatible with fields of the data type string and integer.

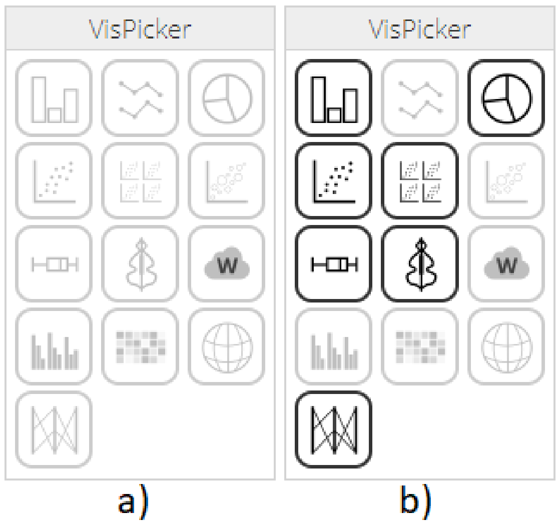
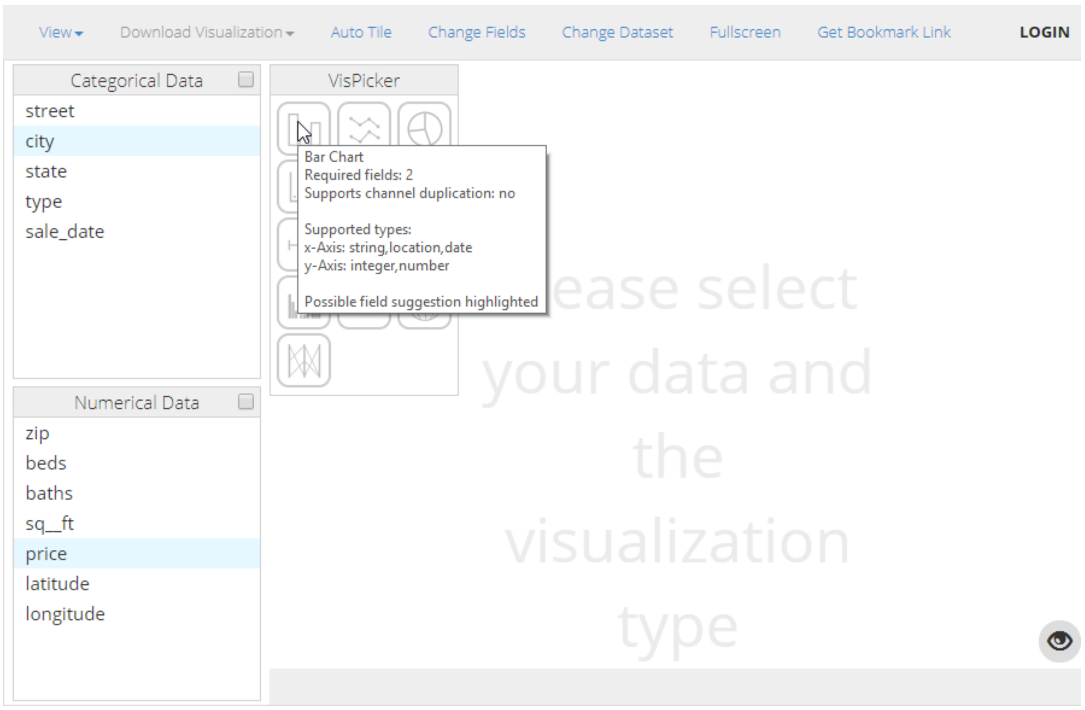


Figure 3.18: *VisPicker* changes based on field selection: a) Nothing selected b) Fields of type string and integer selected

The recommendation of visualizations with the *VisPicker* is actually a two way street. It is not only possible to show compatible visualizations with the selected fields, but it is also possible to receive a recommendation of possible fields for this particular visualization. By hovering with the mouse over a visualization icon, a tooltip will be displayed, explaining how many fields are required for this visualization and what type the fields have to have. Furthermore, if a combination of fields exists to create the visualization, one random possible combination will be highlighted. This can be seen in figure 3.19. The tooltip displays that in order to create a bar chart, it is required to select two fields and that one field has to be of type string, location or date and the other field of type integer or number and that a possible combination has been highlighted. In the data selection section two highlighted fields can be seen.



Bar Chart
Required fields: 2
Supports channel duplication: no

Supported types:
x-Axis: string,location,date
y-Axis: integer,number

Possible field suggestion highlighted

ease select your data and the visualization type

Figure 3.19: *VisPicker*: Tooltip for a visualization and recommendation of compatible fields

In order to create such a mechanism for recommending visualizations based on selected fields and to recommend fields compatible with the visualizations; it was first needed to create a description which defines how fields can be mapped onto visualizations. For this, as a basis, the *Visual Analytics Vocabulary* was taken from Mutlu et al.[Mutlu et al., 2013]. It was converted from an OWL ontology to a simple JSON object and further extended to match all of the needs of *Visualizer*. Every visualization was described by its number of visualization channels and the data types which can be mapped to those visualization channels. Furthermore, there are visualizations which have optional visualization channels, and others where a channel can be duplicated. This information already suffices to describe a visualization in way that a rule based recommendation system can be implemented to recommend compatible visualizations for a selected number of fields. Let us take as an example for a description a bar chart. As visualization channels, it has an x-axis and a y-axis, both of which are mandatory and can only occur once. On the x-axis, only categorical data can be mapped onto, thus the supported data types for this channel are string, location and date. However, the y-axis only supports numerical data, therefore the supported data types are integer and number. For every visualization, there exist additional fields which help speed up the recommendation, like the number of required channels or flags which indicate

if channels can be duplicated. An exhaustive list with the explanations of all the fields in the descriptions of each visualization can be found in Appendix B. Every time, a field is selected or deselected, a check has to be performed if the visualizations are compatible with this configuration or not. As soon as a selection of fields has been made, an array with the selected data types and the field names is created. Before trying out all the possible combinations for all of the visualizations, the visualizations are filtered by checking the requirement of the number of channels. Only the visualizations which can hold all of the selected fields are further examined. Every visualization which meets the channel number requirement has to be checked at this point if a mapping of fields to visualization channels exists. This can be seen as a schema matching problem according to Rahm et al.[Rahm and Bernstein, 2001]. Meaning, the mapping from fields to channels.
In order to perform this check, a brute force approach to this problem is taken. For every visualization, the fields are permuted and checked against the configuration of the visualization channels. If all the types match, the permutation process is stopped and the current mapping from fields to channels is returned. If all permutations get performed with no match, the visualization is not compatible. The problem with this approach lies in the worst case runtime of trying out all the permutations for the selected fields, which is `O(n!)`. Due to the runtime, the maximum number of fields, which can be selected, had to be reduced to ten. There are currently only two visualizations in *Visualizer* which can take more than four fields, and those visualizations are scatterplot matrix and parallel coordinates. A remarkable advantage is that those two visualizations accept almost all data types for their visual channels; thus a mapping can be found very fast. For the scatterplot matrix, the mapping is found in the first iteration, because it has only one channel that can be duplicated and it accepts all data types. Parallel coordinates on the other hand, require that at least two fields get selected and one has to be a category.
Figure 3.20 illustrates the matching process for a bar chart visually for easier understanding. In the illustration it is assumed that a field of type string has been selected and a field of type integer. As can be seen, on the first try, the mapping is not compatible, because neither a string can be mapped on the y-Axis, nor an integer on the x-Axis. In the second round a compatible match is found. As was already mentioned, only the fields are permuted, while the visualization channels stay in place.

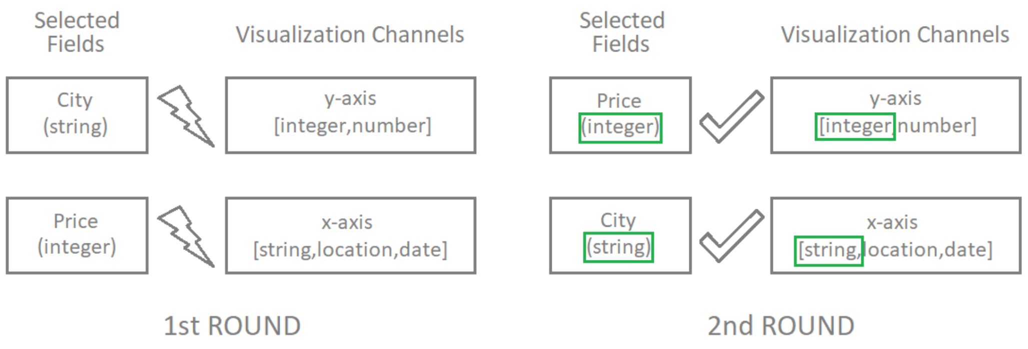


Figure 3.20: Testing mapping combinations for bar chart with a match

Figure 3.21 on the other hand, shows a visualization which matches with the number of required channels, but a mapping could not be found. The visualization in question is a world map, which needs the data type location for its country channel.

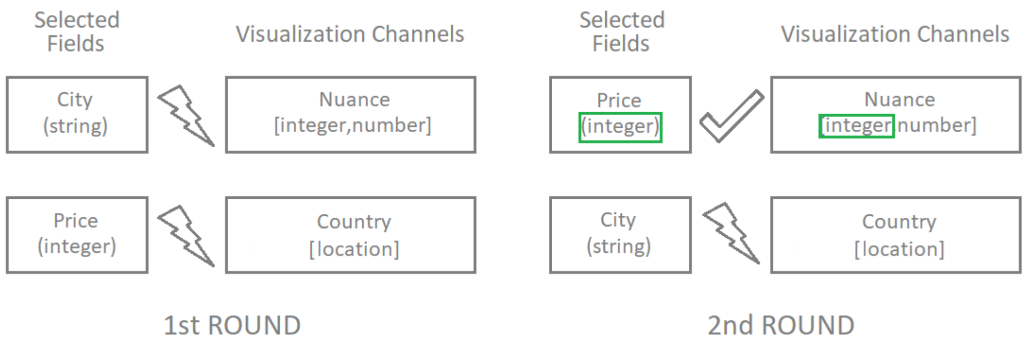


Figure 3.21: Testing mapping combinations for world map without a match

After those checks have been performed, the *VisPicker* displays the visualizations compatible with the selected fields as active, like in figure 3.18 b). Clicking on an active visualization icon will open a dialog to prompt the user to select an aggregation method for the numerical values. In case the dataset has too many entries, after selecting the aggregation an additional warning dialog will get displayed, offering the user an option to reduce the number of data points before creating the visualization. The warning threshold for the number of data points is set per visualization. After the aggregation has been selected and eventually the data points reduced to a displayable number, the visualization is created in the visualization window.

Visualization Windows

After the *VisPicker* has been explained, the visualization windows created by the *VisPicker* will be now analyzed. Figure 3.22 shows a scatter plot created with the *VisPicker*. The data in the scatter plot depicts average sizes and prices of apartments in different places of the Sacramento area.

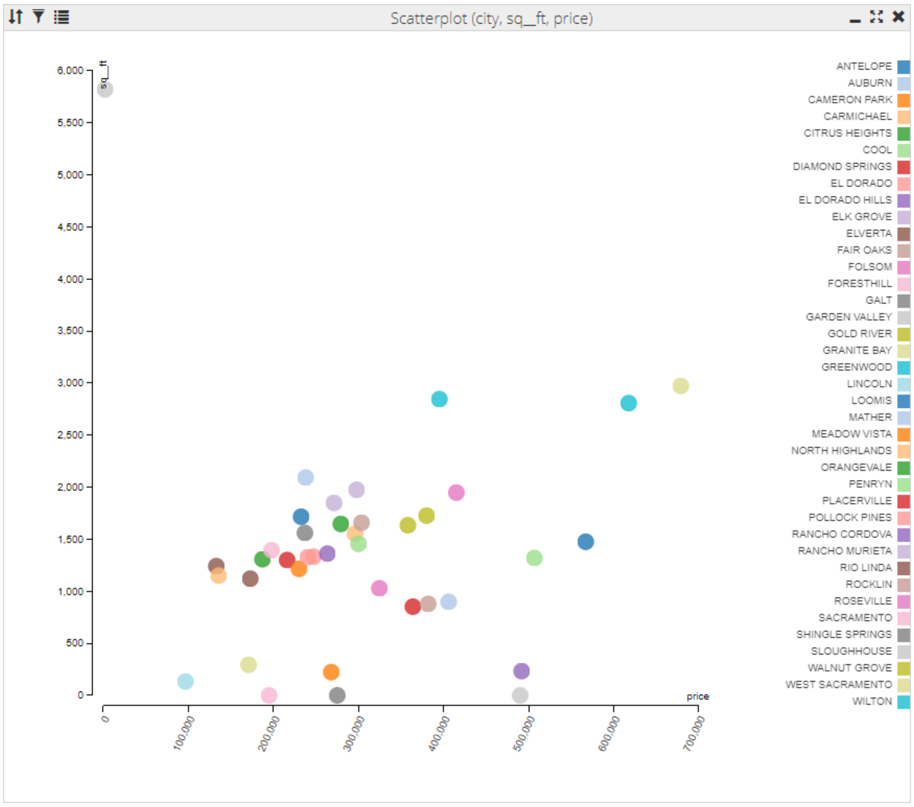
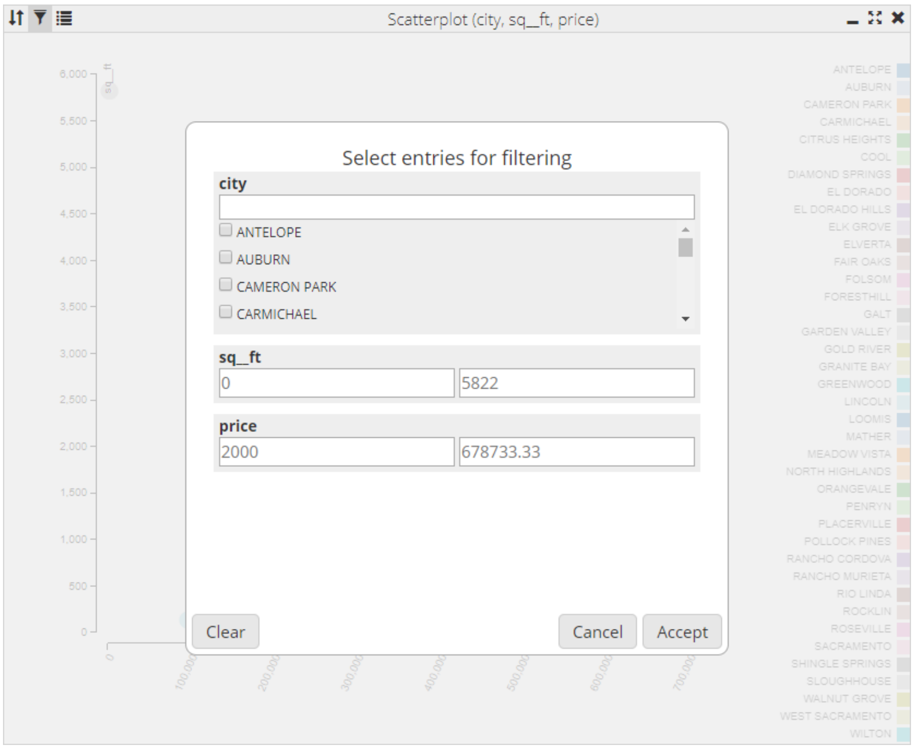


Figure 3.22: Scatter plot created with *VisPicker*

As it can be seen in figure 3.22, the visualization in fact resides in a visualization window. This window can be freely moved around the visualization space or resized, exactly the way a window is expected to behave. In the title bar of the visualization window are a number of different buttons which allow an interaction with the visualization window. The three buttons residing in the top right hand corner of the visualization window will be automatically recognized by most users, especially the users using the Windows operating systems. Those three buttons are from right to left: (i) for closing the visualization window, (ii) maximizing the window - use all available space on the web page, and (iii) minimizing the window - the window is put in the visualization bar. The windows can also be renamed by double clicking the title of the visualization window.

The three buttons on the left are for manipulating the data, displayed in the visualization. The leftmost button is for sorting. Clicking on it will open a context menu which presents all the fields which can be sorted. The second button from the left in the title bar is for filtering. Clicking on the filtering icon will open a dialog that can be seen in figure 3.23.


Scatterplot (city, sq_ft, price)

Select entries for filtering

city
- [ ] ANTELOPE
- [ ] AUBURN
- [ ] CAMERON PARK
- [ ] CARMICHAEL

sq_ft: 0 – 5822

price: 2000 – 678733.33

Figure 3.23: Filter dialog of a visualization

The filter dialog shows every field of the visualization in a separate region. For categorical data, entries of interest can be selected by checking the checkbox beside the entry. The text field on top of all the entries is for filtering in order to quickly find certain entries. Numerical fields can be filtered by giving a range. As a help, the minimum and maximum values of the numerical fields are given as a placeholder. The third button from the left is for channel remapping. Only visualizations which have multiple mappings of fields onto channels have this button; this is the case of the scatter plot. Clicking on this button will open a dialog with the current selection of fields to visual channels. This can be seen in figure 3.24.

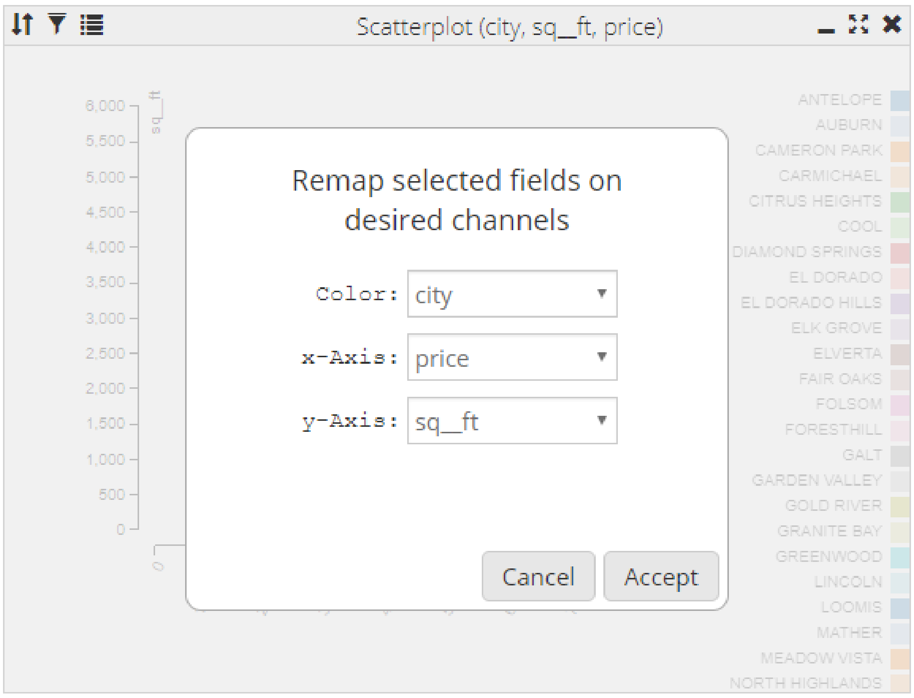


Figure 3.24: Channel remap dialog of a visualization

Only valid remaps are allowed. Thus, opening the select box for “Color”, will display nothing except *city*, but opening the select box for “x-axis”, will show *price* and *sq_ft* (square feet), indicating that a swap is possible. See figure 3.25.

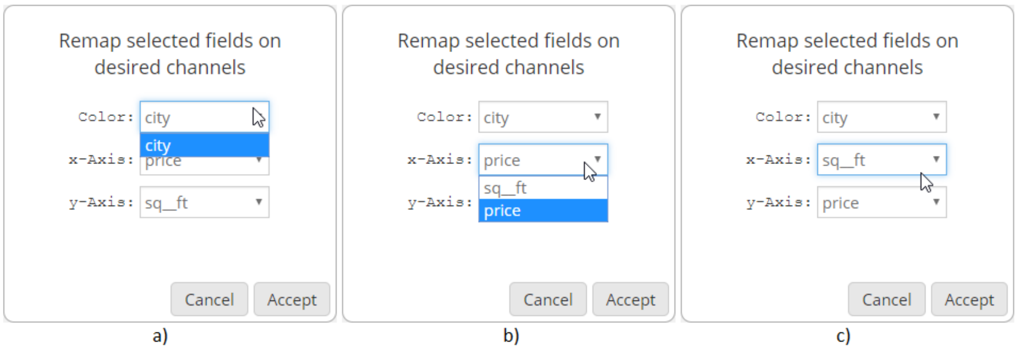


Figure 3.25: Only fields compatible with a channel are displayed. a) Color has only city b) x-Axis allows a swap of price and sq_ft c) selecting a field swaps it automatically

In order to support the visual exploration of data, every visualization is linked with all the other visualizations. Selecting data points in one visualization will highlight matching data points in others. This is called brushing. Figure 3.26

depicts such a brush between a scatter plot and a bar chart. A region in the scatter plot is selected, and the matching bars in the bar chart are highlighted.

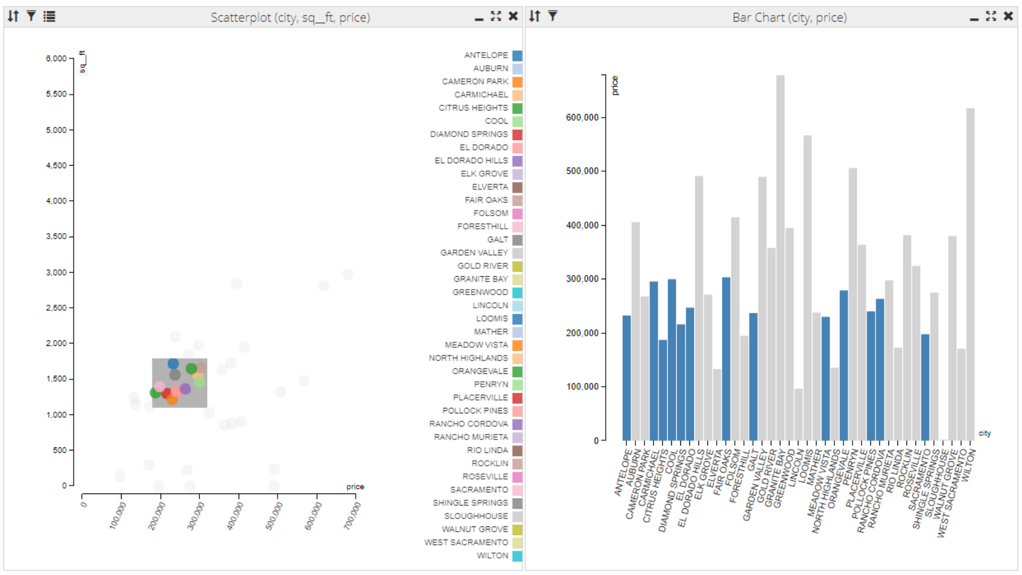


Figure 3.26: Selecting data points in one visualization will highlight matching data points in other visualizations

Toolbar

The toolbar, which can be seen in figure 3.27, was left as the last part of the *Visualization Dashboard*, because the actions performed by the toolbar can influence all the other parts; therefore it was necessary to introduce the rest beforehand.

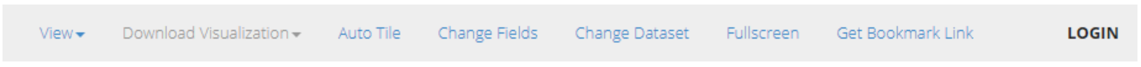


Figure 3.27: Toolbar with limited features for guests

View opens a context menu in which the user can hide the *VisPicker* and the field selection area to extend the visualization space.
Download Visualization shows a list of all opened visualization windows and by clicking on an entry in that list the download of the visualization automatically starts. The images have a resolution of 3000px of the larger side with the aspect ratio kept as it is in the visualization. Because the visualizations are created with SVG elements, upscaling them does not make the image blurry.
Auto Tile orders all the visualizations in the visualization space in a way that

they are all visible and do not overlap. This layout is called ’Cascading Grid’ layout and is especially useful when creating multiple visualizations in a row, because it is possible to order them in a non-overlapping way in just one click.
Change Fields brings the user back to the *Dataset Table* to make changes on the dataset. This does not clear the *Visualization Dashboard*, i.e. it is possible to change back and forth between the *Visualization Dashboard* and the *Dataset Table* while keeping the previously created visualizations. Change Dataset drops all the changes that have been done on the dataset and all the visualizations that have been created and navigates the user back to the file selection screen.
Fullscreen enters the fullscreen mode of the browser, which gives the user even more space for visualizations.
Get Bookmark Link extends the functionality of the “Get Link” functionality of the *Dataset Table*. In the *Dataset Table*, upon clicking on “Get Link”, the current state of the dataset is saved and a configuration link is created. With this configuration link it is possible to apply always the same transformations on a dataset with the same structure. “Get Bookmark Link” gets one step further and in addition to the saved transformations on the dataset, it saves the state of the dashboard in a link. The state of the dashboard includes all created visualizations, the positions and sizes of the visualizations, the current channel mapping, the selected aggregations for the numerical values and even the saved brush. This is done similarly to when the configuration link gets created, by defining visualization actions and pushing them to the action stack when the “Get Bookmark Link” button is pressed. There are two new actions in addition to the actions of the *Dataset Table*; those are the actions for the created visualizations and for the currently used brush. Appendix A describes in detail how the visualization and brushing actions are structured when creating the bookmark link. With this bookmark link it is possible to create the same visualizations with the same aggregations and positions for different datasets with the same structure in an instant. When the bookmark link is pasted in the address bar of the web browser, it is only needed to add a dataset and all the steps performed get recreated.
Login opens a login dialog. If the user does not already have an account, he can open the registration dialog and create a new one. Both, the login dialog and the registration dialog can be seen in Figure 3.28. After the user has logged in, he gets presented with the *Personalized Dashboard*.

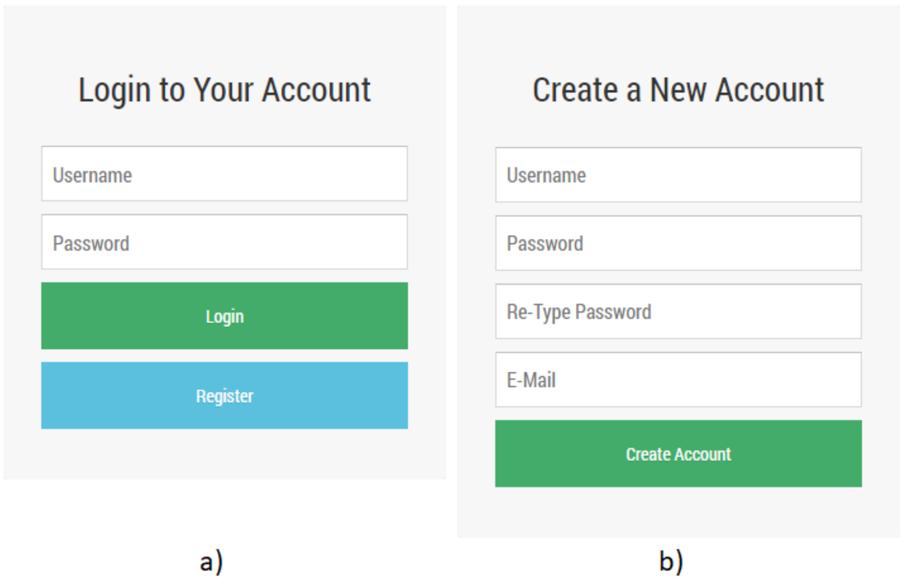


Figure 3.28: a) Login dialog, and b) Registration dialog

3.4.2 Personalized Dashboard

The user login unlocks additional functionalities in the *Visualization Dashboard* which are highlighted in figure 3.29. It is worth noting that the toolbar has additional entries, namely “Customize”, “Upload Visualization”, “Remove Visualization” and “Get Recommendation”. Instead of the “Login” button in the toolbar, the name of the logged in user gets displayed. Furthermore, the *VisPicker* has a toggle in its title bar, which allows the user to switch to the recommended visualizations.

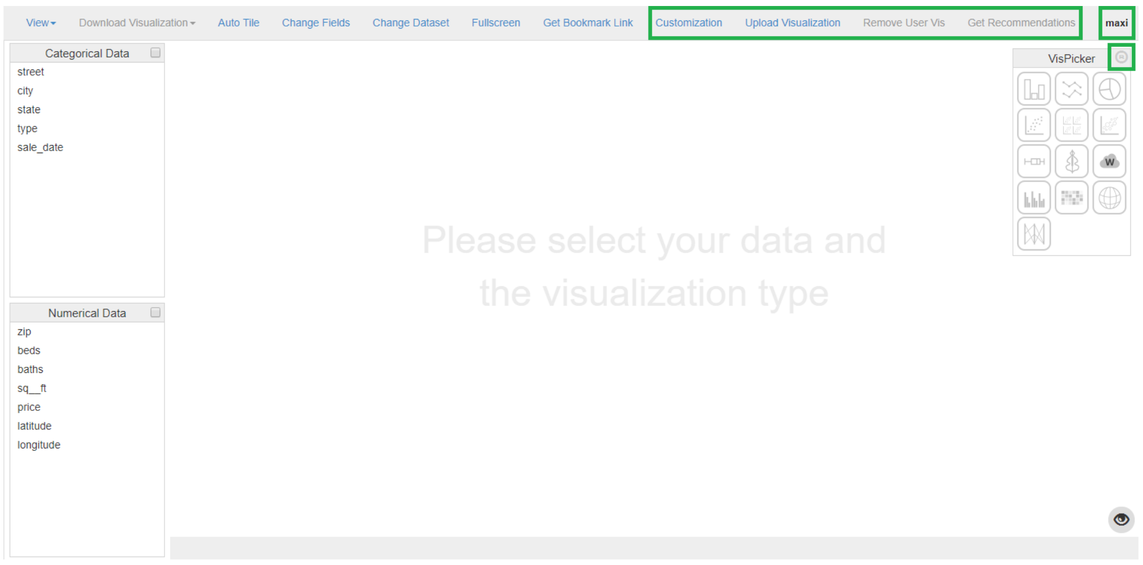
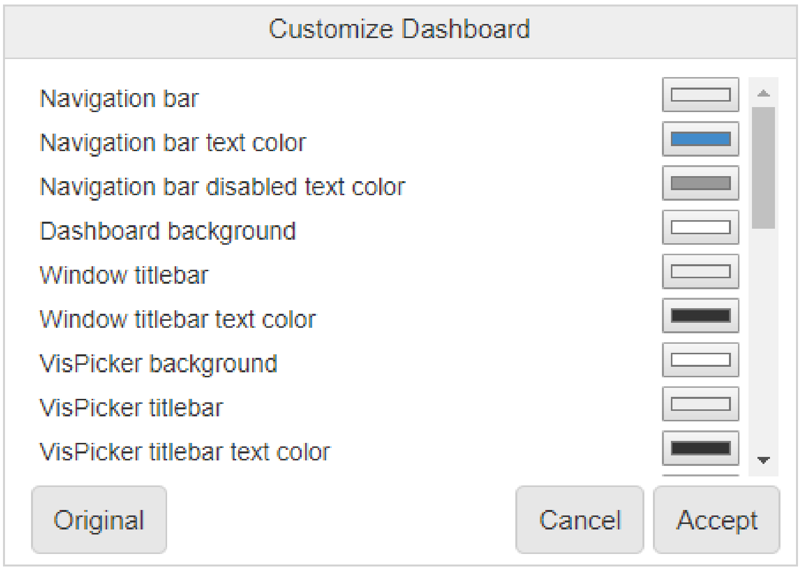


Figure 3.29: *Personalized Dashboard* with additional highlighted features compared to the *Guest Dashboard*

Customize Dashboard

"Customize Dashboard" offers the user an option to tailor the *Visualization Dashboard* styling to his preferences. Clicking on the "Customize Dashboard" button opens a dialog, which contains a great number of different entries which can be modified, as it can be seen in figure 3.30.


Customize Dashboard

Navigation bar	⬜
Navigation bar text color	🟦
Navigation bar disabled text color	▨
Dashboard background	⬜
Window titlebar	⬜
Window titlebar text color	⬛
VisPicker background	⬜
VisPicker titlebar	⬜
VisPicker titlebar text color	⬛

Figure 3.30: Customize Dashboard dialog

The styling granularity is really fine. All of the colors of the dashboard can be changed separately from each other and the fonts and font-sizes are also not bound to each other. Saving the styling configuration in the *Visualization Dashboard* will also save this styling configuration on the server in the user's profile. Thus, when the user logs out and logs in again, he will get presented with his saved stylesheet. Furthermore, it is always possible to revert the styling changes to default if the user is not satisfied with the outcome of his changes

Upload new Visualization

The most prominent feature of the *Personalized Dashboard* is the option to integrate user written visualizations seamlessly into *Visualizer*. Even though *Visualizer* already has thirteen different visualizations, it might be possible that a user needs a very specific one, tailored to a certain use case, which is not included. Therefore, *Visualizer* offers an API, which can be implemented so that new visualizations integrated into *Visualizer* can use all the functionalities offered by it. Those functionalities include automatic channel swapping, filtering, brushing and visualization downloading. The visualization created by the user can also be bookmarked and sent to someone else and it will also work, as long as the other person has the same integrated visualization. If a user wants to upload his own visualization, the visualization plug-in has to have a certain structure. All of the contents of the visualization have to be in one directory, which can have any arbitrary name. This directory has to have at least the folders called *css* and *libs*, a main .js file with an arbitrary name and an icon with the name "icon.png" like in figure 3.31.

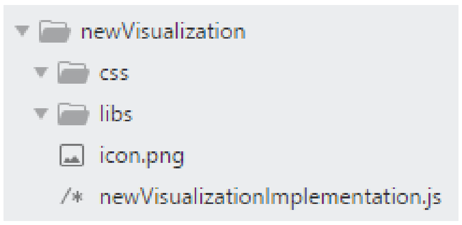


Figure 3.31: Visualization folder structure

The *css* folder holds all stylesheets which will be automatically included when the custom visualization gets generated. The *libs* folder is for all the JavaScript libraries which the visualization needs. Currently it is only possible to put JavaScript files into this folder. Creating subfolders in either the *css* or *libs* folder will cause errors. It is possible to avoid those errors by loading the desired script manually. The *libs* folder should generally be

used for all additional JavaScript files, apart from the main file, irrespective if they are third party libraries, or own written JavaScript files. The *icon.png* will be used as the icon for the visualization. The file type and filename of the icon is mandatory. As a recommendation, the icon should have a size of 256 x 256 pixels. The *newVisualizationImplementation.js* represents the main implementation file for the visualization functionality. This file can actually have any arbitrary name which will be specified when uploading the visualization. This file should implement the API of *Visualizer*.
Every visualization created by *Visualizer* is drawn into its own iframe. This means that the creator of a new visualization can use all the libraries and styles he wishes, without having to worry that they will interfere with the main page. All of *Visualizer's* current visualizations were created using the D3 library, which uses SVG elements to create graphic elements, but it is up to the user to use any libraries; thus it would be also possible to create visualizations with WebGL. The visualizations which are created with SVG elements should append the SVG element always directly to the body element. This ensures that the Download Visualization functionality works as it should. Visualizations utilizing the HTML5 canvas should also append the canvas directly to the body tag. Table 3.4 lists the functions of the plugin API which should be implemented to ensure the visualization works.

Function name	*Parameters*	*Description*
drawVisualization	datarows, channelMappings, visIndex	Called every time the visualization has to be drawn
applyFilter	filteredDatarows	Called when a new filter is set to the visualization
applyCssToSvg	(none)	Called shortly before the download of the visualization starts
revertCssFromSvg	(none)	Called after the download of the visualization has finished

Table 3.4: Visualization API description

drawVisualization - Only the `drawVisualization` method is mandatory so that the visualization can be displayed. The other methods do not have to be implemented, but it is strongly recommended. Otherwise, the visualization will not support all the functionalities. The parameters of the `drawVisualization` method have the following structure:

- `datarows`: This is a two-dimensional array holding the information of a subset of columns of the main dataset. The data is presented as rows and not as columns, because most of the visualizations, which were integrated in *Visualizer*, expect the data to be in this form.

- `channelMappings`: This parameter is an array of objects. It describes what fields were mapped onto which visual channels of the visualization, which data type the fields have and which aggregation was used on this field. This array has the same length, as each row in the `datarows` object. Table 3.5 shows the structure of the `channelMappings` parameter.

Attribute name	*Type*	*Description*
aggregation	string	Aggregation used on this field. Possible values: min, max, avg, sum and group (only for categorical fields)
channel	string	Visualization channel on which the field should be mapped
datatype	string	Data type of the field
label	string	Name of the selected field

Table 3.5: `channelMappings` object description

- `visIndex`: The index of the currently created visualization. It is actually only used as a parameter for the `brushingObserver` so that it knows which visualization to update. The `brushingObserver` will be explained in detail later on in this subsection.

applyFilter - In order to keep the logic separated from the drawing, the data filtering mechanism is created outside the visualization. Thus, the `applyFilter` method should only apply the already filtered data to the visualization. As a single parameter, it receives an array of data rows, which fit into the constraints of the filtering done outside of the visualization and it should update the current visualization accordingly. The data points in the visualization should be filtered, i.e. the data points which are not in the data rows parameter of the `applyFilter` method should be hidden, and not the dataset of the visualization simply limited and redrawn. Figure 3.32 shows a not filtered visualization on the left. In the middle is a correctly filtered visualization and on the right a wrongly filtered one, with the same filter applied.

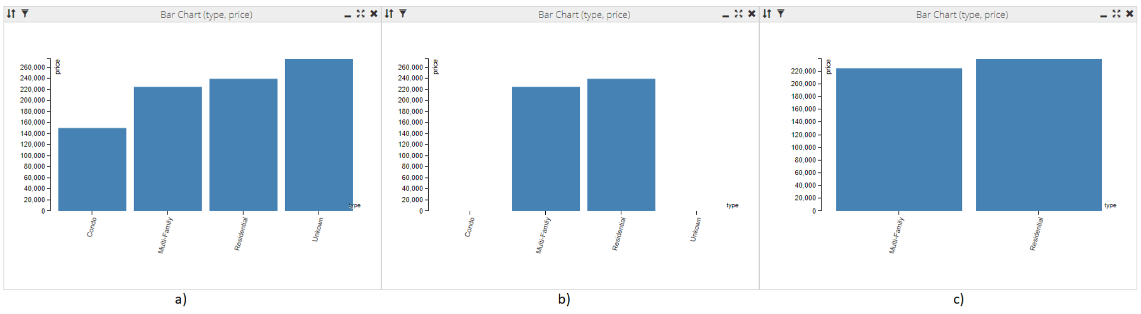


Figure 3.32: a) not filtered bar chart, b) correctly filtered bar chart c) wrongly filtered bar chart

applyCssToSvg and **revertCssFromSvg** - Those two methods are only called before and after the download of the visualization. The purpose of those two methods is to apply styles directly to the SVG elements before a download, and to remove them after. If a visualization has to be downloaded, it has to be rendered into a canvas before. Since the used library canvg can only use styles, which are inline, the styles have to applied there. The revert method is called to revert the changes on time if they might have a negative impact on the visualizations.
brushingObserver - The `brushingObserver` is a global helper object, which all of the visualizations can use. A newly created visualization has the option to register itself to the `brushingObserver` and pass to it its `visIndex` and a callback function. With the help of the `brushingObserver`, the visualization has the option, every time a selection has been done in this visualization, to notify the other visualizations to update their view. Consequently, the visualization also passes a callback to the `brushingObserver`, which is called every time another visualization updates their view. In order to be able to use the `brushingObserver`, a visualization has to register itself in the `brushingObserver`. This is done by calling the `registerListener` method of the `brushingObserver`. The `registerListener` method expects three parameters: (i) a callback function for updating the `visIndex` (ii) the `visIndex` of the current visualization, and (iii) a callback function which is called when an update happened. The passed callback function for updating the visualizations receives two parameters when it is called, the selected data rows and a list of field names. With that information the visualization can update its view to the selected data from another visualization. For example, it can highlight certain data points in the visualization. The update method in the `brushingObserver` is called to signalize that a selection has happened. This method has four parameters: (i) `visIndex` of the current visualization, (ii) selected data rows, (iii) selected dimensions - deprecated, always null and (iv) the list of field names.

Every time a data point is highlighted in a visualization or a number of data points get selected, this method should be called. There are also two other methods in the brushing observer, which are notable:

- `unregister(visIndex)` - unregisters the visualization from the brushing Observer. Should be done upon closing.
- `updateEmpty(visIndex)` - updates other visualizations with an empty selection.

After the visualization has been finished, and the visualization folder has the same structure, which was already described at the start of the subsection about the integration of visualizations; the folder can be zipped and uploaded into *Visualizer*. Clicking the "Upload Visualization" button will open a dialog which prompts the user to specify the configuration of the visualization. As can be seen in figure 3.33 in the dialog, the user is prompted to add the zipped package and to specify the structure of the visualization.

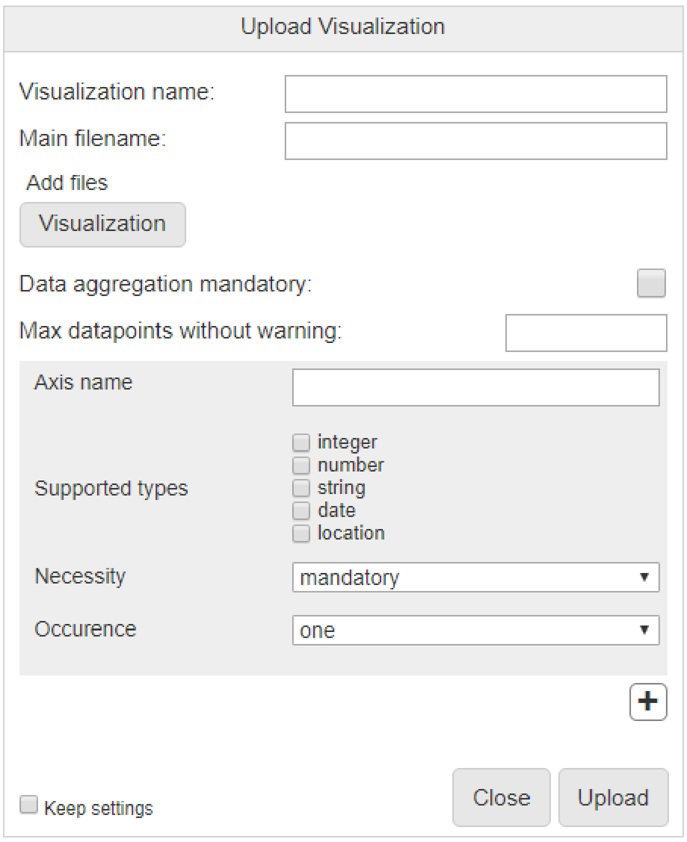


Figure 3.33: Upload Visualization dialog

This structure specification includes: (i) the name which should be used for the visualization in *Visualizer*, (ii) the name of the main file and the configuration for the visual channels, (iii) a checkbox if aggregation is mandatory, (iv) the limit for the maximum number of data points before a warning is issued, and (v) the visualization channel specification. The description of every channel is separated visually by being in a little box. The channel description is needed for the rule based recommender, so that it can toggle the visualization when the criteria for the visualization are met. As mentioned in the subsection about the chart ontology description, for every channel it has to be specified what data types are supported, if the channel is mandatory and if it can be duplicated. Additionally, it is also possible to give a name to the visualization channel. The name for the visualization channel is only used inside the newly uploaded visualization. After all the information has been provided, the visualization can be uploaded. As soon as it has been processed on the server, the newly created visualization is integrated into the *VisPicker*. As can be seen in figure 3.34, the visualizations created by the user have an overlay icon to give the user a visual cue that it is not part of *Visualizer's* default visualizations.

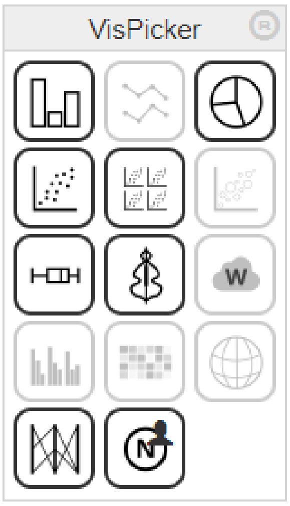


Figure 3.34: *VisPicker* with a user added visualization

Remove User Visualization

The "Remove User Vis" button is only enabled if at least one user uploaded visualization is present in *Visualizer*. It enables the removal of the custom visualizations. This can be done either to replace a custom visualization with a newly updated one, or to reduce the number of custom visualization to declutter the *VisPicker*. In figure 3.35 the "Remove User Visualization" dialog can be seen. It consists only of a selectbox with a list of custom visualization. Selecting one visualization and clicking on accept will remove it from the user folder on the server and automatically update *VisPicker*.

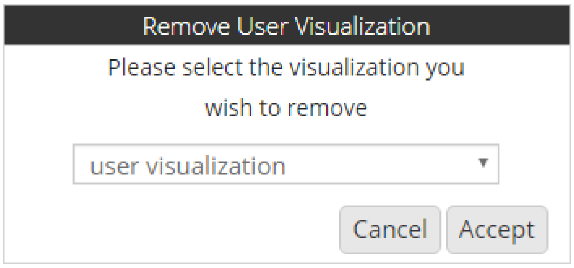

Remove User Visualization
Please select the visualization you wish to remove
user visualization
Cancel
Accept

Figure 3.35: Remove User Visualization dialog

Remove User Visualization

As a part of its personalized component, *Visualizer* also supports personalized recommendations for visualizations. This component is still at an experimental stage. There are multiple reasons for this, but the most important ones are, that the personalized visualization recommender service is still being developed and that the use case for the usage of the recommender is not the exact same one the recommender expects. The recommender was actually designed to be fed a whole dataset at once, to analyze it itself, and based on the dataset alone to generate a list of visualization recommendations for this dataset. In *Visualizer* this step is skipped and the recommender directly receives the compatible visualizations and mappings from fields to visualization channels. Furthermore, the personalized visualization recommender is at its infancy and consequently lacks enough data for accurate recommendations.
The personalized recommender can be used similarly like the rule based recommender by first selecting the fields of interest. The only difference is that the user has to explicitly demand a recommendation by clicking on the "Get Recommendations" button. Clicking the button opens a dialog, which can be seen in figure 3.36, prompting the user to enter tags before getting recommendations.

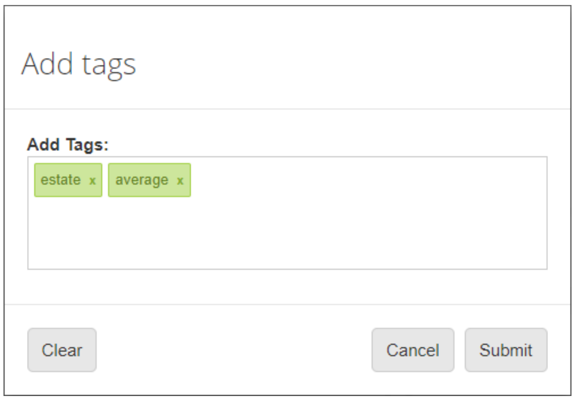


Figure 3.36: Add tags dialog

Those tags can be seen as a kind of search query. The entered tags represent the topics of interest. Clicking on “Accept” in the dialog will send the tags and all the mappings from the fields to visual channels to the recommender service. Based on those topics and the selected fields, the recommender generates its ranking for the compatible visualizations. On the client’s side, the user will notice that after sending the request for a recommendation, in the *VisPicker’s* title bar a loading indicator is displayed. This loading indicator shows that a visualization recommendation is pending. When the recommendation arrives, the indicator is removed and an active **R** can be seen. Clicking on this button will change the view in *VisPicker* to the received ranked visualizations, which can be seen in figure 3.37.

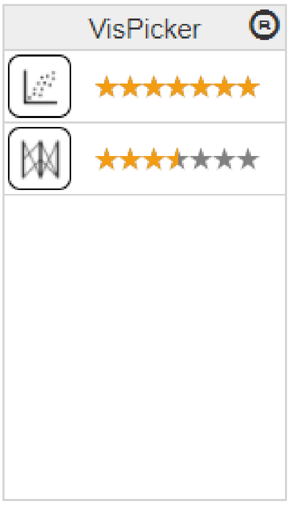


Figure 3.37: *VisPicker* with ranked visualizations

The visualizations are ranked with 1 to 7 stars. 1 indicates that the fields are not suited for this visualization, while 7 indicates that the fields are well suited for this visualization. The recommended visualizations already have a set mapping from fields to channels, thus clicking a recommended visualization will open it with this recommended configuration. Figure 3.38 shows such a visualization created from a visualization recommendation.

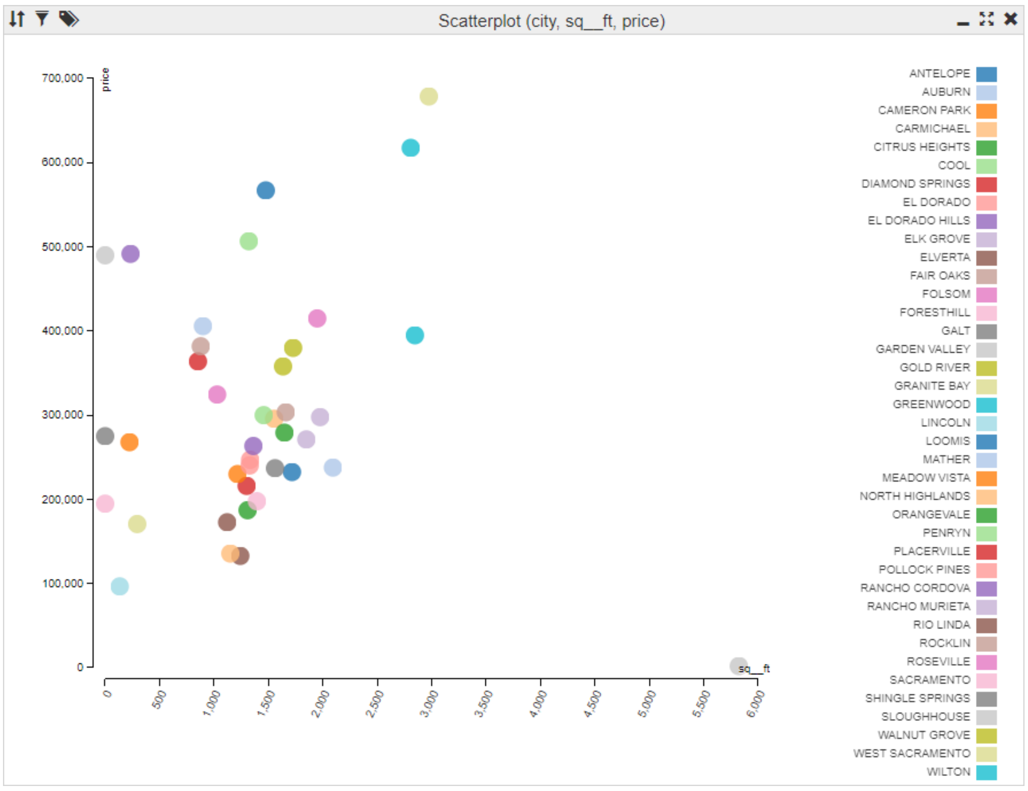
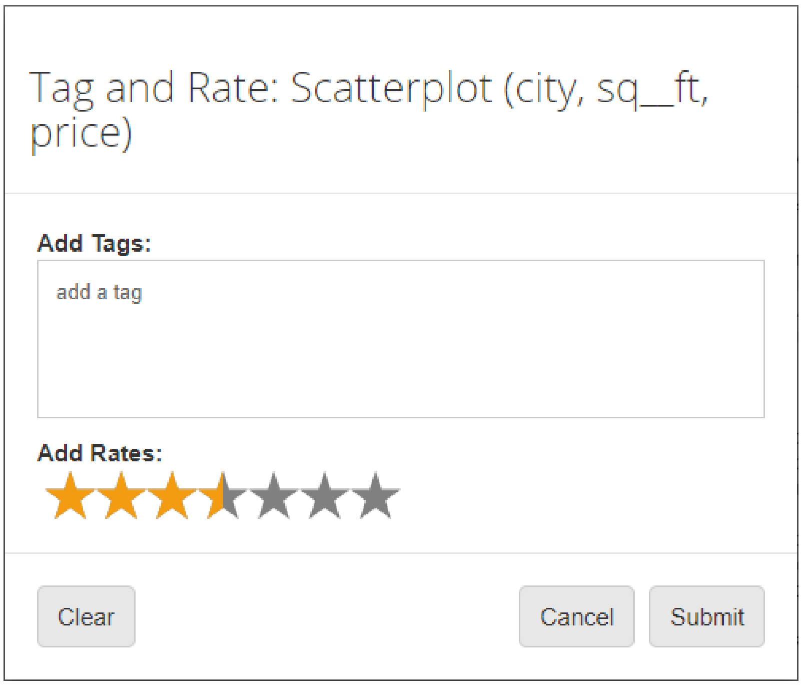


Figure 3.38: Visualization created through the recommended ranked visualizations

For the most part, the visualization looks the same as the others, but there is a slight difference in the title bar of the visualization. First, the option to remap the channels is gone, because the channel mapping is set as a part of the recommendation and second there is an additional button. Clicking these buttons opens a new feedback dialog (figure 3.39) for this recommended visualization. In this feedback dialog, the user can again enter tags and additionally to the tags it is also possible to rate the recommended visualization from 1-7 on a Likert-scale[Likert, 1932]. The tags and the rating are sent to the recommender as a feedback which the recommender can use to improve its visualization recommendations in the future.


Tag and Rate: Scatterplot (city, sq_ft, price)

Add Tags:

add a tag

Add Rates:

Figure 3.39: Tag and Rate dialog: This dialog can be used to give feedback for the generated recommended visualization

3.5 Summary

This chapter gave an overview over *Visualizer* as a whole. In addition to the explanations of every little detail on the front end and the back end, the architecture of the web app and the external services were underlined as well. However, knowing what is included in the tool does not necessarily mean to know how to use it, or in what this tool excels. Therefore the next chapter will cover use cases, which will demonstrate what makes the tool something special.

Chapter 4

Applications

This section will demonstrate the major applications of *Visualizer* based on three scenarios. The first scenario will show how to approach the visual exploration of a dataset. Data will be loaded into *Visualizer*, the data will be prepared and then visualizations will be created in the *Visualization Dashboard* to detect patterns in the data. Based on this scenario the general usage will be presented. The second scenario will demonstrate how it is possible to integrate *Visualizer* into an existing web page and configure it to display visualizations for different datasets with a set schema. This scenario is interesting if *Visualizer* should be used with datasets, which are always generated by the same schema, but only have different content. An example would be the learning behaviour of a student during a course. The third scenario will show how to extend *Visualizer's* basic visualization set, by writing and uploading a user defined visualization.
For the purposes of all the demonstrations in this section, the Open University Learning Analytics dataset[1] was used, specifically the studentInfo.csv, which contains various information about students taking courses in a virtual learning environment. The relevant anonymized information includes gender, location, education degree, course taken, age band, grade, disability, number of previous exam attempts and the number of studied credits.

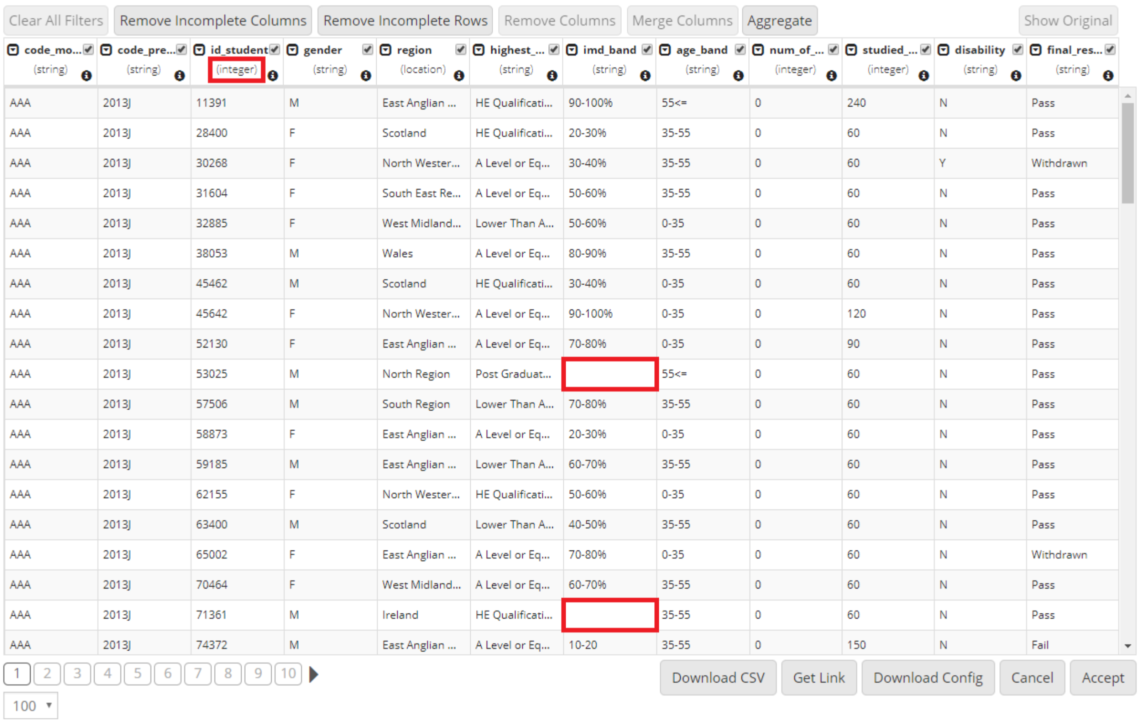
4.1 Data Preparation and Visual Exploration

As mentioned above, the dataset used in this subsection will be the studentInfo.csv of the Open University Learning Analytics dataset. The loaded data will be prepared and after the preparation, the different fields will be visually analyzed in the *Visualization Dashboard*.

[1] `https://analyse.kmi.open.ac.uk/open_dataset`

4.1.1 Data Preparation

When looking at the loaded dataset in figure 4.1, two highlighted parts can be noticed. The first interesting aspect is that the dataset contains empty cells. Generally, empty cells mean missing data, thus two option arise. You can either remove the rows or columns containing those cells, or you can set a value to the empty cells; both of it is possible in *Visualizer*. Even if the empty cells cannot be directly seen, it can be determined that the dataset contains empty cells if the buttons for the removal of incomplete rows or columns are active. For this demonstration the rows containing the empty cells will be removed by clicking on the "Remove Incomplete Rows" button.

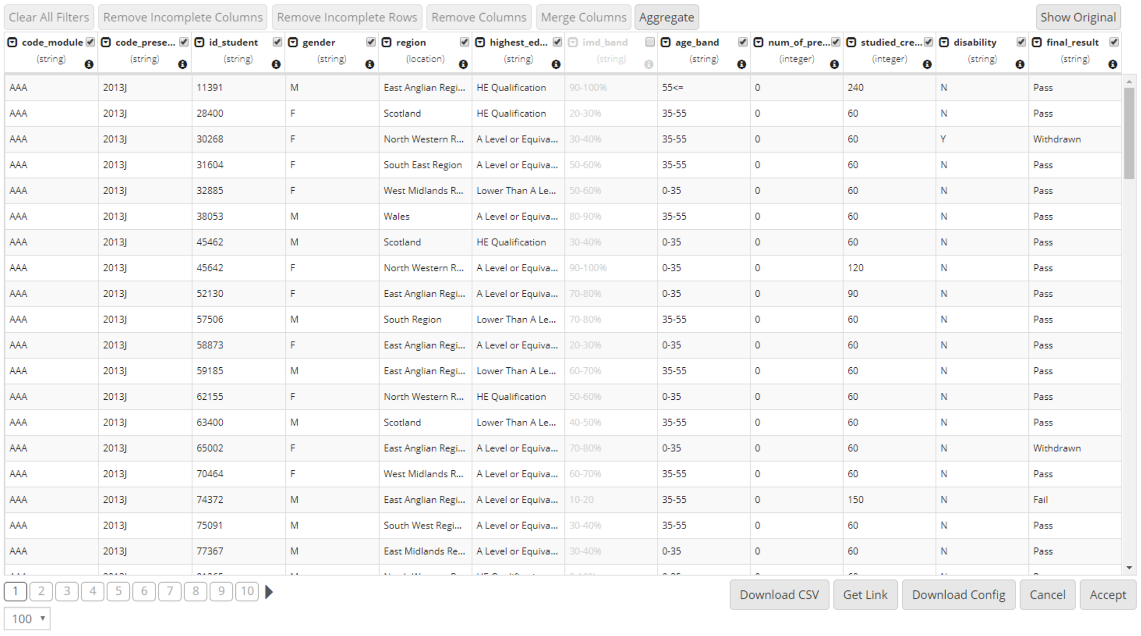


code_mo... (string)	code_pre... (string)	id_student (integer)	gender (string)	region (location)	highest_... (string)	imd_band (string)	age_band (string)	num_of_... (integer)	studied_... (integer)	disability (string)	final_res... (string)
AAA	2013J	11391	M	East Anglian ...	HE Qualificati...	90-100%	55<=	0	240	N	Pass
AAA	2013J	28400	F	Scotland	HE Qualificati...	20-30%	35-55	0	60	N	Pass
AAA	2013J	30268	F	North Wester...	A Level or Eq...	30-40%	35-55	0	60	Y	Withdrawn
AAA	2013J	31604	F	South East Re...	A Level or Eq...	50-60%	35-55	0	60	N	Pass
AAA	2013J	32885	F	West Midland...	Lower Than A...	50-60%	0-35	0	60	N	Pass
AAA	2013J	38053	M	Wales	A Level or Eq...	80-90%	35-55	0	60	N	Pass
AAA	2013J	45462	M	Scotland	HE Qualificati...	30-40%	0-35	0	60	N	Pass
AAA	2013J	45642	F	North Wester...	A Level or Eq...	90-100%	0-35	0	120	N	Pass
AAA	2013J	52130	F	East Anglian ...	A Level or Eq...	70-80%	0-35	0	90	N	Pass
AAA	2013J	53025	M	North Region	Post Graduat...		55<=	0	60	N	Pass
AAA	2013J	57506	M	South Region	Lower Than A...	70-80%	35-55	0	60	N	Pass
AAA	2013J	58873	F	East Anglian ...	A Level or Eq...	20-30%	0-35	0	60	N	Pass
AAA	2013J	59185	M	East Anglian ...	Lower Than A...	60-70%	35-55	0	60	N	Pass
AAA	2013J	62155	F	North Wester...	HE Qualificati...	50-60%	0-35	0	60	N	Pass
AAA	2013J	63400	M	Scotland	Lower Than A...	40-50%	35-55	0	60	N	Pass
AAA	2013J	65002	F	East Anglian ...	A Level or Eq...	70-80%	0-35	0	60	N	Withdrawn
AAA	2013J	70464	F	West Midland...	A Level or Eq...	60-70%	35-55	0	60	N	Pass
AAA	2013J	71361	M	Ireland	HE Qualificati...		35-55	0	60	N	Pass
AAA	2013J	74372	M	East Anglian ...	A Level or Eq...	10-20	35-55	0	150	N	Fail

Figure 4.1: *Dataset Table* with loaded studentInfo.csv

The second highlighted aspect is that the student id under the field `id_student` was detected as belonging to the type integer, which is wrong. The values in this column should actually be detected as categorical fields, but because all the values in this column are numbers, this field is recognized as being of the type integer. To fix this, the type of this field will be changed manually to string, by clicking on the type and selecting in the generated selectbox the type string.
Furthermore, the imd_band field does not contain any valuable information

for non-UK people. When looking at the description of the dataset and the meaning of the field, it says that the contents of the field represent the index of multiple deprivation[2], which is not of interest for this demonstration, so it will be deactivated. After changing the data type of the `id_student` field, removing the rows containing empty cells and deactivating the `imd_band` field, the *Dataset Table* with the loaded dataset will look as in figure 4.2. At this point it is possible to proceed to the *Visualization Dashboard*.



code_module (string)	code_prese... (string)	id_student (string)	gender (string)	region (location)	highest_ed... (string)	imd_band (string)	age_band (string)	num_of_pre... (integer)	studied_cre... (integer)	disability (string)	final_result (string)
AAA	2013J	11391	M	East Anglian Regi...	HE Qualification	90-100%	55<=	0	240	N	Pass
AAA	2013J	28400	F	Scotland	HE Qualification	20-30%	35-55	0	60	N	Pass
AAA	2013J	30268	F	North Western R...	A Level or Equiva...	30-40%	35-55	0	60	Y	Withdrawn
AAA	2013J	31604	F	South East Region	A Level or Equiva...	50-60%	35-55	0	60	N	Pass
AAA	2013J	32885	F	West Midlands R...	Lower Than A Le...	50-60%	0-35	0	60	N	Pass
AAA	2013J	38053	M	Wales	A Level or Equiva...	80-90%	35-55	0	60	N	Pass
AAA	2013J	45462	M	Scotland	HE Qualification	30-40%	0-35	0	60	N	Pass
AAA	2013J	45642	F	North Western R...	A Level or Equiva...	90-100%	0-35	0	120	N	Pass
AAA	2013J	52130	F	East Anglian Regi...	A Level or Equiva...	70-80%	0-35	0	90	N	Pass
AAA	2013J	57506	M	South Region	Lower Than A Le...	70-80%	35-55	0	60	N	Pass
AAA	2013J	58873	F	East Anglian Regi...	A Level or Equiva...	20-30%	0-35	0	60	N	Pass
AAA	2013J	59185	M	East Anglian Regi...	Lower Than A Le...	60-70%	35-55	0	60	N	Pass
AAA	2013J	62155	F	North Western R...	HE Qualification	50-60%	0-35	0	60	N	Pass
AAA	2013J	63400	M	Scotland	Lower Than A Le...	40-50%	35-55	0	60	N	Pass
AAA	2013J	65002	F	East Anglian Regi...	A Level or Equiva...	70-80%	0-35	0	60	N	Withdrawn
AAA	2013J	70464	F	West Midlands R...	A Level or Equiva...	60-70%	35-55	0	60	N	Pass
AAA	2013J	74372	M	East Anglian Regi...	A Level or Equiva...	10-20	35-55	0	150	N	Fail
AAA	2013J	75091	M	South West Regi...	A Level or Equiva...	30-40%	35-55	0	60	N	Pass
AAA	2013J	77367	M	East Midlands Re...	A Level or Equiva...	30-40%	0-35	0	60	N	Pass

Figure 4.2: *Dataset Table* with prepared data from the studentInfo.csv dataset

4.1.2 Visual Data Exploration

Due to the difficulty to see and find patterns in tables, it is much better to resort to visualizations, to aggregate and display data in a form which enhances the possibility to draw conclusions. Even if the user does not know correctly what he is interested in, he has the option to select an arbitrary number of fields which he thinks he is interested in and create visualizations which might lead to new insights. Figure 4.3 shows the *Visualization Dashboard* for the studentInfo.csv dataset after the manipulations on the dataset have been performed. As can be seen, the `id_student` field is in the categorical fields window and `imd_band` is missing.

[2]`https://www.gov.uk/government/statistics/english-indices-of-deprivation-2015`

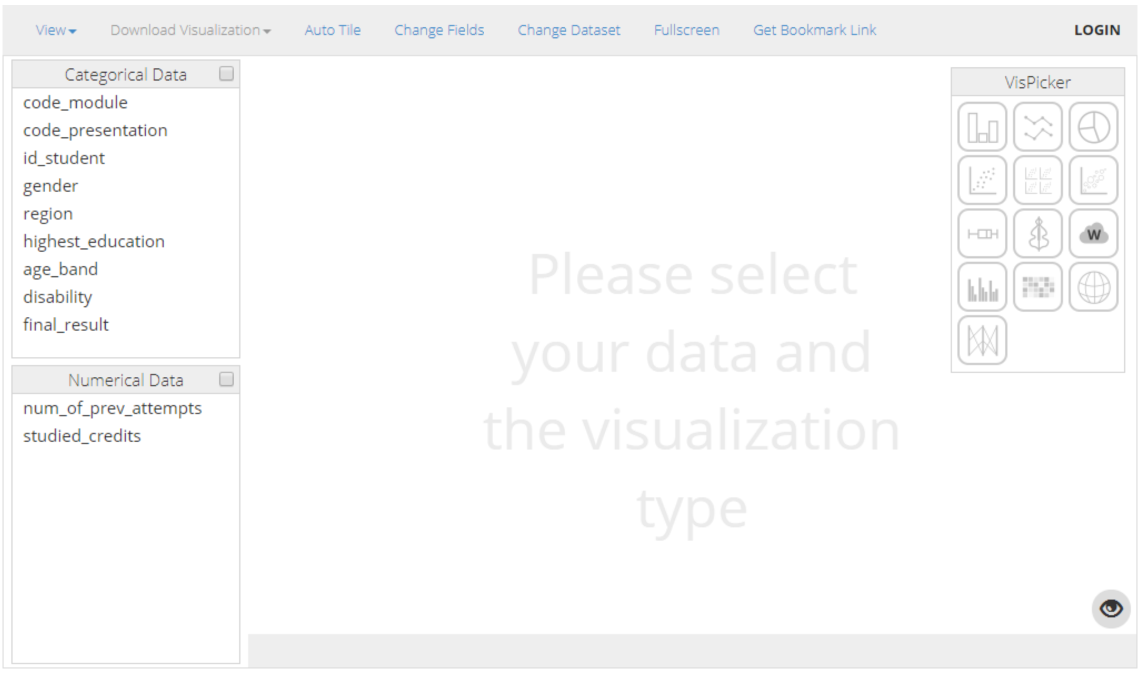


Figure 4.3: *Visualization Dashboard* with loaded data

Composition of class

The first kind of data, which will be explored, are the student compositions of the classes. For this, it is possible to make use of the computed count field. Clicking on a categorical field will create the computed numerical field consisting of a "#" sign and the concatenation of the names of the selected categorical fields. Three bar charts will be created, one to show the number of students per gender, one per age band and one per education. In addition to the bar charts, pie charts were created for the number of students concerning their gender and age band, in order to visually show a ratio of the different groups. The results can be seen in figure 4.4, which shows only the created charts so that the values can be better seen.

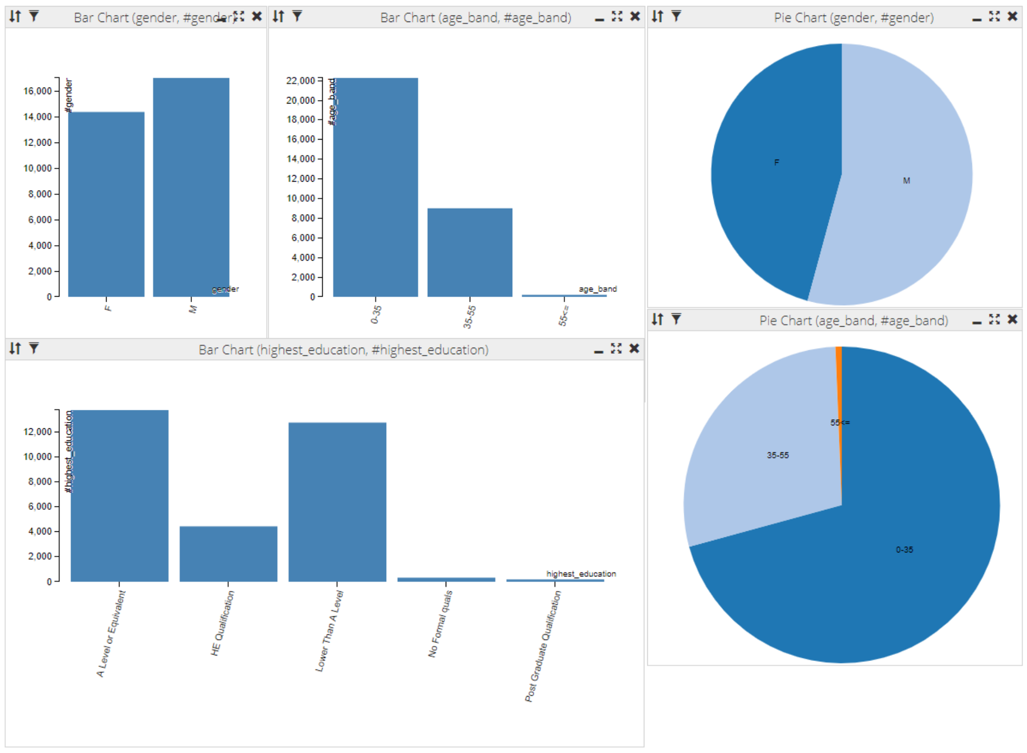


Figure 4.4: Visualizations created to for course student compositions

The bar chart and pie chart for the number of male/female students show that there are noticeably more male students registered for the course. When looking at the age band, it seems that the group of students aged 35 or lower dominate the student composition, and that the group of students below 55 take up almost all of the group. Looking at the education level, most of the students have A level or equivalent education or lower than A level. To clarify, A level education is an educational level specific to the UK which generally takes 12 years of education.

Effort and payoff

For the next example, the numerical fields for the number of previous attempts and studied credits, which are already available, will be explored. For each of those two numerical fields, gender and code module will be selected and a grouped bar chart will be created. Thus it is possible to see students of which gender got more credits and which tried more often per module. The numbers used for the charts are not absolute numbers, as in the previous example, but averages. Looking at figure 4.5 it is evident that even though the last example has shown that there are fewer female students, on average they are more ambitious than their male colleagues and get more credits. The other chart shows that female university students try exams more often, which could either mean that they fail at first and then study harder, or that they are not satisfied with the grade and improve.

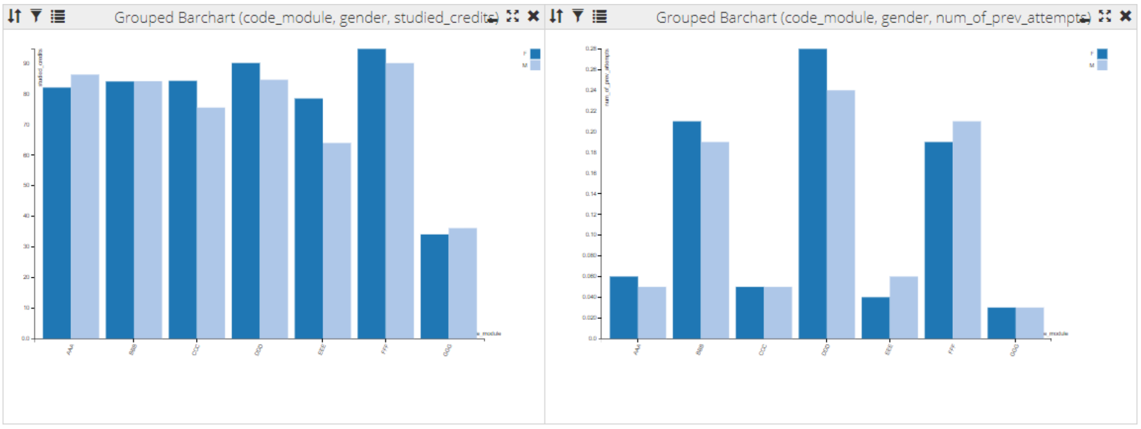


Figure 4.5: Comparison of effort for female and male students

Final result and studied credits per educational level

In the last example, two visualizations will be created, one pie chart and one grouped bar chart. The pie chart will display the final result of the students as a ratio of the entire number of the students, while the grouped bar chart will compare the final result with the number of studied credits and the educational level of the students. Figure 4.6 a) shows both charts one below each other. It is very noticeable that not that many students have passed the courses and that the pattern is very similar across all educational levels. In order to make it easier to compare the differences in studied credits per educational level of students who passed, it is possible to use the feature of the dashboard which links visualizations with one another. Thus, it is possible to simply selected “Pass” in the pie chart and only the passed bars will be highlighted in the grouped bar chart. Figure 4.6 b) shows this selection. It is interesting to note about this current view that the students who have a post graduate education have about the same number of studied credits as students who do not have a degree yet.

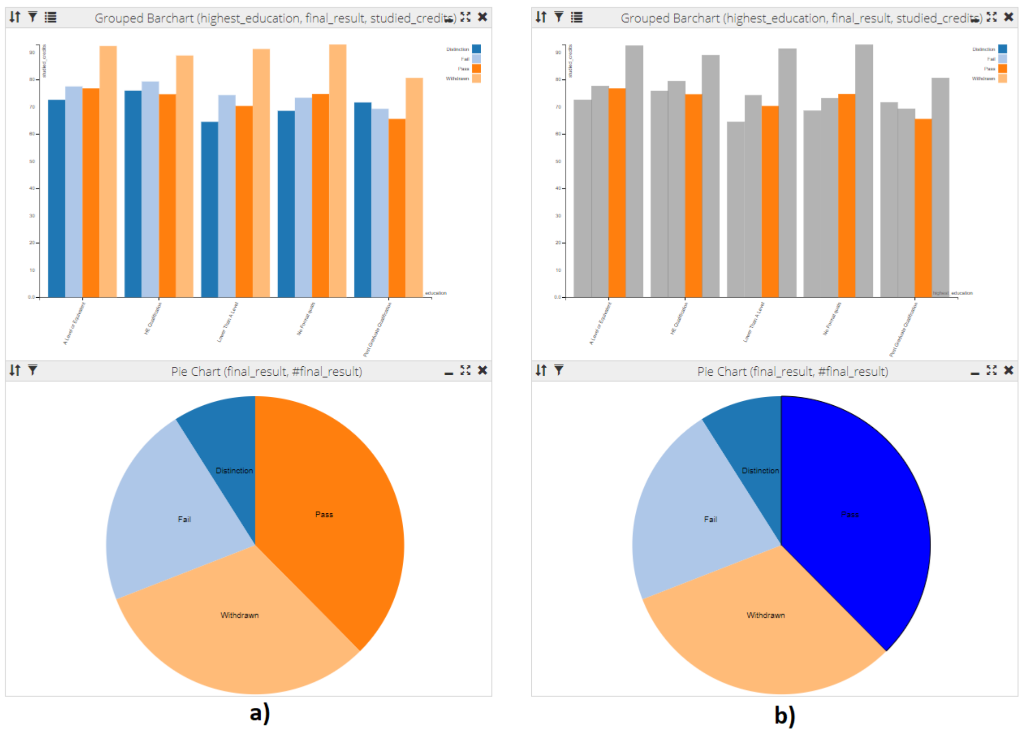

Grouped Barchart (highest_education, final_result, studied_credits)
Distinction
Fail
Pass
Withdrawn
Pie Chart (final_result, #final_result)
Distinction
Fail
Pass
Withdrawn
a)
b)

Figure 4.6: a) Final results compared to educational level and b) Student who passed highlighted

4.1.3 Sorting and Filtering

Interesting pieces of information are also the number of students per region, and which particular regions most students come from. For this purpose, a bar chart can be created by selecting the categorical field `region` and the computed numerical field `#region`. It seems that the bars which get displayed are in no particular order, neither alphabetically nor based on value. The data is simply displayed as it comes from the dataset. In order to change that, the sorting function can be used, and the bar chart is sorted based on value. The unsorted and sorted bar chart can be seen in figure 4.7.

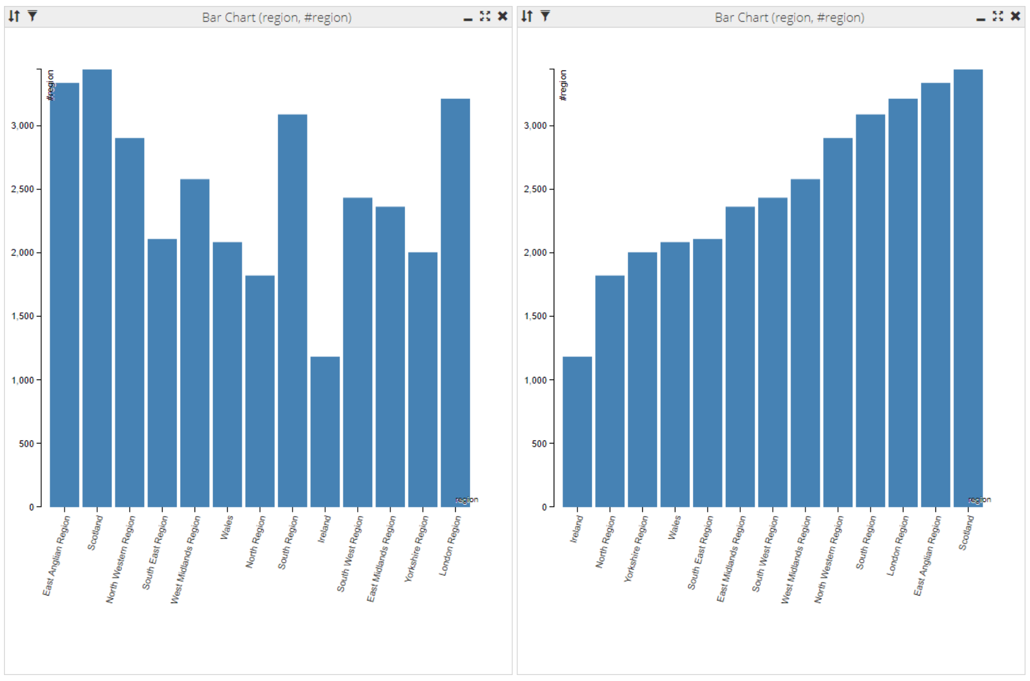


Figure 4.7: Bar chart as first created (left) and sorted bar chart (right)

With the help of the sorting functionality that is part of the visualization, it is now easily possible to determine the most frequent regions based on the number of students. Assuming that the regions with over 3000 students are relevant for examination, it is difficult to see if the “North Western Region” and “South Region” pass this mark or not. For a fine tuning of the visualization, the filtering option can be used. By simply setting the lower bound for “#region” to 3000, all the bars below that threshold are hidden. Figure 4.8 shows the filtering dialog and the result of the filtering. At this point it is evident when looking at the filtered bar chart that “North Western Region” has below 3000 students and that “South Region” has more.

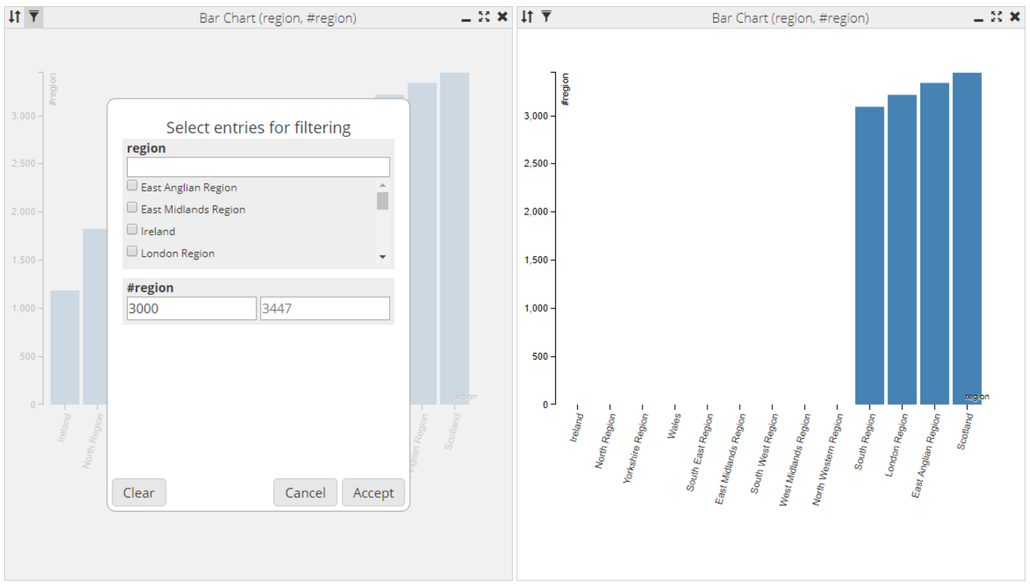


Figure 4.8: The filtering dialog of the bar chart (left) and the filtered bar chart (right)

4.1.4 Conclusion

This subsection has shown a procedure how to explore and prepare a dataset and how to explore the data with *Visualizer*. It is also possible to share the insights by creating a bookmark link and sending it to someone else, or to store the link as a bookmark in the browser for later exploring. There are, of course many more ways to engage in the exploration of data, and it is up to the user to find a way which best suits him.

4.2 Integration of *Visualizer* into an existing web page

The previous subsection has shown how a dataset can be manually explored, but which options are there if certain results should be presented to someone in an easily understandable way, without the need to specify how to arrive there? The solution for this lies in *Visualizer's* ability to store a configuration in a link which automatically generates the visualizations from the dataset. In combination with datasets which can be loaded remotely, the usage of a parameter to automatically disable the navigational UI elements and understanding the URL API of *Visualizer*, it is possible to integrate visualizations into web pages and display different datasets with the same schema automatically on page load, or even generate them with a button click. This is possible due to the fact that *Visualizer* is controllable to a great extent via the URL, and this section will show how it is possible to create navigational elements which allow content to be displayed automatically on a web page with the help of *Visualizer*. In addition to visualizing the data, interactive features such as brushing, filtering and sorting of the visualizations can also be used in the integrated visualizations.

4.2.1 Demo Page Overview

For the purpose of this demonstration a local demo page has been created, which can be seen in figure 4.9, and it consists of: (i) a control area (on the left side) with a select box for choosing a data source and multiple buttons to generate visualizations with, and (b) an iframe (the white area) which will be used to host *Visualizer*. The iframes source is set to "about:blank" when the page is loaded and is then later changed when a button is clicked.

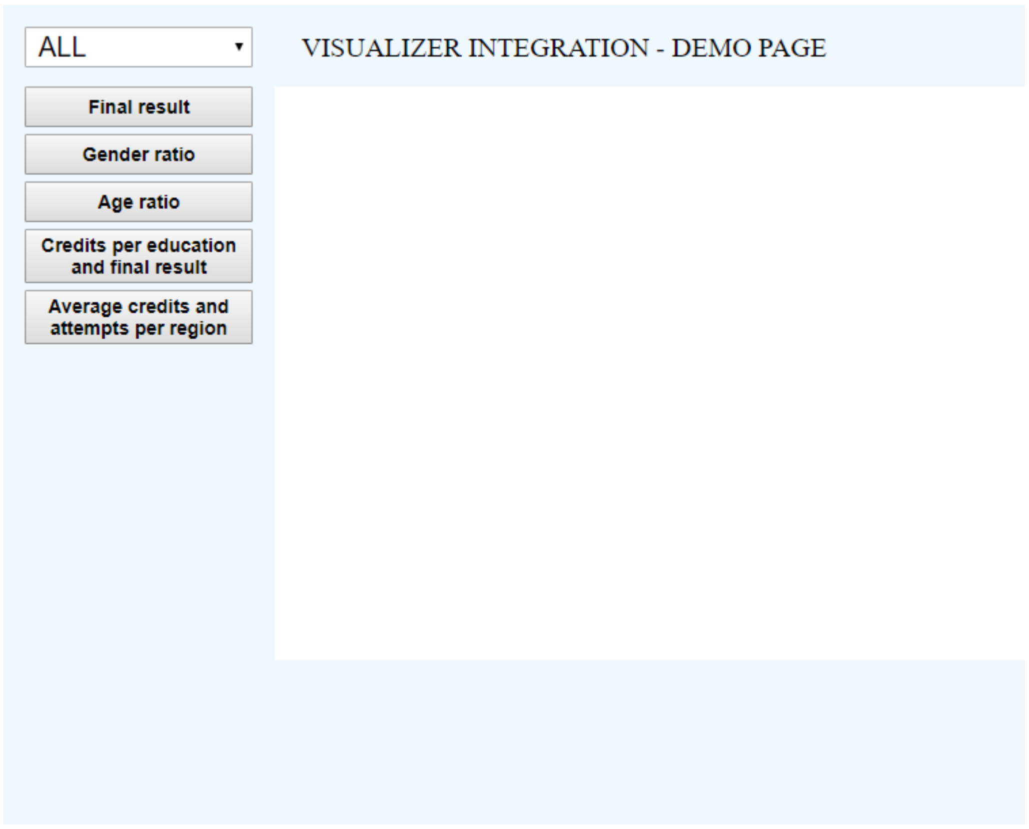

ALL
VISUALIZER INTEGRATION - DEMO PAGE
Final result
Gender ratio
Age ratio
Credits per education and final result
Average credits and attempts per region

Figure 4.9: *Visualizer* integration: Demo page

As a dataset, the studentInfo.csv of the Open University Learning Analytics dataset has been chosen again, but in contrast to the previous subsection, this dataset has been sliced by code module. Therefore, there will be six different student datasets, and additionally one for the whole class which will be selectable via the select box. This should demonstrate that the created configuration for *Visualizer* does not care about the contents of the file, but only the structure. When clicking the buttons they will generate different visualizations which are described on the buttons themselves. The visualizations are created by generating a URL and setting the `src` attribute of the iframe to this URL.

4.2.2 Generating a URL

The way the URL is generated is identical to how *Visualizer* generates the URL when bookmarking a configuration. The only difference is that the user has to generate the URL himself. In addition to the steps that *Visualizer* goes through when creating a URL, it is possible to add another parameter to the URL to jump directly to the preview view. This is done by appending "`&presentationMode=true`" to the generated URL. All in all, the structure of the URL with the used `presentationMode` parameter consists of four parts:

- `host`: the URL where *Visualizer* is hosted
- `dataurl`: the URL which points to a .csv file
- `config`: a stringified json object which contains all the information about the displayed visualizations, their size and position
- `presentationMode`: if set to true, it displays only the visualization windows

Therefore, a URL with all parameters set would have the following structure:

```
host + "?dataurl=" + urltocsv + "&config=" + conf + "&presentationMode=true"
```

The host parameter will be set to the same value in all of the following examples, namely:

```
https://vizrectest.know-center.tugraz.at
```

The parameter `presentationMode` will also be always set to true.
`urltocsv` will change depending on the value selected in the select box and point to the dataset of the selected code module. All the datasets are hosted on the same server as *Visualizer*.
The value of the `config` parameter is generated with the following steps:

1. Create an array of action objects in the order in which the actions should be applied (`actionStack`)
2. Convert the array into a string with the `JSON.stringify()` method
3. Encode the string with the `encodeURIComponent()` method

The mechanics of the actionStack have already been dealt with in section 3.3.3 and section 3.4.1 when talking about the "Get Bookmark Link" feature. Additionally, all the possible action objects have a detailed description in Appendix A, therefore the complete explanation will be omitted here, and only the used `actionObjects` will be explained.

4.2.3 Generating Visualizations

Each of the buttons on the demo page have similar functionalities. They generate different `actionStacks` and then set the URL of the iframe accordingly. The `actionObjects` used in the demo page will be used for the generation of different visualizations and the `actionObject` for setting a brush.

Single Chart Creation

The four buttons in the demo page generate one visualization which fits the whole iframe. How this is achieved will be shown by explaining the code of the function for the "Final result" button and then pointing out the differences of the others.
Clicking the button calls the onclick handler of this button which looks as follows:

```javascript
function showBarchart(){
    var actionStack = [];
    actionStack.push({
        'action' : 'visualization',
        'visualization' : "barchart",
        'labels' : "final_result|#final_result",
        'aggregations': ["","count"],
        'visTitle' : 'Final Result',
        'sortBy': {
            'ascending' : true,
            'label': 'final_result'
        }
    });

    var csvURL = $("#dataSource").val();
    var config = encodeURIComponent(JSON.stringify(actionStack));
    var iframeURL = visualizerHost;
    iframeURL += "?dataurl=" + csvURL;
    iframeURL += "&config=" + config;
    iframeURL += "&presentationMode=true";
    document.getElementById("visualizerFrame")
            .setAttribute("src", iframeURL);
}
```

This function generates an `actionObject` to create a bar chart with the categorical field “final_result” and the computed numerical field “#final_result”, which indicates the count of each entry in the field “final_result”. Additionally, a title is set for the visualization and the visualization is sorted by name. The sorting method is explicitly set to guarantee the order of labels, independent of which dataset has been loaded. This action object is pushed unto a `actionStack`. Following that, the actual dataset is determined, by getting the value set in the select box. After that, the URL for the iframe is generated by concatenating the `host` URL of *Visualizer*, the `dataurl` of the dataset, the `actionStack` and the `presentationMode`. Setting the URL to the iframes source will generate a bar chart. Similarly to how the bar chart was generated, the pie charts for “Gender ratio”, “Age ratio” and “Credits per education and final result” are generated as well.
Gender ratio sets the visualization attribute of the `actionObject` to “piechart” and takes “gender” and “#gender” as fields.
Age ratio uses the same visualization as Gender ratio, but uses “age_band” and “#age_band” as fields.
Credits per education and final result are slightly different, because they create a “groupedbarchart” with three fields, namely: “highest_education”, “final_result” and “studied_credits”. Additionally, the “studied_credits” is aggregated by calculating the average.

All four generated visualizations for different courses can be seen in figure 4.10.

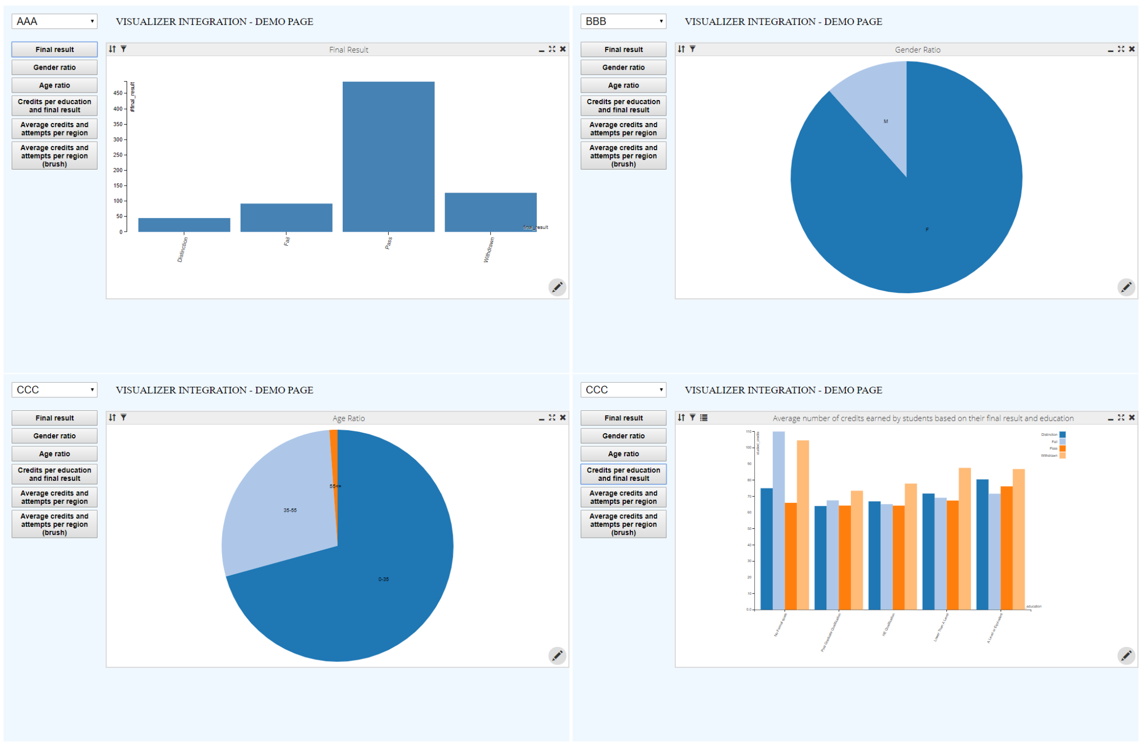

AAA
VISUALIZER INTEGRATION - DEMO PAGE
Final result
Gender ratio
Age ratio
Credits per education and final result
Average credits and attempts per region
Average credits and attempts per region (brush)
Final Result
Distinction
Fail
Pass
Withdrawn
BBB
Gender Ratio
M
F
CCC
Age Ratio
35-55
0-35
55<=
Average number of credits earned by students based on their final result and education

Figure 4.10: Generating Visualizations: (upper left) bar chart, counting final result occurrences in course AAA (upper right), pie chart with gender ratio in course BBB (lower left), age ratio in course CCC (lower right), and credits per education and final result also in course CCC

Single Chart Creation

The two buttons at the bottom create two bar charts in one iframe. While it is not necessary to consider positioning when creating one visualization, this changes when creating multiple charts. The default position of every visualization is the top left corner of the iframe and the default width and height of the visualization are set to 100%. If more than one visualization was created with this configuration, they would overlap each other and only the last one would be seen. Therefore, position and size have to be considered as well when creating multiple charts with *Visualizer*.
The button with the label “Average credits and attempts per region” creates two bar charts: The first bar chart uses the fields “region” and the average of “studied_credits”. This bar chart has a width set to 50% and height to 100%. The position is set to top: 0 and left: 0, i.e. it will be positioned in the upper

left corner, fill the whole vertical space and half of the horizontal space of the iframe. The second bar chart uses the fields "region" and the average of "num_of_prev_attempts". Width, height and top are the same as in the other bar chart, but left is set to 50, i.e. it should start in the middle and take up the other half of the horizontal space. Both visualizations are sorted ascending by value. Thus, it can be easily seen, from which region in average the students have studied the most, and from which region the students have in average the least number of attempts. Figure 4.11 shows the generated bar charts for the course BBB.

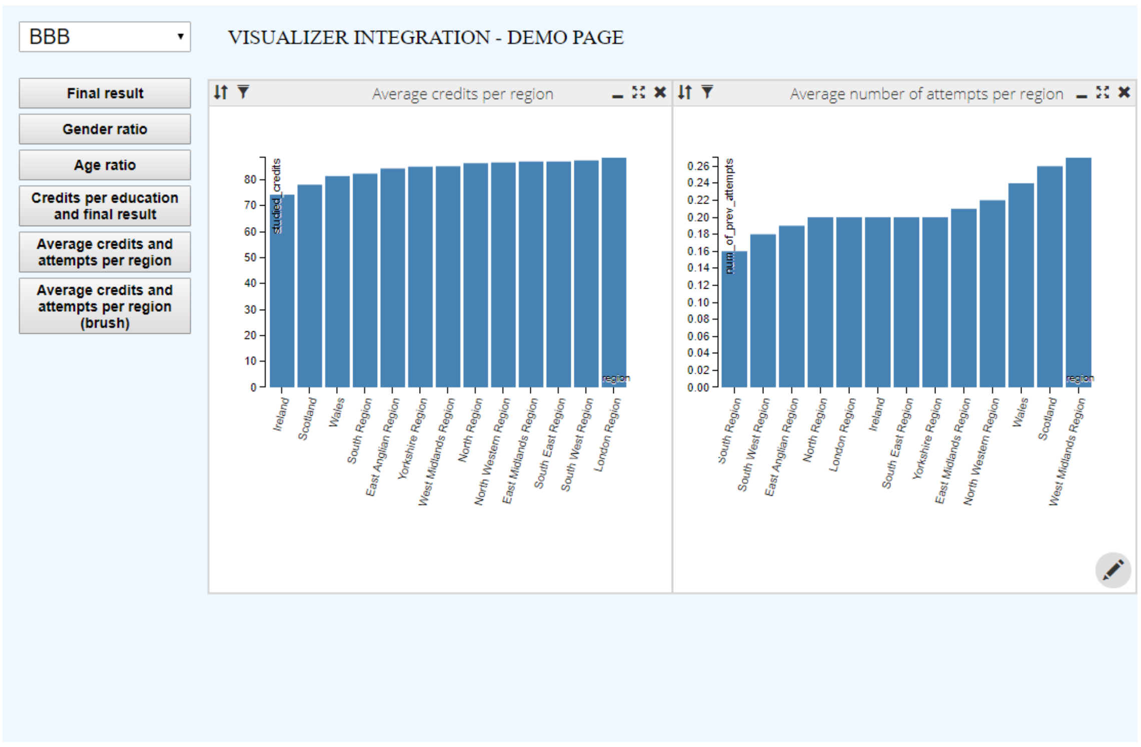


Figure 4.11: Multiple visualizations generated with one click - (left) Average number of studied credits per region, and (right) Average number of attempts per region

The two actions pushed to the `actionStack` in order to create those two bar charts look as follows:

```
{
    'action' : 'visualization',
    'visualization' : "barchart",
    'labels' : "region|studied_credits",
    'aggregations': ["","avg"],
    'visTitle' : 'Average credits per region',
    'left' : 0,
    'top' : 0,
    'width' : 50,
    'height' : 100,
    'sortBy': {
       'ascending' : true,
       'label': 'studied_credits'
    }
}

{
    'action' : 'visualization',
    'visualization' : "barchart",
    'labels' : "region|num_of_prev_attempts",
    'aggregations': ["","avg"],
    'visTitle' : 'Average number of attempts per region',
    'left' : 50,
    'top' : 0,
    'width' : 50,
    'height' : 100,
    'sortBy': {
       'ascending' : true,
       'label': 'num_of_prev_attempts'
    }
}
```

If multiple visualizations are created in the same iframe, they are automatically linked with each other. This means that highlighting bars in one bar chart will highlight the matching bars in the other bar chart. This is especially useful for comparing different values of the same category. It is also possible to activate a brush upon creation of the visualizations in order to highlight certain aspects for the user in advance. This can be done by pushing an `actionObject` with the `brush` action onto the actionStack. It is important to note that if a selection should be applied, this `actionObject` has to be pushed first. The button with the label “Average credits and attempts per region (brush)” utilizes this actionObject. Clicking on this button will generate the same bar charts as in figure 4.11, but will always highlight the regions “Scotland” and “Ireland” after being created, which can be seen in figure 4.12.

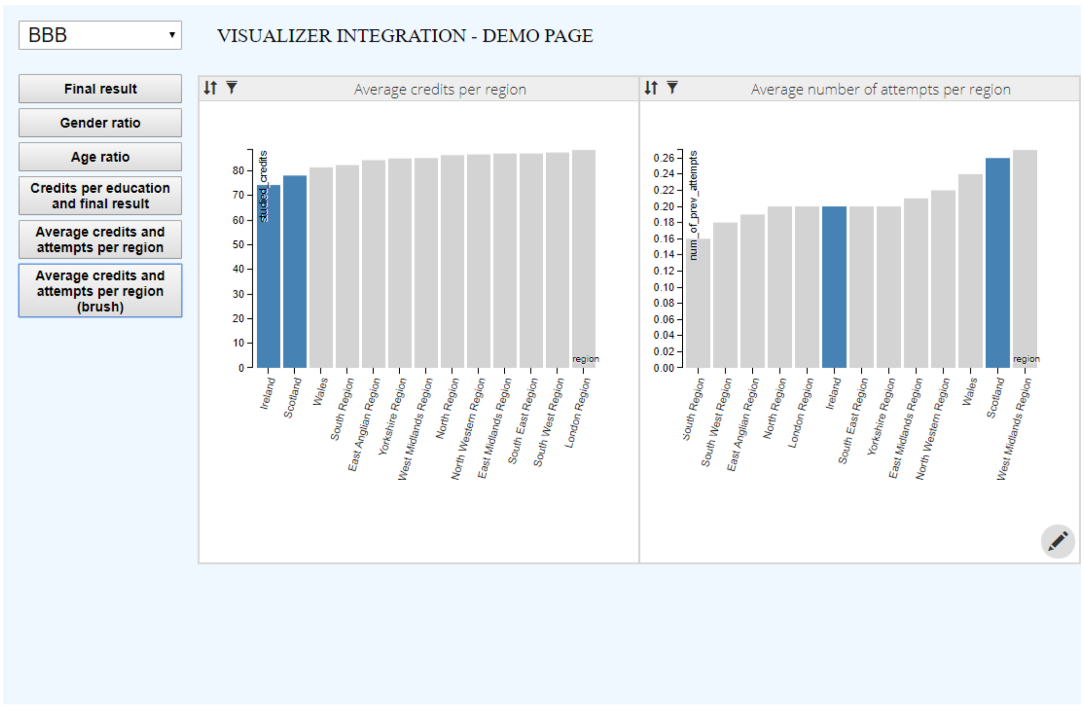


Figure 4.12: Multiple visualizations with activated brush

This can be achieved by pushing the following `actionObject` on the action-Stack, before pushing the objects for the visualizations.

```
{
    'action' : 'brush',
    'labelList' : ['region'],
    'selectedData' : [["Scotland"],["Ireland"]]
}
```

4.2.4 Usages

The main advantage of this is that web pages do not have to implement the data visualizations themselves. They can embed *Visualizer* in their web page and pass the data and the desired visualizations, without having to worry how to parse the CSV file, or how to do the actual drawing. The configuration features allow to display the same visualizations for different datasets. This could be utilized for example in a class, in which a CSV file showing his progress is generated for every user. Therefore, visualizations could be automatically generated for each student by only loading a different dataset. An example can be seen in figure 4.13 where the "final_result" is displayed for different courses.

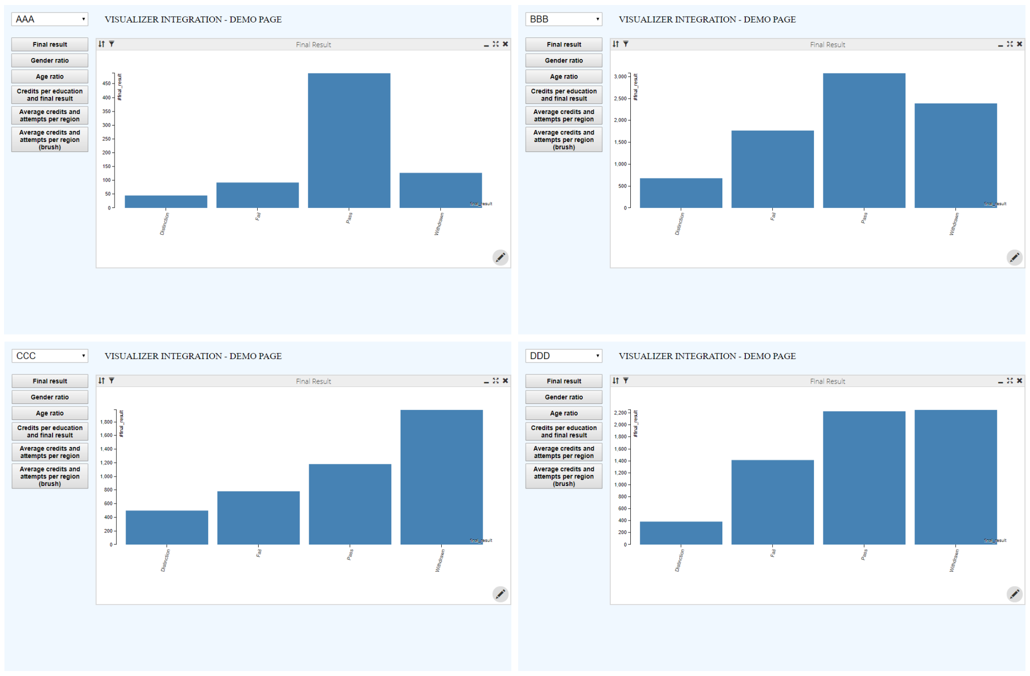

AAA
BBB
CCC
DDD
VISUALIZER INTEGRATION - DEMO PAGE
Final result
Gender ratio
Age ratio
Credits per education and final result
Average credits and attempts per region
Average credits and attempts per region (brush)
Final Result
final_result
Distinction
Fail
Pass
Withdrawn

Figure 4.13: Final result configuration applied to different courses: (upper left) Course AAA, (upper right) Course BBB, (lower left) Course CCC, and (lower right) Course DDD

4.3 Extension of Visualizations

The third major application of *Visualizer* is the extension of *Visualizer* itself by writing and integrating one's own visualizations. This is done by implementing *Visualizer's* chart API and uploading the visualizations to *Visualizer* as a logged in user. Before starting to explain the process of writing and uploading one's own visualizations, the API will be explained. The responsibilities in *Visualizer* are clearly divided; the framework handles all the data manipulations and passes the prepared data to the visualizations so that the visualizations themselves are only responsible for drawing the data in a responsive manner. Responsive in this case means that the visualization should adapt to the size changes of the container into which it gets drawn. Therefore, the API is limited to passing data to the visualizations and calling the drawing function. The drawing functionality is the minimum functionality which has to be implemented for the new visualization to work with *Visualizer*. There are additional functionalities (filtering, brushing, downloading) which do not have to implemented, but which break proper integration of the visualizations if missing.

4.3.1 Chart API

The whole chart API with the description of every variable and function can be seen in Appendix C. The focus in this subsection will be on the four functions which are meant to be written by the creator of a new visualization, and which are implemented by every visualization, namely:

- `drawVisualization`
- `applyFilter`
- `applyCssToSvg`
- `revertCssFromSvg`

It is suggested that the developer overwrites only those four functions. The other functions guarantee the proper handling of the calls from the framework. The other functions can be also overwritten for custom functionalities, but this is not recommended.

drawVisualization

The `drawVisualization` function is the most important function of the whole visualization. It is called every time the visualization should be redrawn. This happens when the visualization window is resized, a channel remap occurs or the data rows are sorted. The function receives three parameters: **`datarows`** `[array]` – This is a two-dimensional array. Each entry in this array holds one row from the selected fields. Therefore, this parameter contains the raw information which should be drawn. An example of the contents for a `datarows` array of a bar chart would be:

```
[
    ["audi", "10"],
    ["bmw","15"],
    ["ford","7"],
    ["vw","12"]
]
```

`channelMappings` `[array]` – This is an array of objects. Each object describes a mapping from a selected field to a visual channel of the visualization. Based on this parameter, the visualization can determine which index of each of the rows goes unto which visual channel. In addition to the mapping, this object also holds the type of the selected field and the aggregation if it is a numerical field. An example of a `channelMappings` array based on the `datarows` shown previously would be:

```
[
    {
       "channel" : "x-Axis",
       "label" : "manufacturer",
       "datatype" : "string",
       "aggregation" : ""
    },
    {
       "channel" : "y-Axis",
       "label" : "soldUnits",
       "datatype" : "integer",
       "aggregation" : "sum"
    }
]
```

`visIndex` `[integer]` - This is the index of the created visualization, which is also stored in the global variable `gnChartRowIndex`. It is only used for initializing the `brushingObserver`, which is described in Appendix C.

applyFilter

The `applyFilter` function should change the visualization so that only the data rows passed to the function stay displayed, while the others are hidden. It is important to emphasize that the visualization should be filtered and not only redrawn with the limited dataset. The difference is that the whole view stays the same and that the points of interest are accentuated. This can be seen in figure 4.14. The `applyFilter` function gets only one parameter, and this parameter is called `filteredDatarows`, which is a subset of the data rows used in the visualization.

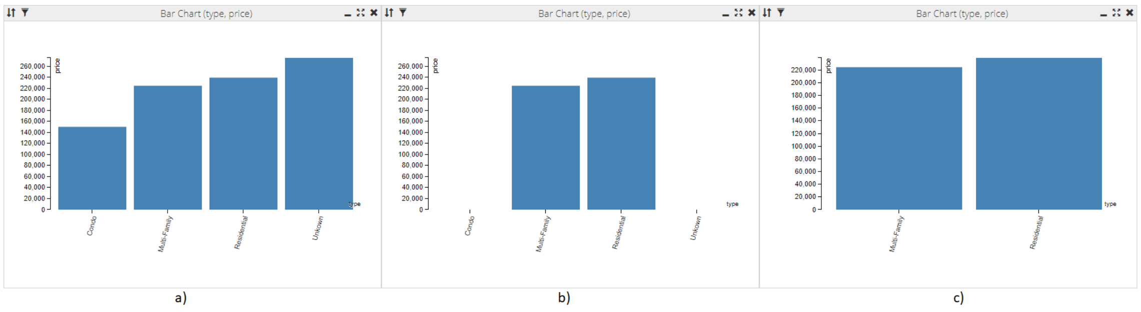


Figure 4.14: a) not filtered bar chart, b) correctly filtered bar chart c) wrongly filtered bar chart

applyCssToSvg

The `applyCssToSvg` function does not receive any parameters and is more of a helper function. This function is called shortly before the visualization download is started. It should be used to temporarily write all of the styles applied to the SVGs from the CSS directly inline. This step is necessary, because when rendering the SVGs into a canvas element, the styles are only applied, if they are inline. If the download of the visualization should be prohibited, because it results in broken or faulty images, the function can return false and an alert will automatically pop up to notify the users that this visualization does not offer a download option.

revertCssToSvg

`revertCssToSvg` is also a helper function like `applyCssToSvg` and does not receive any parameters either. This function is called after the download of the image of the visualization has finished and can be used to remove the redundant inline styling from the SVGs.

4.3.2 Project Structure

Knowing the chart API it is possible to start writing one's own visualizations, but before starting to actually write anything, the project structure has to be set up. When starting a new project, the minimum required file structure would look as follows:

```
libs/
css/
icon.png
main.js
```

The *libs* folder should hold all the libraries and additional JavaScript files that the visualization needs. All the files will be automatically loaded in *Visualizer* when the visualization gets created. Similarly to the *libs* folder, the *css* folder should contain all the stylesheets needed for the visualization, which will also load automatically. Both the *libs* and the *css* folder do not support hierarchies, therefore all of the .js and .css files should be put directly into those folders.
As can be seen, there are no html files. It is expected of the developer of the new visualization to draw the visualizations via JavaScript directly to the html body. This is possible due to the fact that every visualization exists in its own iframe and is isolated from the rest of the page. Therefore, the developer of the visualization can use libraries and stylesheets as needed, without having

to worry to ruin anything in the visualization framework. The *main.js* file should implement the chart API and will be called when the visualization gets created. This file does not have to be named main.js, but can have an arbitrary name, which will be specified when uploading the visualization.
The last file which has to be added is the *icon.png*, which should be an icon for the visualization that will be used in the *VisPicker*. Ideally, it should have a resolution of 256 x 256px in order to be sharp enough, without being too big. The *main.js* file is the only file in the shown project structure which can be renamed.

4.3.3 Getting Started

After getting familiar with the chart API and setting up the project structure for the new visualization, it is possible to start implementing a new visualization. The next immediate question which arises is, where to start. A good starting point is bl.ocks.org[3] which offers a rich variety of visualizations done with D3 under a GPLv3[4] or MIT[5] License. Those visualizations can be taken as a foundation, but still have to be adapted to work with *Visualizer*. For demonstration purposes, a sankey diagram[6] from the blocks page will be integrated into *Visualizer*.
After having chosen a visualization to start with, the next step would be to download all of the necessary libraries and files needed for this visualization. Inspecting the isolated demo page[7] with the developer tools (figure 4.15), it is possible to see that the files *d3.v3.min.js* and *sankey.js* are needed to run this visualization. Therefore, those files have to be downloaded and put into the *libs* folder of the project.

[3]`https://bl.ocks.org/mbostock`
[4]`https://opensource.org/licenses/GPL-3.0`
[5]`https://opensource.org/licenses/MIT`
[6]`http://bl.ocks.org/d3noob/c9b90689c1438f57d649`
[7]`http://bl.ocks.org/d3noob/raw/c9b90689c1438f57d649/`

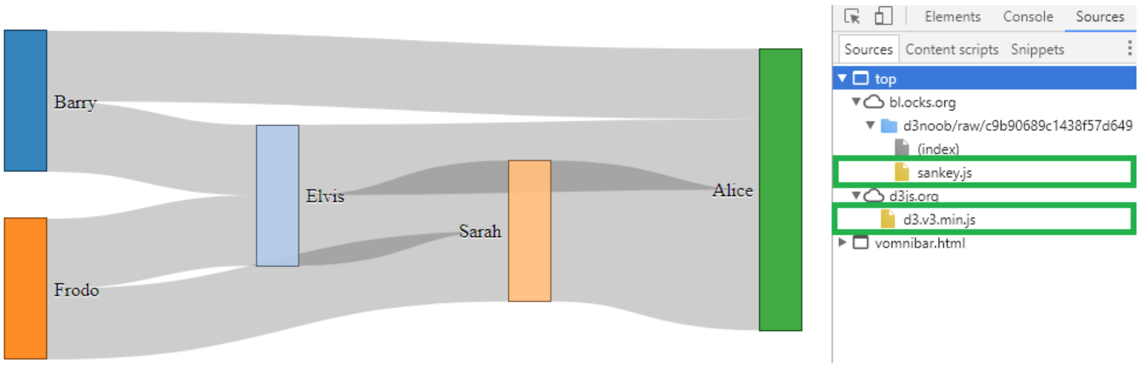


Figure 4.15: Sankey diagram: Needed libraries highlighted

In this demo of the sankey diagram, the css and main script are embedded directly into the index.html, which can be seen in Appendix D. For the purposes of *Visualizer*, this will have to be split up. Consequently, the contents of the style tag in the *index.html* will be put in a newly created *sankeyStyle.css* and the contents of the script tag in the *main.js* file. Before pasting the code from the demo page in the *main.js* file, the *main.js* file has to be set up. The basic structure of the *main.js* file would look as follows:

```
applyCssToSvg = function () {
    return false;
}

revertCssFromSvg = function () {
}

applyFilter = function (filteredDatarows) {
}

var drawVisualization = function (datarows, channelMappings, visIndex){
}
```

Those are the four functions which are recommended to be overwritten so that the visualization works correctly. In the first step, the code from the script tag of the *index.html* of the sankey demo will go into the `drawVisualization` function. After the contents of the script tag have been pasted to the `drawVisualization` function, the code of the sankey diagram has to be adapted.
Going through the code top-down, it can be immediately noticed that the width and height of the visualization are hard-coded to 700 and 300 pixels respectively, minus the margins. This has to be adapted to the responsive design of the visualization window so that the width and height of the container

are taken. The *sankey.js* file expects the width, height and margin variables to be global, therefore the `var` will be omitted. The code for the width and height will be changed from:

```
var margin = {top: 10, right: 10, bottom: 10, left: 10},
    width = 700 - margin.left - margin.right,
    height = 300 - margin.top - margin.bottom;
```

to:

```
margin = {top: 10, right: 10, bottom: 10, left: 10};
width = window.innerWidth - margin.left - margin.right;
height = window.innerHeight - margin.top - margin.bottom;
```

Furthermore, the selector, where the visualization should be appended, should be changed from `"#chart"` to `"body"`. Now, the data has to be adapted to function properly with the visualization. In its current form, the Sankey diagram loads the data directly from a CSV file, so the wrapper will have to be removed, and the `datarows` parameter passed to the `drawVisualization` function mapped correctly in the data variable. In order to do that, it is first necessary to understand how the Sankey diagram expects the data, and then the `datarows` have to be mapped the same way. Looking at the *sankey.csv*, which is also in Appendix D, there are three columns in the CSV file: source, target and value. Or differently put: There are two categorical columns, followed by a numerical. This will be something which will have to be considered later, to configure the visualization upload properly. The *d3.csv* function parses a CSV and creates an array of objects which have the attributes of the column names, and the values of the index of the row from the original CSV file. Therefore, the passed `datarows` will have to be transformed from a two-dimensional array to an array of objects with the attributes: source, target and value. This can be done with a simple .map call on the `datarows` array, which looks as follows:

```
data = datarows.map((row) => {
    let elem = {};
    elem[channelMappings[0].channel] = row[0];
    elem[channelMappings[1].channel] = row[1];
    elem[channelMappings[2].channel] = row[2];
    return elem;
});
```

The assumption is that `channelMappings` already describes the channels of the visualization. Therefore, in the `channelMappings` object at index 0, the attribute `channel` would equal to source, at index 1 to target and at index 2 to value. With this, the new visualization can already be used in *Visualizer*.

After adding an icon, the project structure should now look like this:

```
libs/
    d3.v3.min.js
    sankey.js
css/
    sankeyStyle.css
icon.png
main.js
```

The project structure can now be zipped and uploaded to *Visualizer*.

4.3.4 Uploading the Visualization

In order to upload the visualization, a new user profile has to be created in *Visualizer*. This is done by loading any dataset, going to the *Visualization Dashboard* and clicking on “Login”, then “Register”. A new user with the name “visUser” will be created for demonstration purposes. After the user has been created and the user has logged in, the visualization can be uploaded. This can be done by clicking on the “Upload Visualization” button. After clicking on the button, the visualization upload dialog pops up. The fields of the upload visualization dialog will be filled as follows:

- *Visualization name*: Sankey Diagram
- *Main filename*: main.js
- *Data aggregation mandatory*: True
- *Skip aggregation*: False
- *Max datapoints without warning*: 20

Three channels will be created: source, target and value, with the following description:

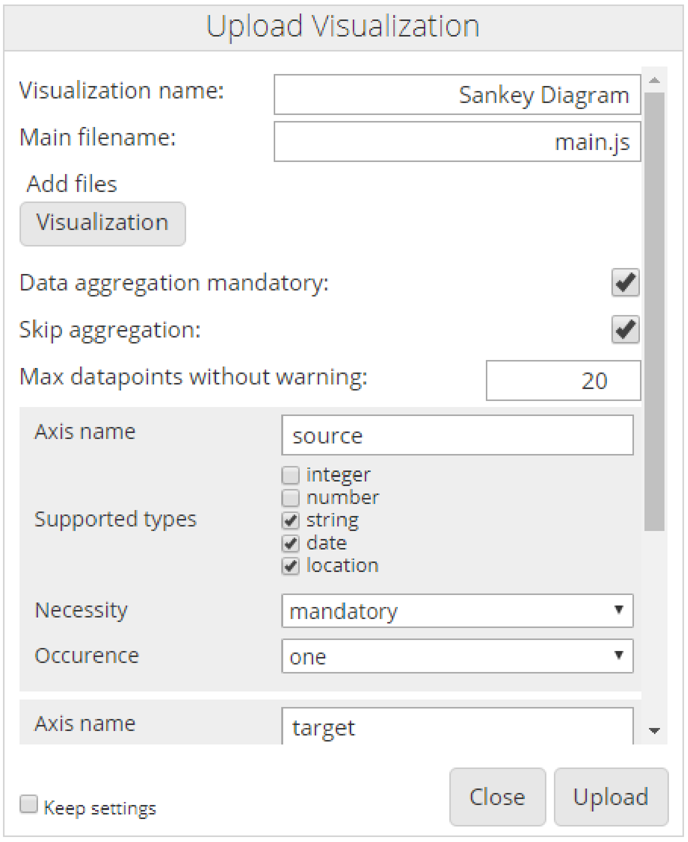
- *Axis name*: source
- *Supported types*: string, data, location
- *Necessity*: mandatory
- *Occurence*: one

- *Axis name*: target
- *Supported types*: string, data, location
- *Necessity*: mandatory

- *Occurence*: one

- *Axis name*: value
- *Supported types*: integer, number
- *Necessity*: mandatory
- *Occurence*: one

After filling everything out, the Upload Visualization dialog should look like in figure 4.16.


Upload Visualization

Visualization name: Sankey Diagram
Main filename: main.js
Add files
Visualization
Data aggregation mandatory: ✓
Skip aggregation: ✓
Max datapoints without warning: 20

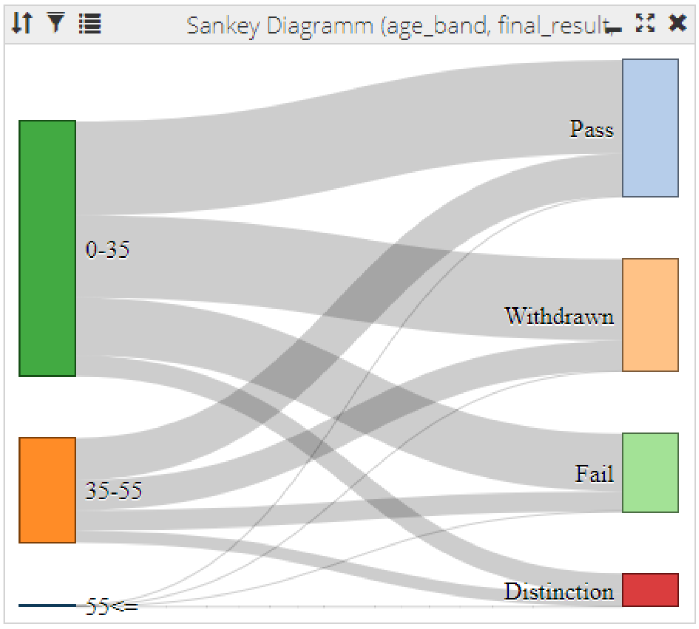
Axis name: source
Supported types: integer [ ] number [ ] string [✓] date [✓] location [✓]
Necessity: mandatory
Occurence: one

Axis name: target

Keep settings [ ]
Close
Upload

Figure 4.16: Visualization Upload dialog: Description of Sankey Diagram

Clicking on the "Upload" button will upload the visualization to *Visualizer* and update the *VisPicker*. The sample visualization and the *VisPicker* with the Sankey diagram and user overlay icon indicating that it is a user visualization, can be seen in figure 4.17



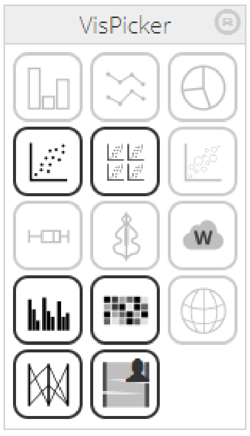


Figure 4.17: Sankey diagram (left) and *VisPicker* with user visualization (right)

4.3.5 Extended functionalities

Currently, the sankey diagram was only adapted to display the data in *Visualizer* correctly, but it does not fully support all of the functionalities yet. In this subsection the `applyFilter` and the `brushingObserver` functionalities will be added. Additionally, for simplicity reasons, the JavaScript library jQuery was added to the visualization in order to simplify the process of manipulating DOM elements.

Store Information in the Visualization

Before starting to add those functionalities, it is needed for convenience to store the information about every data row to the matching structures in the visualization, so that they can be found more easily when filtering and brushing. In case of the sankey diagram, the structures of relevance are the links between the blocks. Three attributes will be added to each link, namely: `data-source`, `data-target` and `data-value`, respectively for the information of each channel per row.
The creation of the links will be changed from:

```
var link = svg.append("g").selectAll(".link")
              .data(graph.links)
              .enter().append("path")
              .attr("class", "link")
              .attr("d", path)
              .style("stroke-width", function(d) {
                  return Math.max(1, d.dy);
              })
              .sort(function(a, b) {
                  return b.dy - a.dy;
              });
```

to:

```
var link = svg.append("g").selectAll(".link")
              .data(graph.links)
              .enter().append("path")
              .attr("class", "link")
              .attr("d", path)
              .style("stroke-width", function(d) {
                  return Math.max(1, d.dy);
              })
              .attr("data-source", function(d){
                return d.source.name;
              })
              .attr("data-target", function(d){
                return d.target.name;
              })
              .attr("data-value", function(d){
                return d.value;
              })
              .sort(function(a, b) {
                  return b.dy - a.dy;
              });
```

With this information directly attached to the links, it is much easier to find the links bound to the rows, because each link represents exactly one row in the data rows.

Implementing applyFilter

It is important to note that the filtering is completely done by the framework. The visualization only has to adapt the visualization to the filtered rows which are passed as a parameter to the `applyFilter` function. The procedure of applying the filter is then simple. First, all the links have to be hidden. After that, the passed `filteredDatarows` parameter is iterated through and then each link matching a row is shown again. The code for the complete `applyFilter` function looks as follows:

```
applyFilter = function(filteredDatarows) {
  $("path.link").hide();

  if(filteredDatarows.length == 0) {
    $("path.link").show();
    return;
  }

  filteredDatarows.forEach(function(row){
    $(‘path
       .link
       [data-source="${row[0]}"]
       [data-target="${row[1]}"]
     ‘).show();
  });
}
```

Implementing the brushingObserver

In order to implement the `brushingObserver` it is first needed to register the visualization on it. This is done by calling the `registerListener` function of the global `brushingObserver` like this:

```
brushingObserver.registerListener((newVisIndex) =>
    {visIndex = newVisIndex;},
    visIndex,
    brush);
```

The first parameter is a callback function to update the visualization index of the visualization by the `brushingObserver`, the second is the index of the current visualization, and the third is a callback function which gets called when an update in another visualization happens. The full description of the `brushingObserver` can be found in Appendix C.
Having registered the visualization on the `brushingObserver` it is needed to implement the passed callback function brush. The callback function has two parameters. The first parameter are the data rows selected by another visualization, and the second parameter are the matching labels (field names). A lot of error detection is needed in this function, because it is not certain if the selection from another visualization matches the selected fields in this visualization. Therefore, the first thing to do is to find the matching indexes in the `labelList` parameter for the source and target channel. Following that, arrays for the sources and targets get built, which then get iterated through and the matching links get highlighted, while the not matching ones get dimmed by increasing and decreasing the opacity of the links. The

implemented brush callback function looks like this:

```
function brush(selectedData, labelList) {
  let sourceIndex = -1;
  let targetIndex = -1;

  // gaChannelMappings[0] -> source
  // gaChannelMappings[1] -> target
  if(typeof(labelList) != "undefined") {
    sourceIndex = labelList.indexOf(gaChannelMappings[0].label);
    targetIndex = labelList.indexOf(gaChannelMappings[1].label);
  }

  if( (sourceIndex === -1 && targetIndex === -1)
      || selectedData.length === 0) {
    $("path.link").css("stroke-opacity","0.2");
    return;
  }

  $("path.link").removeClass("active");

  let sources = [];
  let targets = [];
  selectedData.forEach((row) => {
    if(sourceIndex !== -1)
      sources.push(row[sourceIndex]);
    if(targetIndex !== -1)
      targets.push(row[targetIndex]);
  });

  if(sourceIndex !== -1 && targetIndex !== -1) {
    $("path.link").each(function() {
      let source = $(this).attr("data-source");
      let target = $(this).attr("data-target");

      if(sources.indexOf(source) !== -1
        && targets.indexOf(target) !== -1)
        $(this).css("stroke-opacity","0.2");
      else
        $(this).css("stroke-opacity","0.05");
    });
  }
  else if(sourceIndex !== -1) {
    $("path.link").each(function() {
      let source = $(this).attr("data-source");
      if(sources.indexOf(source) !== -1)
        $(this).css("stroke-opacity","0.2");
      else
        $(this).css("stroke-opacity","0.05");
    });
  }
  else if(targetIndex !== -1) {
    $("path.link").each(function() {
      let target = $(this).attr("data-target");
      if(targets.indexOf(target) !== -1)
        $(this).css("stroke-opacity","0.2");
      else
        $(this).css("stroke-opacity","0.05");
    });
  }
}
```

With this, the implemented sankey diagram can receive updates from other visualizations. The only thing left to do at this point is to add the functionality to select links in the visualization and to trigger updates in other visualizations. In order do that, each link will get an onclick function. This onclick function has three very simple tasks. The first is to toggle the “active” class on the link, which should highlight the link. In order to enable the highlighting, the CSS rule `.link.active {stroke-opacity: .5 !important;}` will be added to the *sankeyStyle.css*. The second task is to gather all the links which have the “active” class, and for each link a data row based on the `data-source` and `data-target` attributes have to be created. Only the categorical attributes are relevant for the `brushingObserver`. Finally, the third task is to call the update function of the `brushingObserver` with the selected rows, the index of the visualization and the `labelList`. The code for adding and implementing the click-handler looks as follows:

```
$("path.link").on("click", function(event){
    $(this).toggleClass("active");

    let labelList = [gaChannelMappings[0].label,
                     gaChannelMappings[1].label];

    let selectedData = [];
    $("path.link.active").each(function() {
         selectedData.push([$(this).attr("data-source"),
                            $(this).attr("data-target")]);
    });

    brushingObserver.update(visIndex,
                            selectedData,
                            null,
                            labelList);
});
```

Conclusion

After adding jQuery, implementing the `applyFilter` function and the `brushingObserver`, and updating the *sankeyStyle.css*, the visualization project is zipped and uploaded again to *Visualizer*. The old visualization should be removed beforehand to not clutter the *VisPicker*. The `applyCssToSvg` and `revertCssToSvg` have been omitted from the description of the upload of a new Visualization, because those functionalities have much less importance and do not break immersion when exploring data. All in all a good insight has been given into how simple it can be to integrate new visualizations in *Visualizer*.

4.4 Summary

In this chapter, the main applications of *Visualizer* were demonstrated by going through different scenarios step by step and showing how different tasks can be accomplished with relative ease. The next chapter will look at the performance of *Visualizer* and what its limitations are.

Chapter 5

Benchmarks

5.1 Introduction

In the previous chapters, the components of *Visualizer* and its main applications were explained in detail. This chapter will cover benchmarks for certain key computations in *Visualizer* based on differently sized datasets.
For the benchmarks, the dataset for the real estate transactions in the Sacramento area [1] was taken again, and the rows were duplicated to create differently sized datasets. At first, from the original file the number of rows were duplicated to create one million rows. Following that, those one million rows were copied and datasets were created which contained one to five million rows. This was done to ensure a linear increase in data between the differently sized datasets.
The following three benchmarks in the most critical parts of the application were performed:

1. *Loading the dataset* – This is the time which *Visualizer* takes to load the dataset from the file system into memory and display it in the *Dataset Table*. Additionally, the memory consumption for the browser tab was monitored.
2. *Aggregating data* – The time was measured which *Visualizer* takes to aggregate the selected fields prior to displaying the visualization.
3. *Applying configurations* – The time was measured to apply certain configurations to datasets

The machine on which the benchmarks were performed was a Lenovo Thinkpad t470p with the following specification:

[1] `http://samplecsvs.s3.amazonaws.com/Sacramentorealestatetransactions.csv`

CPU	Intel Core i7-7700HQ @2.8 GHZ
RAM	16GB DDR4
SSD	512GB
GPU	NVidia GeForce 940MX
Operating System	Windows 10 Pro 64-Bit

Table 5.1: The specification of the machine the benchmarks were run on

The web browser in which the benchmarks were performed was Google Chrome[2] with the version 63.0.3239.108.
In order to ensure that the benchmarks were carried out fairly, before starting each benchmark type, the machine on which the benchmarks were performed was restarted. After restarting the machine, only the web browser was started, and all nonessential services were disabled. Additionally, all plug-ins in the browser were disabled. Every single benchmark was run in a new incognito window of Chrome to ensure that Visualizer gets loaded every time with a fresh browser cache.

5.2 Loading the dataset

As was explained in the chapter 3, loading the dataset from the file system and displaying it in the browser requires multiple steps. First, the CSV file has to be parsed and the data structure containing the raw entries from the CSV file has to be created. Following that, certain computations have to be performed on the raw data, like determining the data type or counting the unique elements in a column. All of this can take a substantial amount of time and memory. Therefore, this benchmark should showcase the performance and limitations regarding the size of the datasets in *Visualizer*. The time measuring starts when the `Accept` button is clicked on the file selection page. The time stops when the complete *Dataset Table* is displayed. This is also the moment when the memory consumption is measured in the web browser's task manager. Figure 5.1 shows the memory consumption of *Visualizer* before a dataset is loaded.

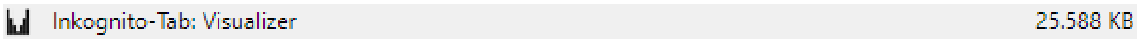

Inkognito-Tab: Visualizer 25.588 KB

Figure 5.1: Memory consumption of *Visualizer* before a dataset gets loaded

The benchmarks were performed five times for every dataset to eliminate possible spikes. The tables 5.2 to 5.6 display the results of the benchmarks.

[2] `https://www.google.com/chrome/browser/desktop/index.html`

	Time in ms	Memory in KB
1.	4474,29	933300
2.	4607,92	967292
3.	4606,51	966872
4.	4605,98	966360
5.	4551,89	933552
AVERAGE	**4569,318**	**953475,2**

Table 5.2: Time taken to load the dataset and memory consumption - 1000000 rows

	Time in ms	Memory in KB
1.	10734,76	1555088
2.	9568,18	1569364
3.	9538,96	1569556
4.	9648,87	1557092
5.	9555,59	1569260
AVERAGE	**9809,272**	**1564072**

Table 5.3: Time taken to load the dataset and memory consumption - 2000000 rows

	Time in ms	Memory in KB
1.	16497,33	2317968
2.	14511,47	2320516
3.	14644,07	2321967
4.	14537,47	2320072
5.	14603,3	2319924
AVERAGE	**14958,728**	**2320089,4**

Table 5.4: Time taken to load the dataset and memory consumption - 3000000 rows

	Time in ms	Memory in KB
1.	21960,27	2903260
2.	19903,38	2903332
3.	19725,38	2903428
4.	19754,14	2903172
5.	19383,16	2903652
AVERAGE	**20145,266**	**2903368,8**

Table 5.5: Time taken to load the dataset and memory consumption - 4000000 rows

	Time in ms	Memory in KB
1.	N/A	N/A
2.	N/A	N/A
3.	N/A	N/A
4.	N/A	N/A
5.	N/A	N/A
AVERAGE	**N/A**	**N/A**

Table 5.6: Time taken to load the dataset and memory consumption - 5000000 rows

Figure 5.2 shows the time and memory consumption as line charts. As can be seen in this figure, the time taken and the memory consumption grows linearly with the number of rows. It is noteworthy that the dataset with 5 000 000 rows could not be loaded. This is due to two reasons:

1. Google Chrome has a memory limitation per browser tab of 4GB. This is by design, as certain attacks can be performed by allocating more than 4GB of memory[3]
2. While loading the dataset, the memory consumption temporarily spikes over the recorded end memory consumption. Monitoring the memory spikes showed that the last spikes happened at around 3.7 GB of memory usage in the tab before the crash happened.

Therefore, the dataset with 5 000 000 rows will be omitted from now.

[3]`https://bugs.chromium.org/p/chromium/issues/detail?id=416284#c5`

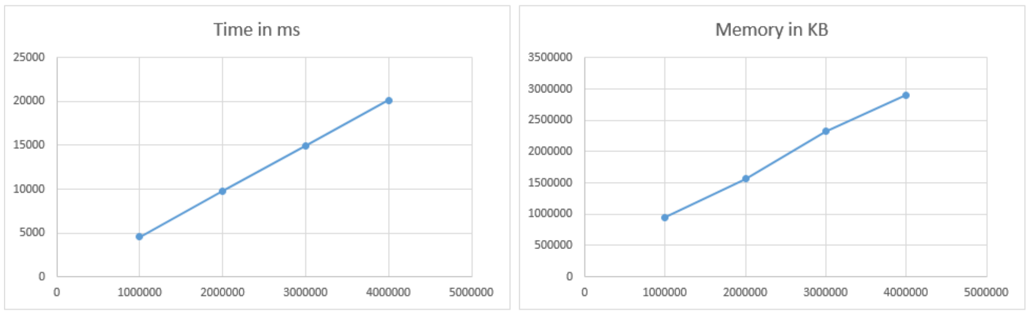


Figure 5.2: Line charts showing linear increase in time and memory for linear increase in the number of rows

This benchmark showed that while the time taken to load the dataset into *Visualizer* is actually pretty fast, the memory consumption can cause many troubles and a solution for bigger datasets should be considered.

5.3 Aggregating data

The process of visual exploration of data takes the creation of multiple visualizations, therefore, the time to aggregate the data of the selected fields has to be fast to ensure the user can progress with his exploration more swiftly. Each visualization takes a different amount of time to display the data, yet the aggregation of the data has to be performed in the same way for every visualization. Therefore, the second performed benchmark measures the time taken to aggregate a certain number of selected categorical and numerical fields. The time starts when the `Accept` button is clicked in the *Select aggregations dialog* and stops after the data is prepared for the visualization to be drawn. The benchmarks were performed for three cases:

1. **Case 1**: 1 categorical field and 1 numerical field were selected. This case can be used to create a bar chart, a pie chart or a box plot.
2. **Case 2**: 2 categorical fields and 1 numerical field. This case can be used to create a scatter plot, a heat map or grouped bar chart
3. **Case 3**: 5 categorical fields and 7 numerical fields. This case was used for stress testing. With this selection, a parallel coordinates and a scatter plot matrix visualization can be created.

The first two cases occur very often and the third case was used to put the implementation of the data aggregation under a stress test and to show its limits.

	Case 1	Case 2	Case 3
1.	239,49	600,96	2322,5
2.	228,89	618,79	2336,26
3.	228,31	592,94	2302,74
4.	233,95	637,84	2313,83
5.	233,58	592,27	2157,19
AVERAGE	**232,844**	**608,56**	**2286,504**

Table 5.7: Time taken in milliseconds to aggregate the data of the selected fields - 1000000 rows

	Case 1	Case 2	Case 3
1.	449,79	1216,73	4760,69
2.	455,77	1198,44	4796,02
3.	476,58	1206,22	4816,2
4.	460,88	1191,09	4797,93
5.	471,75	1211,59	4847,25
AVERAGE	**462,954**	**1204,814**	**4803,618**

Table 5.8: Time taken in milliseconds to aggregate the data of the selected fields - 2000000 rows

	Case 1	Case 2	Case 3
1.	723,69	1862,44	8217,75
2.	733,28	1843,34	8397,69
3.	690,07	1886,21	8300,05
4.	712,02	1822,65	8256,75
5.	684,12	1867,22	7966,12
AVERAGE	**708,636**	**1856,372**	**8227,672**

Table 5.9: Time taken in milliseconds to aggregate the data of the selected fields - 3000000 rows

	Case 1	Case 2	Case 3
1.	952,63	3355,58	N/A
2.	973,38	3409,91	N/A
3.	926,26	3411,45	N/A
4.	949,85	3422,04	N/A
5.	956,65	3405,23	N/A
AVERAGE	**951,754**	**3400,842**	**N/A**

Table 5.10: Time taken in milliseconds to aggregate the data of the selected fields - 4000000 rows

Figure 5.3 shows the increase in time for the different cases. For one selected categorical and numerical field the time increase between the datasets seems linear. Having two categorical fields and one numerical field, at first there is a linear increase and suddenly for the last dataset the time doubles. The reason for this is that the dataset with the 4 000 000 rows already uses a lot of memory, as could be seen in the previous benchmark. Selecting two categorical fields and one numerical field produces a temporary data structure which in addition to the original dataset uses almost all of the available memory. The assumption here is that the browser uses the garbage collection quite often to not run out of memory, which drastically increases the time.
As can be seen from the table and from the line chart, case 3 for 4 000 000 rows could not be performed. The reason was that *Visualizer* ran out of memory again. As already mentioned, holding the 4 000 000 rows of the dataset in memory already takes up a huge chunk of the maximum memory. Performing the aggregation with that many selected fields produced an additional temporary data structure which was big enough to exhaust the remaining accessible memory.

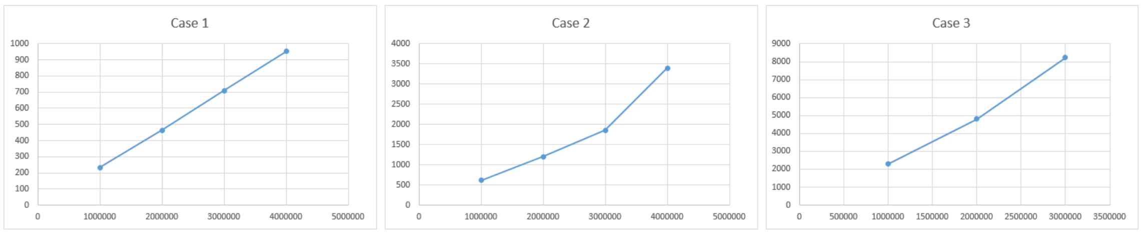


Figure 5.3: Line charts displaying the time taken to aggregate the selected fields for the three cases

In conclusion, the aggregation works very fast with a linear increase in time for a linear increase in the number of rows. The only concern again is that, when the memory of the browser tab is running out, the time taken to perform the aggregation suddenly increases or cannot even be performed.

5.4 Applying configurations

The third benchmark combines the first two benchmarks and performs additional manipulations on the dataset in the *Dataset Table*. At first, a number of actions were performed on the smallest dataset and following that a configuration link was created which applied all of those performed actions in succession. The same configuration link was then used on all of the datasets. The time measuring starts upon clicking on the `Accept` button in the file selection page and stops when all visualizations have been created. The following actions were performed in the configuration link after loading the dataset:

1. Filter the dataset by `type`, so that only the `Residential` type is displayed.
2. Filter the dataset by `price`, so that only real estates with a price below 300 000 are displayed.
3. Create a bar chart from the fields `city` and `price`. The average price is taken.
4. Create a heat map from the fields `type`, `sale_date` and `sq_ft`. The average for `sq_ft` is taken.

The tables 5.11 to 5.11 show the times it takes for Visualizer to apply all the actions in the configuration link.

	Time taken in ms
1.	6862,18
2.	6826,15
3.	6758,55
4.	6813,27
5.	6839,86
AVERAGE	**6820,002**

Table 5.11: Time taken in milliseconds to apply the configuration link - 1000000 rows

	Time taken in ms
1.	14411,14
2.	13700,03
3.	13718,97
4.	14067,53
5.	13690,16
AVERAGE	**13917,566**

Table 5.12: Time taken in milliseconds to apply the configuration link - 2000000 rows

	Time taken in ms
1.	23508,15
2.	23486,7
3.	23436,66
4.	23429,87
5.	23821,12
AVERAGE	**23536,5**

Table 5.13: Time taken in milliseconds to apply the configuration link - 3000000 rows

	Time taken in ms
1.	57451,79
2.	56959,46
3.	59059,97
4.	61160,65
5.	56626,31
AVERAGE	**58251,636**

Table 5.14: Time taken in milliseconds to apply the configuration link - 4000000 rows

Figure 5.4 shows the increase in time when applying the configuration link to the differently sized datasets. As is evident from the figure, for the three smaller datasets, the time taken to apply all of the actions increases linearly, as does the size of the dataset. But for the last dataset a big jump can be noticed in the time taken. This is again due to the fact that the memory of the browser tab is running out, and therefore the garbage collector is used extensively, which impacts the time it takes to perform the actions from the configuration link.

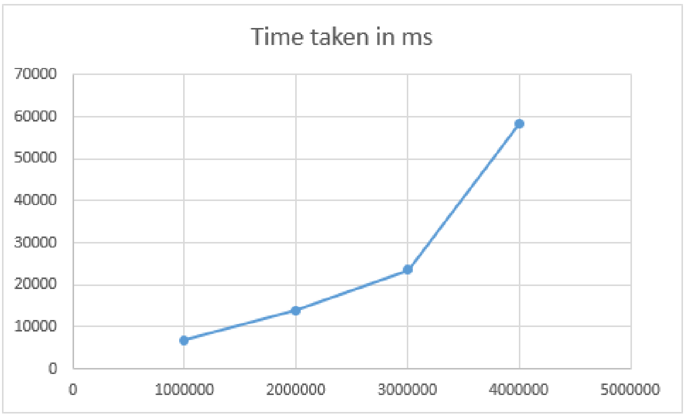


Figure 5.4: Line charts displaying time taken to apply the configuration link to different sized datasets

5.5 Summary

This chapter showed the results of the extensive benchmarks of *Visualizer*. As can be seen from the results, the computation scales very well with the size of the dataset, as long as the memory limit of the browser tab is not reached. On the other hand, a major problem is the memory consumption of the application, which will be difficult to solve, because the whole dataset has to be loaded into memory and the most efficient way to store the data is by storing the entry of each cell as a string in an array.
In order to make it possible to handle even bigger datasets, the memory consumption would be the main aspect which would have to be solved.
The next chapter will conclude this thesis and give an overview of the tasks which could be done in the future to further improve *Visualizer*.

Chapter 6

Conclusion

6.1 Summary

In this thesis *Visualizer*, a personalized and extensible web application for client side visual data analysis and data exploration, was presented and explained in detail. With its easy to use interface, fast data manipulation, user assistance, personalization and extensibility, a stable foundation was laid, upon which it is possible to expand the application in many different directions.
In order to ensure an easy way of platform independence, *Visualizer* was implemented as a web application. One of the main aspects of this was also to move as much as possible of the code and computations to the client side, to alleviate the server. This transfer to the client side had many advantages, mainly regarding the responsiveness of the page and computation speed. However, it reached limits as far as memory usage was concerned. Therefore, in this area, much more work could be done in order to optimize the memory handling and allow the user to load and work with even bigger files. Another aspect which leaves room for improvement concerns the drawing mechanisms. *Visualizer* used SVG to draw its visualizations. While SVG is considerably easy to use and offers the developer a simple way to generate appealing graphics, it has its limitations. Drawing many data points slows down the visualization and makes it difficult to move the visualization window. This is due to the fact that SVG writes its graphics elements directly to the DOM and therefore, when moving the visualization, the browser has to update the positions of many elements. Creating the visualizations with WebGL would increase the performance drastically.
In conclusion, within the scope of this thesis a web application was developed which is equally meant for entry level users and expert users. For entry level users the recommendation of compatible visualizations and a friendly, easy to understand interface was added, while the experts can manipulate bigger

datasets and extend the platform itself through the simple visualization API. Through the customization options, both groups can adapt the interface to their needs. The integrated personalized visualization recommender might be still in its infancy, yet when it is finished it will greatly improve the recommendation of suitable visualizations for the users.

6.2 Future Work

There is still a considerable amount of work that can be done on the platform to further improve the user experience and add many additional features. The most pressing tasks are the following:

- **Extensive user evaluation**: Various evaluations could be performed with this web application with distinct aims: (i) A usability evaluation of the platform for different user groups. For this evaluation beginners, semi-experts and experts would be needed (ii) Evaluation of the visualization API. For this many developers would be needed (iii) An evaluation of the configurability and integration of *Visualizer* in another platform. Performing all of those evaluations would give great insight in the areas which would need improvement.
- **Support for more data sources**: Currently, *Visualizer* supports only CSV, with the option to easily add JSON support as well. Nevertheless, it would be of utmost importance to support direct connections to databases, without the need of dumping the database tables into CSVs. In order to achieve this, first the JSON support have to be added and in the next step, connectors would have to be written on the server side which could connect to databases.
- **Loading multiple datasets at once**: Currently, it is possible to load and work with only one dataset at a time in *Visualizer*. In the future, it should be possible to load multiple datasets and have the option to seamlessly switch between the different tables in the *Visualization Dashboard* and even to merge datasets to create completely new ones.
- **Reimplement visualizations in WebGL**: This was already mentioned, but the visualizations written in SVG reach a limit concerning performance, when a big amount of data points have to be created. WebGL would improve the performance drastically.
- **Annotations**: Sometimes it is necessary to add annotations to the visualizations. This helps especially when continuing to work on a dashboard which was previously shared. Those annotations could be placed in the form of sticky notes directly on the dashboard, or in a comment sidebar.

- **Collaboration**: At the moment, it is possible to work on a dashboard only for one user at a time. The state of the dashboard can be saved and shared with a bookmark link, but when the other users open this bookmark, it will only create a new instance which looks the same as the one of the user that shared the dashboard. The instances would not be linked. Therefore, real time collaboration on the same dashboard could drastically reduce the time it takes to create *Visualization Dashboards*.
- **Visualization / theme store**: *Visualizer* already has user management, and the uploaded visualizations and dashboard customizations are already stored in a user profile. Therefore, it would be an improvement for the users to have a store in which they can select the visualizations and themes which they would like to have in *Visualizer* and add them to their dashboards by simply installing the new visualization or theme directly from the store.
- **Connection to cloud services**: Connectors to different cloud services, such as Google Drive[1], Dropbox[2], or Microsoft's OneDrive[3] could be added. With those connections it would be possible to directly access datasets from the users' personal cloud storage. Additionally, it would be possible to also store the user visualizations in a dedicated directory in the cloud service, which would give the user the possibility to add as many visualizations as he desires.
- **Periodic data update**: After adding connectors to different databases, it could be possible to refresh the visualizations periodically based on changes in the database. The refresh could be triggered, for instance, when the dataset gets new entries, or, another possible scenario could be a limitation of the dataset to the last 30 minutes of data. Therefore, every minute new entries would be added and the old entries would be dropped and the visualizations updated.
- **Reducing the memory consumption**: In order to be able to handle bigger datasets the memory consumption would have to be reduced as Chapter 5 has shown. This task is not that trivial, because of the browsers' memory limitations of 4GB per tab. To circumvent this limit, either IndexedDB[4] could be used, or the dataset could be split across multiple web workers[5].

[1] `https://www.google.com/drive/`
[2] `https://www.dropbox.com`
[3] `https://onedrive.live.com`
[4] `https://developer.mozilla.org/docs/IndexedDB`
[5] `https://developer.mozilla.org/de/docs/Web/API/Web_Workers_API`

List of Abbreviations

API	Application Programming Interface
CGI	Common Gateway Interface
CORS	Cross-Origin Resource Sharing
CSS	Cascading Style Sheets
CSV	Comma-separated values
DB	Database
DOM	Document Object Model
ECMA	European Computer Manufacturers Association
ES6	ECMAScript6
GPL	General Public License
GUI	Graphical User Interface
HSV	Hue Saturation Value
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LOD	Linked Object Data
MIT	Massachusetts Institute of Technology
RDF	Resource Description Framework
SVG	Scalable Vector Graphics

UI	User Interface
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Bibliography

Mohamed Aly, Anis Charfi, and Mira Mezini. On the extensibility requirements of business applications. In *Proceedings of the 2012 Workshop on Next Generation Modularity Approaches for Requirements and Architecture*, NEMARA ’12, pages 1–6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1127-4. doi: 10.1145/2162004.2162006. URL `http://doi.acm.org/10.1145/2162004.2162006`.

J. Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. ESRI Press, 2011. ISBN 9781589482616.

M. S. T. Carpendale. Considering visual variables as a basis for information visualisation. Technical report, University of Calgary, Calgary, AB, 2003. URL `http://pharos.cpsc.ucalgary.ca/Dienst/UI/2.0/Describe/ncstrl.ucalgary_cs/2001-693-16`.

Woei-Kae Chen and Kuo-Hua Chung. A table presentation system for database and web applications. In *e-Technology, e-Commerce and e-Service, 2004. EEE ’04. 2004 IEEE International Conference on*, pages 492–498, March 2004. doi: 10.1109/EEE.2004.1287352.

Eric D. Ragan, Alex Endert, Jibonananda Sanyal, and Jian Chen. Characterizing provenance in visualization and data analysis: An organizational framework of provenance types and purposes. 22, 08 2015.

Stephen Few. *Information Dashboard Design: The Effective Visual Communication of Data*. O’Reilly Media, Inc., 2006. ISBN 0596100167.

James Foley. *Computer graphics : principles and practice*. Addison-Wesley, Reading, Mass, 1995. ISBN 0321210565.

D. P. Groth and K. Streefkerk. Provenance and annotation for visual exploration systems. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1500–1510, Nov 2006. ISSN 1077-2626. doi: 10.1109/TVCG.2006.101.

Pat Hanrahan. Vizql: A language for query, analysis and visualization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 721–721, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142560. URL `http://doi.acm.org/10.1145/1142473.1142560`.

Pawandeep Kaur and Michael Owonibi. A review on visualization recommendation strategies. 02 2017.

R. Likert. *A Technique for the Measurement of Attitudes*. Number Nr. 136-165 in A Technique for the Measurement of Attitudes. publisher not identified, 1932.

J. Mackinlay, P. Hanrahan, and C. Stolte. Show me: Automatic presentation for visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1137–1144, Nov 2007. ISSN 1077-2626. doi: 10.1109/TVCG.2007.70594.

Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, April 1986. ISSN 0730-0301. doi: 10.1145/22949.22950. URL `http://doi.acm.org/10.1145/22949.22950`.

J.H. McDonald and University of Delaware. *Handbook of Biological Statistics*. Sparky House Publishing, 2009.

Belgin Mutlu, Patrick Höfler, Vedran Sabol, Gerwald Tschinkel, and Michael Granitzer. Automated visualization support for linked research data. In *I-SEMANTICS*, 2013.

Belgin Mutlu, Eduardo Veas, and Christoph Trattner. Vizrec: Recommending personalized visualizations. *ACM Trans. Interact. Intell. Syst.*, 6(4):31:1–31:39, November 2016. ISSN 2160-6455. doi: 10.1145/2983923. URL `http://doi.acm.org/10.1145/2983923`.

Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, Dec 2001. ISSN 0949-877X. doi: 10.1007/s007780100057. URL `https://doi.org/10.1007/s007780100057`.

C. T. Silva, J. Freire, and S. P. Callahan. Provenance for visualizations: Reproducibility and beyond. *Computing in Science Engineering*, 9(5):82–89, Sept 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.106.

Chris Stolte and Pat Hanrahan. Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. In *Proceedings of the IEEE Symposium on Information Vizualization 2000*, INFOVIS '00, pages 5–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0804-9. URL `http://dl.acm.org/citation.cfm?id=857190.857686`.

Manasi Vartak, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. Seedb: Automatically generating query visualizations. *Proc. VLDB Endow.*, 7(13):1581–1584, August 2014. ISSN 2150-8097. doi: 10.14778/2733004.2733035. URL `http://dx.doi.org/10.14778/2733004.2733035`.

Ji Soo Yi, Youn ah Kang, John Stasko, and Julie Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, November 2007. ISSN 1077-2626. doi: 10.1109/TVCG.2007.70515. URL `http://dx.doi.org/10.1109/TVCG.2007.70515`.

Appendix

Appendix A

Config parameter actions description

This chapter describes the different actions which can be used in the `config` parameter to control *Visualizer* via URL.

A.1 removeColumns

Removes columns with given indexes.
JSON structure:

```
{
  "action": "removeColumns",
  "columnIndexes": [1,3,5]
}
```

Attribute description:

- `action`: Name of action to be executed
- `columnIndexes`: Array with the indexes of the columns to be removed

A.2 mergeColumns

Merges columns with given indexes after the dataset has been loaded.
JSON structure:

```
{
  "action": "mergeColumns",
  "aggregation": "concat",
  "columnIndexes": [1,3],
  "newColumnName": "resultColumn",
  "newColumnType": "newColumnType"
}
```

Attribute description:

- `action`: Name of action to be executed
- `aggregation`: Name of aggregation to be performed. Can be one of the following five : sum, min, max, avg and concat. sum, min, max and avg should only be used when merging numerical columns. concat can be used for all columns
- `columnIndexes`: An array with indexes of the columns to be merged
- `newColumnName`: Name of the newly created column
- `newColumnType`: Type of the newly created column

A.3 renameColumn

Changes the name of a column with a given index.
JSON structure:

```
{
  "action": "renameColumn",
  "columnIndex": 3,
  "newColumnName" : "new name of column"
}
```

Attribute description:

- `action`: Name of action to be executed
- `columnIndex`: Index of column which should be renamed
- `newColumnName`: New name of the column

A.4 aggregate

Performs an aggregation on the columns with the given indexes and the results of the aggregation are then saved in the column with the given index.
JSON structure:

```
{
  "action": "aggregate",
  "aggregations": ["min","max","’],
  "catAggregation": "count",
  "columnIndex": 1,
  "values": [5,7,8]
}
```

Attribute description:

- `action`: Name of the action to be executed
- `aggregations`: An array with the aggregations which should be performed on the columns with the indexes in the values array. The order has to be the same as in the values array. Possible aggregations are: min, max, sum, avg and "". Empty quotes should be used for columns which are not numeric.
- `catAggregation`: The type of aggregation for the categorical fields. Possible aggregations are: group, count and ratio. count - the data will be aggregated and a new column will be created which consists of the number of occurrences of data points for the selected columns. ratio - the data will be aggregated and a new column will be created which displays the quantitative relation between number of occurrences of data points for the selected columns and the total number of data points. group - the data will only be aggregated. No new column will be created
- `columnIndex`: Index of categorical column in which the values should be inserted
- `values`: The indexes of the columns which should be taken for the aggregation

A.5 filter

Filters the whole dataset by a given filter array.
JSON structure:

```
{
  "action": "filter",
  "filterArray": [filterObj1, filterObj2]
}
```

Attribute description:

- `action`: Name of the action to be executed
- `filterArray`: An array which contains `filterObj`"s with information on how to filter the dataset

A.5.1 filterObj

The `filterObj` holds all the active filters which should be applied to a column with the given index.
JSON structure:

```
{
  filterList: {
    "from": NaN
    "keywords": ["cat", "dog", "bird"]
    "to": NaN
  }
  index: 1
}
```

Attribute description:

- `filterList`:
  - `from`: Used only for numerical columns. Indicates minimum value of range
  - `to`: Used only for numerical columns. Indicates maximum value of range
  - `keywords`: An array of strings by which the column should be filtered index
- `index`: Of the column on which the filter should be applied

A.6 adddefault

Adds a default value to all empty cells in a column with the given index.
JSON structure:

```
{
  "action": "adddefault",
  "columnIndex": 1,
  "inputValue": 0
}
```

Attribute description:

- `action`: Name of the action to be executed
- `columnIndex`: The index of the column to add the default value to
- `inputValue`: The new value for the empty cells

A.7 notactive

Disables the columns on the given indexes in the `notActiveArray` array.
JSON structure:

```
{
  "action": "notactive",
  "notActiveArray": [1,3,5]
}
```

Attribute description:

- `action`: Name of the action to be executed
- `notActiveArray`: An array with indexes of columns which should be disabled

A.8 changetype

Changes the data type of a column in the dataset on the given index.
JSON structure:

```
{
  "action": "changetype",
  "columnIndex": 1,
  "newType": "integer"
}
```

Attribute description:

- `action`: Name of the action to be executed
- `columnIndex`: The index of the column for the data type change
- `newType`: The new data type of the column. Can be one of the following values: integer, number, location, date, string

A.9 sort

Sorts the column with the given index in ascending or descending order.
JSON structure:

```
{
  "action": "sort",
  "index": 2,
  "isDescending": true, // OPTIONAL
}
```

Attribute description:

- `action`: Name of the action to be executed
- `index`: The index of the column by which the dataset should be sorted
- `isDescending [optional]`: Default is false

A.10 replace

Replaces the value of one cell in a column with the given index.
JSON structure:

```
{
  "action": "replace",
  "column": 1,
  "row": 132,
  "value": "cat"
}
```

Attribute description:

- `action`: Name of the action to be executed
- `column`: The index of the column in which the value of a cell should be changed
- `row`: The row in the selected column
- `value`: The new value to be inserted

A.11 replaceall

Replaces all occurrences of a certain entry in a column, defined by the index with another one.
JSON structure:

```
{
  "action": "replaceall",
  "column": 2,
  "oldValue": "dog",
  "newValue": "cat"
}
```

Attribute description:

- `action`: Name of the action to be executed
- `column`: The index of the column where the values should be replaced
- `oldValue`: The value which should be replaced
- `newValue`: The new value

A.12 visualization

Creates a visualization window.
JSON structure:

```
{
  "action" : "visualization",
  "visualization" : "barchart",
  "labels" : "resource|hours",
  "aggregations": ["group","avg"],
  "visTitle" : "Barchart",
  "from" : 0,
  "to" : 30,
  "left" : 0,
  "top" : 0,
  "width" : 50,
  "height" : 100,
  "zIndex" : 1,
}
```

Attribute description:

- `action`: Name of the action to be executed
- `visualization`: The name of the visualization which should be created. Possible visualizations are: barchart, linechart, piechart, scatterplot, scatterplotmatrix, bubblechart, boxplot, violinPlot, wordcloud, grouped-barchart, heatmap, geovis, parallelCoordinates

- `labels`: The names of the columns which should be mapped to the visual channels of a visualization. The labels are mapped into the respective channels in the order in which they appear. See Appendix B for the description of each visualization and their visual channels
- `aggregations`: The aggregations applied to the columns. Same order and number as in the labels array. Possible values for numerical fields: min, max, sum, avg Possible values for categorical fields: group See Appendix B for more information about which visualizations have mandatory aggregations
- `visTitle [optional]`: The title of the visualization. If a title is not entered, a title will be generated which consists of the name of the visualization and the selected labels
- `from [optional]`: With the from and to parameters it is possible to reduce the number of displayed data points. from is the index of the first datapoint to be displayed. Default value is 0 (indicating the first data point)
- `to [optional]`: Index of the last data point to be displayed. Default value is -1 (indicating the last data point)
- `left [optional]`: Relative distance to the left border of the visualization window in relation to the parent (in percent). Default value is 0
- `top [optional]`: Relative distance to the top border of the visualization window in relation to the parent (in percent). Default value is 0
- `width [optional]`: Relative width of the visualization window in relation to the parent (in percent). Default value is 100
- `height [optional]`: Relative height of the visualization window in relation to the parent (in percent). Default value is 100
- `zIndex [optional]`: In the case that the visualizations are overlapping, the visualization with the highest zIndex will be on top. Default value is 1
- `sortBy [optional]`: object with the attributes “ascending” and “label”. “label” defines which field should be sorted “ascending” if true the field is sorted ascending and if false then descending

A.13 brush

Applies the brushing selection to the created visualizations.
JSON structure:

```
{
  "action": "brush",
  "selectedData": [["cat",3], ["dog","1"]],
  "labelList": ["animal","quantity"],
}
```

Attribute description:

- `action`: Name of the action to be executed
- `selectedData`: An array with the data points which should be selected in the visualizations. The order of the elements in each sub-array should be the same as in the `labelList`
- `labelList`: An array of the label names of the columns

Appendix B

Visualization description

This chapter describes the structure of the visualizations and what is required for a visualization to be created automatically.

B.1 Attributes description

- `label [string]`: The name of the visualization which the user see on the dashboard
- `fileName [string]`: The exact filename of the main file for the visualization
- `folderName [string]`: The name of the folder where all files regarding the visualization are saved
- `libs [string]`: A list of libraries which should be included before the main visualization file. The different libraries are separated by a pipe symbol '|'
- `css [string]`: A list of style sheets which should be included before the main visualization file. The different style sheets are separated by a pipe symbol '|'
- `requiredChannels [integer]`: The number of visual channels which have to be set for the visualization to be displayed
- `maxChannels [integer]`: The maximum number of channels supported by a visualization. If the value is equal to -1 then the visualization supports an arbitrary number of channels
- `maxDatapointsWithoutWarning [integer]`: After how many data points should a warning be displayed that the visualization will not be able to display the data points correctly anymore

- `hasMultiplicity [boolean]`: If the visualization has a channel which can occur multiple times
- `aggregationMandatory [boolean]`: Is aggregating the data mandatory before the visualization will be displayed
- `URIPrefix [string]`: This value is generated for new visualizations and is used for the recommender
- `URISuffix [string]`: This value is generated for new visualizations and is used for the recommender
- `hasVisualAttributes [array]`: Contains channel objects which describe each visualization channel

B.1.1 Channel object

An object which describes how a single visualization channel is structured.
JSON structure:

```
{
  Axis : {
    label : "x-Axis",
    supportedScaleOfMeasurement : [
      DataTypes.STRING,
      DataTypes.DATE,
      DataTypes.LOCATION
    ],
    hasPersistance : Persistance.MANDATORY,
    hasOccurrence : Occurrence.ONE
  }
}
```

Attribute description:

- `label [string]`: The name of the visualization channel
- `supportedScaleOfMeasurement [array]`: The datatypes supported by this channel which is one of the following values: integer, number, location, date, string
- `hasPersistance [string]`: Is a mapping to this channel mandatory or not. Possible values: mandatory and optional
- `hasOccurrence [string]`: Can the channel occur once or multiple times. Possible values: One and Multiplicity

B.2 mapping.js

This is the code of the mapping.js file, which describes all visualizations integrated in Visualizer.

```
var Persistance = {
  MANDATORY: "mandatory",
  OPTIONAL: "optional"
}

var Occurrence = {
  ONE: "One",
  MULTIPLICITY: "Multiplicity"
}

var DataTypes = {
  INTEGER: "integer",
  NUMBER: "number",
  LOCATION: "location",
  DATE: "date",
  STRING: "string"
}

var mapping = [
  // BarChart
  {
    label: "Bar Chart",
    fileName: "d3-bar.js",
    folderName: "barchart",
    libs: "jquery-3.1.0.min.js|lodash.js|bootstrap.min.js|\
d3-v3.1.7.min.js|filter-svg.js",
    css: "bootstrap.min.css|vis.css",
    requiredChannels: 2,
    maxChannels: 2,
    maxDatapointsWithoutWarning: 100,
    hasMultiplicity: false,
    aggregationMandatory: true,
    URIPrefix: "http://eexcess.eu/visualisation-ontology",
    URISuffix: "Barchart",
    hasVisualAttributes: [{
        Axis: {
          label: "x-Axis",
          supportedScaleOfMeasurement: [DataTypes.STRING,
            DataTypes.LOCATION, DataTypes.DATE
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "y-Axis",
          supportedScaleOfMeasurement: [DataTypes.INTEGER,
            DataTypes.NUMBER
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      }
    ]
  },
  // LineChart
  {
    label: "Line Chart",
    fileName: "d3-multiline-class.js",
    folderName: "linechart",
    libs: "jquery-3.1.0.min.js|lodash.js|bootstrap.min.js|\
d3-v3.1.7.min.js|filter-svg.js",
    css: "bootstrap.min.css|vis.css",
    requiredChannels: 3,
    maxChannels: 3,
    maxDatapointsWithoutWarning: 50,
    hasMultiplicity: false,
    aggregationMandatory: false,
    URIPrefix: "http://eexcess.eu/visualisation-ontology",
    URISuffix: "Linechart",
    hasVisualAttributes: [{
        Axis: {
          label: "x-Axis",
          supportedScaleOfMeasurement: [DataTypes.DATE],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "y-Axis",
          supportedScaleOfMeasurement: [DataTypes.NUMBER,
            DataTypes.INTEGER
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "Lines",
          supportedScaleOfMeasurement: [DataTypes.STRING,
            DataTypes.LOCATION
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      }
    ]
  },
  // Pie Chart
  {
    label: "Pie Chart",
    fileName: "pieChart.js",
    folderName: "piechart",
    libs: "jquery-3.1.0.min.js|lodash.js|bootstrap.min.js|\
d3-v3.1.7.min.js|filter-svg.js",
    css: "bootstrap.min.css|vis.css",
    requiredChannels: 2,
    maxChannels: 2,
    maxDatapointsWithoutWarning: 30,
    hasMultiplicity: false,
    aggregationMandatory: true,
    URIPrefix: "http://eexcess.eu/visualisation-ontology",
    URISuffix: "PieChart",
    hasVisualAttributes: [{
        Axis: {
          label: "x-Axis",
          supportedScaleOfMeasurement: [DataTypes.STRING,
            DataTypes.LOCATION, DataTypes.DATE
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "y-Axis",
          supportedScaleOfMeasurement: [DataTypes.INTEGER,
            DataTypes.NUMBER
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      }
    ]
  },
  // ScatterPlot
  {
    label: "Scatterplot",
    fileName: "d3-bubble-class.js",
    folderName: "scatterplot",
    libs: "jquery-3.1.0.min.js|lodash.js|bootstrap.min.js|\
\d3-v3.1.7.min.js|jquery.tipsy.js|filter-svg.js",
    css: "bootstrap.min.css|tipsy.css|vis.css",
    requiredChannels: 2,
    maxChannels: 3,
    maxDatapointsWithoutWarning: 100,
    hasMultiplicity: false,
    aggregationMandatory: false,
    URIPrefix: "http://eexcess.eu/visualisation-ontology",
    URISuffix: "Scatterplot",
    hasVisualAttributes: [{
        Axis: {
          label: "x-Axis",
          supportedScaleOfMeasurement: [DataTypes.NUMBER,
            DataTypes.INTEGER, DataTypes.STRING, DataTypes.LOCATION,
            DataTypes.DATE
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "y-Axis",
          supportedScaleOfMeasurement: [DataTypes.NUMBER,
            DataTypes.INTEGER, DataTypes.STRING, DataTypes.LOCATION,
            DataTypes.DATE
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "Color",
          supportedScaleOfMeasurement: [DataTypes.STRING,
            DataTypes.LOCATION, DataTypes.DATE
          ],
          hasPersistance: Persistance.OPTIONAL,
          hasOccurrence: Occurrence.ONE
        }
      }
    ]
  },
  // ScatterPlotmatrix
  {
    label: "Scatterplotmatrix",
    fileName: "d3-scatterplotmatrix.js",
    folderName: "scatterplotmatrix",
    libs: "jquery-3.1.0.min.js|lodash.js|bootstrap.min.js|\
d3-v3.1.7.min.js|filter-svg.js",
    css: "bootstrap.min.css|tipsy.css|vis.css",
    requiredChannels: 1,
    maxChannels: -1,
    maxDatapointsWithoutWarning: 99999,
    hasMultiplicity: true,
    aggregationMandatory: false,
    dontUseForRecommender: true,
    URIPrefix: "http://eexcess.eu/visualisation-ontology",
    URISuffix: "Scatterplotmatrix",
    hasVisualAttributes: [{
      Axis: {
        label: "x-Axis",
        supportedScaleOfMeasurement: [DataTypes.STRING,
          DataTypes.NUMBER, DataTypes.INTEGER, DataTypes.DATE,
          DataTypes.LOCATION
        ],
        hasPersistance: Persistance.MANDATORY,
        hasOccurrence: Occurrence.MULTIPLICITY
      }
    }]
  },
  // BubbleChart
  {
    label: "Bubble Chart",
    fileName: "d3-bubble-class.js",
    folderName: "bubblechart",
    libs: "jquery-3.1.0.min.js|lodash.js|bootstrap.min.js|\
d3-v3.1.7.min.js|jquery.tipsy.js|filter-svg.js",
    css: "bootstrap.min.css|tipsy.css|vis.css",
    requiredChannels: 4,
    maxChannels: 4,
    maxDatapointsWithoutWarning: 100,
    hasMultiplicity: false,
    aggregationMandatory: false,
    URIPrefix: "http://eexcess.eu/visualisation-ontology",
    URISuffix: "Bubblechart",
    hasVisualAttributes: [{
        Axis: {
          label: "x-Axis",
          supportedScaleOfMeasurement: [DataTypes.NUMBER,
            DataTypes.INTEGER
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "y-Axis",
          supportedScaleOfMeasurement: [DataTypes.NUMBER,
            DataTypes.INTEGER
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "Size",
          supportedScaleOfMeasurement: [DataTypes.NUMBER,
            DataTypes.INTEGER
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "Color",
          supportedScaleOfMeasurement: [DataTypes.STRING,
            DataTypes.LOCATION, DataTypes.DATE
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      }
    ]
  },
  // Boxplot
  {
    label: "Boxplot",
    fileName: "brushboxplot.js",
    folderName: "boxplot",
    libs: "jquery-3.1.0.min.js|lodash.js|bootstrap.min.js|\
d3.v3.min.js|filter-svg.js",
    css: "bootstrap.min.css|boxplot.css",
    requiredChannels: 2,
    maxChannels: 2,
    maxDatapointsWithoutWarning: 1000,
    hasMultiplicity: false,
    aggregationMandatory: false,
    URIPrefix: "http://eexcess.eu/visualisation-ontology",
    URISuffix: "Boxplot",
    hasVisualAttributes: [{
        Axis: {
          label: "x-Axis",
          supportedScaleOfMeasurement: [DataTypes.NUMBER,
            DataTypes.INTEGER
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "Category",
          supportedScaleOfMeasurement: [DataTypes.STRING,
            DataTypes.DATE, DataTypes.LOCATION
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      }
    ]
  },
  // Violin plot
  {
    label: "Violinplot",
    fileName: "violinplot.js",
    folderName: "violinPlot",
    libs: "jquery-3.1.0.min.js|lodash.js|bootstrap.min.js|\
d3.v3.min.js|filter-svg.js|distrochart.js",
    css: "bootstrap.min.css|distrochart.css",
    requiredChannels: 2,
    maxChannels: 2,
    maxDatapointsWithoutWarning: 6,
    hasMultiplicity: false,
    aggregationMandatory: false,
    URIPrefix: "http://eexcess.eu/visualisation-ontology",
    URISuffix: "Violinplot",
    hasVisualAttributes: [{
        Axis: {
          label: "x-Axis",
          supportedScaleOfMeasurement: [DataTypes.NUMBER,
            DataTypes.INTEGER
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "Category",
          supportedScaleOfMeasurement: [DataTypes.STRING,
            DataTypes.DATE, DataTypes.LOCATION
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      }
    ]
  },
  // Grouped Barchart
  {
    label: "Grouped Barchart",
    fileName: "groupedBarChart.js",
    folderName: "groupedbarchart",
    libs: "jquery-3.1.0.min.js|lodash.js|bootstrap.min.js|\
d3-v3.1.7.min.js|jquery.tipsy.js|filter-svg.js",
    css: "bootstrap.min.css|tipsy.css|vis.css",
    requiredChannels: 3,
    maxChannels: 3,
    maxDatapointsWithoutWarning: 30,
    hasMultiplicity: false,
    aggregationMandatory: true,
    URIPrefix: "http://eexcess.eu/visualisation-ontology",
    URISuffix: "Groupedbarchart",
    hasVisualAttributes: [{
        Axis: {
          label: "x-Axis",
          supportedScaleOfMeasurement: [DataTypes.STRING,
            DataTypes.LOCATION, DataTypes.DATE
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "y-Axis",
          supportedScaleOfMeasurement: [DataTypes.NUMBER,
            DataTypes.INTEGER
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "Bar",
          supportedScaleOfMeasurement: [DataTypes.STRING,
            DataTypes.LOCATION, DataTypes.DATE
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      }
    ]
  },
  // Map
  {
    label: "Map",
    fileName: "d3-geovis.js",
    folderName: "geovis",
    libs: "jquery-3.1.0.min.js|lodash.js|bootstrap.min.js|\
d3-v3.1.7.min.js|jquery.tipsy.js|filter-svg.js",
    css: "bootstrap.min.css|tipsy.css|vis.css",
    requiredChannels: 1,
    maxChannels: 2,
    maxDatapointsWithoutWarning: 400,
    hasMultiplicity: false,
    aggregationMandatory: true,
    URIPrefix: "http://eexcess.eu/visualisation-ontology",
    URISuffix: "Map",
    hasVisualAttributes: [{
        Axis: {
          label: "Country",
          supportedScaleOfMeasurement: [DataTypes.LOCATION],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "Nuance",
          supportedScaleOfMeasurement: [DataTypes.NUMBER,
            DataTypes.INTEGER
          ],
          hasPersistance: Persistance.OPTIONAL,
          hasOccurrence: Occurrence.ONE
        }
      }
    ]
  },
  // Parallel Coordinates
  {
    label: "Parallel Coordinates",
    fileName: "parallelCoordinates.js",
    folderName: "parallelCoordinates",
    libs: "jquery-3.1.0.min.js|lodash.js|bootstrap.min.js|\
d3-v3.1.7.min.js|jquery.tipsy.js|filter-svg.js",
    css: "bootstrap.min.css|tipsy.css|vis.css",
    requiredChannels: 2,
    maxChannels: -1,
    maxDatapointsWithoutWarning: 40,
    hasMultiplicity: true,
    aggregationMandatory: false,
    stopAfterFirstMatch: true,
    URIPrefix: "http://eexcess.eu/visualisation-ontology",
    URISuffix: "ParallelCoordinates",
    hasVisualAttributes: [{
        Axis: {
          label: "color",
          supportedScaleOfMeasurement: [DataTypes.STRING,
            DataTypes.DATE, DataTypes.LOCATION
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.ONE
        }
      },
      {
        Axis: {
          label: "x-Axis",
          supportedScaleOfMeasurement: [DataTypes.STRING,
            DataTypes.NUMBER, DataTypes.INTEGER, DataTypes.DATE,
            DataTypes.LOCATION
          ],
          hasPersistance: Persistance.MANDATORY,
          hasOccurrence: Occurrence.MULTIPLICITY
        }
      }
    ]
  },
]
```

Appendix C

Chart API

C.1 Global Objects

- `gaDatarows [array]`: An array which holds all of the data rows of the visualization
- `gaChannelMappings [array]`: An array of objects which describe what field was mapped to what visual channel of the visualization and how the field was aggregated and the type of the field
- `gnChartRowIndex [integer]`: The index of the visualization (needed for brushing)
- `gnFrom [integer]`: An integer which has the index of the first value if the data rows were sliced. -1 if dataset was not sliced
- `gnTo [integer]`: An integer which has the index of the last value if the data rows were sliced. -1 if dataset was not sliced
- `gaLabelList [array]`: The list of labels of the current visualization
- `gaFilteredDatarows [array]`: A subset of the data rows, which holds only the data rows filtered by the framework
- `brushingObserver [object]`: A global object which is used for sending and getting updates from the other visualizations

C.2 brushingObserver

C.2.1 registerListener

Registers a visualization to the `brushingObserver`.
Parameters:

- `callbackUpdateChartRowIndex [function]`: Callback function used to update index of visualization
- `chartRowIndex [integer]`: Index of visualization to register
- `callbackData [function]`: Callback function to be called when an update is called based on raw data from
- `callbackDimensions [function]`: Could be used to check raw data. Not in use currently

C.2.2 unregister

Called to remove a visualization from the update list of the brushing observer. This is done automatically when a visualization gets closed.
Parameters:

- `chartRowIndex [integer]`: Index of the visualization which should be unregistered

C.2.3 update

Function which should be used to update the views of other visualizations. Should be called every time a selection on the source visualization happens.
Parameters:

- `originChartRowIndex [integer]`: Index of visualization from which the update originates. The origin visualization gets skipped in the update
- `selectedData [array]`: An array of all the selected values
- `selectedDimensions [array]`: An array of all raw values
- `labelList [array]`: An array of field names from which the `selectedData` comes

C.2.4 initializeSelfUpdate

Can be used to call a `brushingUpdate` on itself.
Parameters:

- `selfChartRowIndex [integer]`: Index of the visualization from which the update originates

C.2.5 updateEmpty

Calls an update on all the visualizations with no data. Can be used to clear selections.
Parameters:

- `originChartRowIndex [integer]`: Index of visualization from which the update originates

C.3 Chart Functions

C.3.1 getChannelMappings

Returns `gaChannelMappings`.
Parameters: (none)

C.3.2 getChartRowIndex

Returns `gnChartRowIndex`.
Parameters: (none)

C.3.3 getFrom

Returns `gnFrom`.
Parameters: (none)

C.3.4 getTo

Returns `gnTo`.
Parameters: (none)

C.3.5 filterBy

Handles the filtering of the data rows and fills the `gaFilteredDatarows` array.
Parameters:

- `filterObjs [array]`: An array of objects holding the values for each field to filter by. Each object has two attributes: label and values.
  - `label [string]`: Name of field
  - `values [array]`: Values to filter the field by

C.3.6 updateVisualization

Swaps the fields to visual channel mapping and redraws the visualization.
Parameters:

- `labelList [array]`: The list of fields in the new order

C.3.7 sortBy

Sorts the `gaDatarows` array.
Parameters:

- `label [string]`: Name of field which the data should be sorted
- `ascending [boolean]`: true -> ascending, false -> descending

C.3.8 unregisterVisualization

Removes the visualization from the global `brushingObserver`.
Parameters: (none)

C.3.9 refreshVisualization

Redraws the visualization and applies the set filters.
Parameters: (none)

C.3.10 showVisualization

Is called upon creation. Sets the visualization global variables, sorts the data rows if needed and draws the visualization.
Parameters:

- `datarows [array]`: A two-dimensional array holding the data rows of the selected fields
- `channelMappings [array]`: An array of objects which describe the mapping of selected field to visual channels. Each object has the following attributes:
  - `channel [string]`: Name of visual channel
  - `label [string]`: Name of selected field
  - `datatype [string]`: Datatype of selected field
  - `aggregation [string]`: Aggregation of selected field

- `targetSelector [string]`: The CSS target selector of the element where the visualization should be appended. In *Visualizer's* case it is always "body"
- `chartRowIndex [integer]`: Index of the visualization. The index gets incremented with each created visualization
- `from [integer]`: Index of the first element if data was sliced. Only used for bookmarking purposes
- `to [integer]`: Index of the last element if data was sliced. Only used for bookmarking purposes
- `sortByName [string]`: Name of the field to sort by
- `ascending [boolean]`: true -> sort ascending, false -> sort descending

C.3.11 applyCssToSvg

To be implemented per chart. It should apply styles inline to the SVG, so that the visualization can be downloaded.
Parameters: (none)

C.3.12 revertCssFromSvg

To be implemented per chart. It should revert the changes of applyCssToSvg if needed.
Parameters: (none)

C.3.13 applyFilter

To be implemented per chart. It should apply the filtered datarows to the visualization.
Parameters:

- `filteredDatarows [array]`: A subset of `gaDatarows`, which holds only the data rows which fulfill the constraints of the filters

C.3.14 drawVisualization

To be implemented per chart. It Should draw the visualization based on the datarows and `channelMappings`.
Parameters:

- `datarows [array]`: the `gaDatarows` parameter passed from the `showVisualization` function

- `channelMappings [array]`: the `channelMappings` parameter passed from `showVisualization`
- `visIndex [integer]`: the `chartRowIndex` parameter passed from `showVisualization`. Renamed due to easier understanding

Appendix D

User visualization code

D.1 Sankey diagram - original index.html

The source code was taken from:
`http://bl.ocks.org/d3noob/c9b90689c1438f57d649`

```
<!DOCTYPE html>
<meta charset="utf-8">
<title>SANKEY Experiment</title>
<style>
.node rect {
  cursor: move;
  fill-opacity: .9;
  shape-rendering: crispEdges;
}

.node text {
  pointer-events: none;
  text-shadow: 0 1px 0 #fff;
}

.link {
  fill: none;
  stroke: #000;
  stroke-opacity: .2;
}

.link:hover {
  stroke-opacity: .5;
}
</style>
```

```
<body>
<p id="chart">

<script src="http://d3js.org/d3.v3.min.js"></script>
<script src="sankey.js"></script>
<script>

var units = "Widgets";

var margin = {top: 10, right: 10, bottom: 10, left: 10},
    width = 700 - margin.left - margin.right,
    height = 300 - margin.top - margin.bottom;

var formatNumber = d3.format(",.0f"),    // zero decimal places
    format = function(d) { return formatNumber(d) + " " + units; },
    color = d3.scale.category20();

// append the svg canvas to the page
var svg = d3.select("#chart").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
    .append("g")
    .attr("transform",
          "translate(" + margin.left + "," + margin.top + ")");

// Set the sankey diagram properties
var sankey = d3.sankey()
    .nodeWidth(36)
    .nodePadding(40)
    .size([width, height]);

var path = sankey.link();

// load the data (using the timelyportfolio csv method)
d3.csv("sankey.csv", function(error, data) {

  //set up graph in same style as original example but empty
  graph = {"nodes" : [], "links" : []};
    data.forEach(function (d) {
      graph.nodes.push({ "name": d.source });
      graph.nodes.push({ "name": d.target });
      graph.links.push({ "source": d.source,
                         "target": d.target,
                         "value": +d.value });
   });

   // return only the distinct / unique nodes
   graph.nodes = d3.keys(d3.nest()
     .key(function (d) { return d.name; })
     .map(graph.nodes));
```

```
// loop through each link replacing the text with its index from node
graph.links.forEach(function (d, i) {
  graph.links[i].source = graph.nodes.indexOf(graph.links[i].source);
  graph.links[i].target = graph.nodes.indexOf(graph.links[i].target);
});

//now loop through each nodes to make nodes an array of objects
// rather than an array of strings
graph.nodes.forEach(function (d, i) {
  graph.nodes[i] = { "name": d };
});

sankey
  .nodes(graph.nodes)
  .links(graph.links)
  .layout(32);

// add in the links
var link = svg.append("g").selectAll(".link")
    .data(graph.links)
  .enter().append("path")
    .attr("class", "link")
    .attr("d", path)
    .style("stroke-width", function(d) { return Math.max(1, d.dy); })
    .sort(function(a, b) { return b.dy - a.dy; });

// add the link titles
link.append("title")
      .text(function(d) {
      return d.source.name + " ? " +
             d.target.name + "\n" + format(d.value); });

// add in the nodes
var node = svg.append("g").selectAll(".node")
    .data(graph.nodes)
  .enter().append("g")
    .attr("class", "node")
    .attr("transform", function(d) {
      return "translate(" + d.x + "," + d.y + ")"; })
  .call(d3.behavior.drag()
    .origin(function(d) { return d; })
    .on("dragstart", function() { this.parentNode.appendChild(this); })
    .on("drag", dragmove));
    // add the rectangles for the nodes
    node.append("rect")
        .attr("height", function(d) { return d.dy; })
        .attr("width", sankey.nodeWidth())
        .style("fill", function(d) {
          return d.color = color(d.name.replace(/ .*/, "")); })
        .style("stroke", function(d) { return d3.rgb(d.color).darker(2); })
        .append("title")
        .text(function(d) { return d.name + "\n" + format(d.value); });

    // add in the title for the nodes
    node.append("text")
        .attr("x", -6)
        .attr("y", function(d) { return d.dy / 2; })
        .attr("dy", ".35em")
        .attr("text-anchor", "end")
        .attr("transform", null)
        .text(function(d) { return d.name; })
      .filter(function(d) { return d.x < width / 2; })
        .attr("x", 6 + sankey.nodeWidth())
        .attr("text-anchor", "start");

    // the function for moving the nodes
    function dragmove(d) {
      d3.select(this).attr("transform",
         "translate(" + d.x + "," + (
                 d.y = Math.max(0, Math.min(height - d.dy, d3.event.y))
             ) + ")");
      sankey.relayout();
      link.attr("d", path);
    }
});
</script>
</body>
</html>
```

D.2 Sankey.csv

```
source,target,value
Barry,Elvis,2
Frodo,Elvis,2
Frodo,Sarah,2
Barry,Alice,2
Elvis,Sarah,2
Elvis,Alice,2
Sarah,Alice,4
```