Peter HASITSCHKA BSc.

Visualization and Analysis of Recommendation Histories using WebGL

Master's Thesis

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme: Software Development and Business Management

submitted to
Graz University of Technology

Supervisor
Dipl.Ing. Dr.techn. Vedran SABOL

Co-advisor
Dr.techn. Eduardo Enrique VEAS MSc.

Knowledge Technologies Institute

Graz, September 2016

Statutory Declaration

Eidesstattliche Erklärung

Abstract

Content-based recommender systems are commonly used to automatically provide context-based resource suggestions to users. This work introduces *ECHO* (***E**xplorer of **C**ollection **H**ist**O**ries*), a visual tool supporting visualization of a recommender system's entire query history. It provides an interactive three-dimensional scene resembling the *CoverFlow* layout to browse through all collections in several Levels of Detail, compare collections, and find similarities in previous result sets. The user has the possibility to analyze a single collection through an intuitive visual representation of the results and their metadata, which is embedded into the 3D scene. These visualizations give insights into the metadata distribution of a collection and support the application of faceted filters on the whole query-history. Search results can be explored by the user in detail, organized in bookmark-collections for a later usage, and may also be used in external tools such as editors. The *ECHO* implementation supports graphics card acceleration to avoid rendering performance issues and to provide smooth, animated transitions by using the *WebGL* technology.

Kurzfassung

Durch den Einsatz inhaltsbasierter Empfehlungssysteme können Benutzern automatisiert kontextbasierte Empfehlungen geliefert werden. In dieser Arbeit wird mit *ECHO* (***E****xplorer of* ***C****ollection* ***H****ist****O****ries*) ein Werkzeug vorgestellt, welches es erlaubt, vollständige Such-Historien eines Empfehlungssystems zu visualisieren. Es bietet eine dreidimensionale Szene, welche dem *CoverFlow* Layout ähnelt. In dieser können Benutzer sämtliche Suchergebnis-Sammlungen in verschieden Detaillierungsgraden durchsuchen, mehrere Sammlungen vergleichen und weiters Ähnlichkeiten in vorhergehenden Sammlungen aufspüren. Eine, in diese 3D-Szene eingebettete, intuitive visuelle Repräsentation der Suchergebnisse und deren Meta-Daten erlaubt Benutzern einzelne Sammlungen zu analysieren. Durch diese Visualisierungen können Benutzer Einblicke in die Verteilung der Meta-Daten einer Sammlung erlangen und facettierte Filter auf die gesamte Such-Historie anwenden. Die einzelnen Suchergebnisse können durch Benutzer im Detail erforscht, in Lesezeichen-Sammlungen für den späteren Gebrauch gespeichert und in externe Werkzeuge wie zum Beispiel Editoren eingebunden werden. Um Leistungsprobleme zu verhindern, unterstützt *ECHO* grafikkartenbasierte Hardwarebeschleunigung und erlaubt dadurch das Darstellen von flüssigen, animierten Übergängen durch den Einsatz der *WebGL*-Technologie.

Acknowledgments

After months of hard work I write these lines as the finalization of my thesis. This wouldn't have been possible without the support and expertise of my supervisor Dr. Vedran Sabol. I want to express my thanks to him for his great guidance and patience during this exciting time.

Furthermore I want to thank Dr. Eduardo Veas for his scientific advices and my colleague Gerwald Tschinkel for supporting me during implementing *ECHO*.

With finishing this thesis also my Master-Study at the Graz University of Technology finds an end. This would not have been possible without the most important people in my life. I want to express my deepest gratitude to my parents who made this study possible for me and gave me all support I needed through this time and I need to thank my wife Sarah for her love and motivation whenever I needed it during the process of creating this work.

Contents

List of Figures

1 Introduction

The first chapter discusses the project's motivation (Section 1.1), and provides a preview of the work discussed in the following chapters (Section 1.2) as well as an overview of this work's structure (Section 1.3).

1.1 Motivation

Recommender systems help the user in finding resources from a network, that match best depending on factors like user preferences or previous results. *EEXCESS* (Chapter 3) is a project that supports people while using their web-browser to read or create content by recommending documents containing educational, cultural and scientific information from several providers.

The project provides the possibility to analyze a single query by using the built-in *Recommendation Dashboard* (*RD*) (Section 3.5.3). However, former queries and their results are no longer accessible. Saving and listing previous search-queries allows the user to access them, but "browsing through a long list of documents to locate the information needed could be a mentally exhausting task" (Chau, 2011, p. 2). Since the human visual system has enormous power to perceive information from visualized data (Ware, 2012) it leads to considerations about a proper visualization of *EEXCESS*' query history. Further it should be possible to visualize a set of the *RD*'s bookmark-collection to allow analyze already stored results.

The user should not only be able to see the different query- or bookmark-collections, but also to navigate through the graph to easily access specific collections or their recommendations.

The possibility to visualize relations between collections to show similarities between different queries could help identify other interesting collections during navigation.

Since recommendations contain several metadata-values, the user should be able to see an intuitive summary of those values all over the results of one collection. This summary could provide a visual *characterization* of the collection. Metadata result should also be used to apply filters that help the user to narrow down the result set, and find desired documents within all collections.

1.2 Overview

To realize the ideas mentioned in Section 1.1, the ***E****xplorer of* ***C****ollection* ***H****ist****O****ries* (***ECHO***) was developed.

It provides a 3D-visualization of the query history or a set of bookmark-collections in the user's browser, where the hardware-acceleration of the graphic card is accessed through the *WebGL*-API. The user can easily browse through the scene by using the keyboard or the mouse for navigation. It is possible to access different Levels of detail (such as an overview over the different queries or detailed result information) that provide different information and interaction possibilities.

The recommendation results are visualized as a ring of icons surrounding the collection node, which is placed in the center. The distance from the center and size of the recommendation nodes represent their relevance inside the collection. The space between the results and the center node is used to show the results' facet-values as a sunburst-diagram (Chapter 4.2.4) and makes it possible to recognize the collection's metadata distribution, but it can also be used as an interface for applying filters at the same time.

Recommendation nodes that can contain a preview image of the underlying document can be used to gain more information about the underlying document or to access it through external tools for content-creation (*Wordpress*, *Google-Docs*) that are linked to the *RD*.

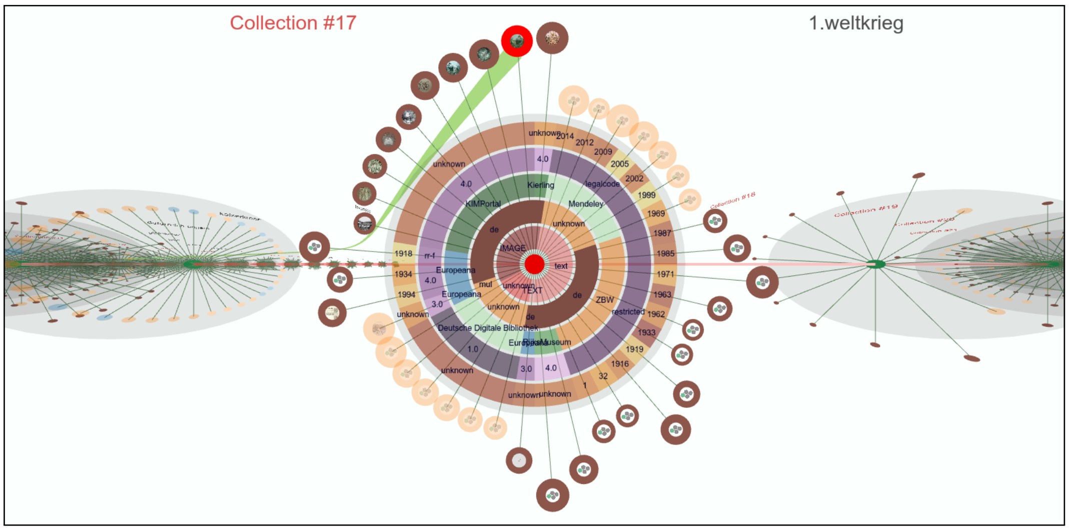


Figure 1.1: *Focusing a query-collection in ECHO with an activated filter and the visualization of a relation to another collection.*

The user can activate a comparison mechanism that allows them to visualize relations to other collections by linking the same documents over the whole scene.

Figure 1.1 shows a demonstration of *ECHO* in use: a query-collection of results was focused while its results are shown around its center. A filter for the value ″de″ on the *language*-facet is activate in the figure. The green spline on the top refers to another collection that also shares the highlighted recommendation.

1.3 Structure

This work is structured as follows:

- Chapter 2 provides an overview of **related work** on recommender systems and graph visualization.
- Chapter 3 gives an introduction to the ***EEXCESS*** project and the *RD*.

- Chapter 4 is the main chapter of this work, and it describes the functionality of ***ECHO*** in detail.
- Chapter 5 goes into details of *ECHO*'s **implementation** and discusses selected details of the implementation.
- Chapter 6 provides a **case study** to demonstrate the full functionality of *ECHO* and its benefits in a typical scenario.
- Chapter 7 concludes this work by providing **summary**, a discussion of lessons learned, and suggestions for potential future work.

2 Related work

Before discussing further details of this project, it is necessary to provide some background information on the topics of *recommender systems*, *graph visualization* and finally the *visualization techniques for recommender systems* in the following chapter.

2.1 Recommender Systems

Recommender systems or *recommendation systems* (RSs) are built to automatically (i.e. without an explicit user request) provide relevant items to the users of specific systems, depending on their earlier behavior (like doing search queries or buying specific items) on the system. Typical RSs work with algorithms to suggest a personalized list of items to the user. Other systems that do not personalize items, such as those that work by providing the top ten list of bought books in an online-shop, may be more simple but are usually not addressed by research on RSs (Ricci et al., 2010, p. 2).

2.1.1 Development of Recommender Systems Over the Years

An initial motivation for developing RSs could be based on a simple human behavior: people tend to make their real-life decisions using recommendations (Ricci et al., 2010, p. 2). For instance, an employer uses recommendation-letters as a decision-making aid. When deciding whether or not to buy a book or watch a movie, recommendations can have a massive influence on that decision.

When RSs began to evolve into an independent field of scientific research in the 1990s, there was a high focus on letting other users of the same system rate results, and on using simple algorithms by comparing the similarities of users to retrieve accurate results. This approach is called*collaborative filtering* (CF).

With the rise of e-commerce, users had to decide between hundreds of products in online-shops. Thus, it became a more and more important task to prevent an overload of products and to offer a small amount of best-matching items. Scientists began to improve RSs by using more complex algorithms and machine learning techniques, and started to track and use more data about the user's behavior. Thus, not only purchased items were the base data for recommendations, but also the user's navigation, product-ratings, and reactions to previously recommended products like declining or clicking on them.

(Konstan, 2004), mentions five works that influenced RSs in the first years of the 21st century.

- (Herlocker et al., 2004) investigated the similarities in the performance of several published accuracy metrics on one content domain. First, they came to the conclusion that those metrics do not measure the same things explicitly but can be grouped into clusters with similar measurements. Secondly, they found that not only those accuracy measurements are important when rating a RS's usefulness but also factors like the novelty of a result.
- (Middleton, Shadbolt, and Roure, 2004), cover improvements of RSs by creating profiles of the users' interests using ontologies of research article topics. This technology helps prevent the cold-start problem of (new) RSs, which affects CF-based systems.
- In (Hofmann, 2004), "probabilistic latent semantic analysis and expectation-maximization algorithms" (Konstan, 2004, p. 2) were used to reduce dimensions and further build up a *preference space*. The model describes independent factors that describe users' preferences as weighted vectors which leads to better predictions in RSs.
- The fourth article mentioned is (Huang, Chen, and Zeng, 2004), which deals with sparse RSs. The authors investigate the influence of

spreading-activation algorithms to improve the recommender's results for new as well as for existing users.
- Finally, (Deshpande and Karypis, 2004), describes ways to find a list of best matching items as is common in e-commerce systems. Therefore, algorithms based on co-purchase are used.

Over the years, development of RSs have become a multi-discipline challenge: specialists for "Artificial intelligence, Human Computer Interaction, Information Technology, Data Mining, Statistics, Adaptive User Interfaces, Decision Support Systems, Marketing, or Consumer Behavior" (Ricci et al., 2010, p. vii) are all involved in building different kinds of RSs.

Nowadays every internet user is faced with RSs. Well-known e-commerce companies like *Amazon.com* or multimedia platforms such as *Netflix*, *YouTube* and *Spotify* all recommend products, videos or music depending on the user's preferences. The value of a perfect RS can be illustrated by the *Netflix prize* that offers $ 1 Million USD for significantly improving its RS (Ricci et al., 2010, p. 3).

Even social-media platforms like *Facebook* and *Twitter* make use of RSs by suggesting people to *follow* or to *add as a friend*. Dealing with user preferences and connections to suggest other users in Web 2.0 emerged into a new subtopic called *social RSs* (Ricci et al., 2010, p. 646).

2.1.2 Visualization For Recommender Systems

This section gives an overview of possibilities for visualizing a recommender's data following a number of differently motivated approaches.

2.1.2.1 TalkExplorer

An approach for visualizing results of RSs was proposed by (Verbert et al., 2013). Specifically, the authors created a tool to improve the *Conference Navigator 3* (CN3)[1].

[1]Parra et al., 2012.

CN3 is a social platform created for academic conference attendees that allows the users to browse through conferences and talks and find information about other people related to a specific conference, like authors or other attending users. It is possible to create personal schedules and add tags to talks. Users can follow each other, as on other social networks, and can browse lists of the most popular tags, most popular articles, most active users, etc. CN3 can recommend conferences and talks to users based on the collected information (*"People who scheduled this presentation also scheduled ..."*).

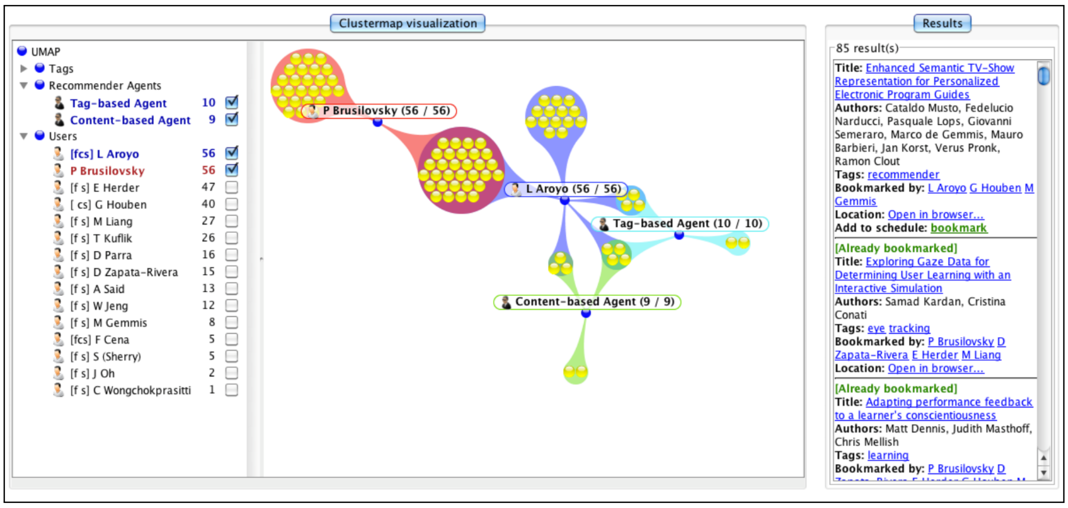
In order to visually connect all of this information, the authors developed a component called *TalkExplorer*. Using the *Aduna clustermap* visualization library, the tool allows interactive visualizations of CN3's data which allow explorations and control by the end user. In detail, the author's goal was to visualize the users' interactions, tags and recommender agents. Figure 2.1 shows the *neighborhood* of different users and items they bookmarked (yellow dots).

Finally, the user has the possibility to explore "both interrelationships between users as well as interrelationships between agents and users (i.e. which other users have bookmarked talks that are recommended to them by one or more agents)" (Verbert et al., 2013, p. 5). Relations between users, tags, and talks are also visualized to help users find relevant talks to attend.

2.1.2.2 PeerChooser

In RSs based on CF, sparsely populated matrices, which hold correlations between rated items and users, can be a big problem for identifying similar users or items to recommend due to a lack of ratings overlap. (O'Donovan et al., 2008), tried to deal with that problem by visualizing similarities between the actual user and other users and their preferences.

In particular, they described the problem through a movie-recommender system where users can rate movies with specific genres. To visualize the relations to other users and movie genres they like, the authors developed *PeerChooser*. This Java and OpenGL based tool tries to support the user in



UMAP
- Tags
- Recommender Agents

Tag-based Agent	10	✓
Content-based Agent	9	✓

- Users

[fcs] L Aroyo	56	✓
P Brusilovsky	56	✓
[f s] E Herder	47	☐
[cs] G Houben	40	☐
[f s] M Liang	27	☐
[f s] T Kuflik	26	☐
[f s] D Parra	16	☐
[f s] D Zapata-Rivera	15	☐
[f s] A Said	13	☐
[f s] W Jeng	12	☐
[f s] M Gemmis	8	☐
[fcs] F Cena	5	☐
[f s] S (Sherry)	5	☐
[f s] J Oh	2	☐
[f s] C Wongchokprasitti	1	☐

85 result(s)

Title: Enhanced Semantic TV-Show Representation for Personalized Electronic Program Guides
Authors: Cataldo Musto, Fedelucio Narducci, Pasquale Lops, Giovanni Semeraro, Marco de Gemmis, Mauro Barbieri, Jan Korst, Verus Pronk, Ramon Clout
Tags: recommender
Bookmarked by: L Aroyo G Houben M Gemmis
Location: Open in browser...
Add to schedule: bookmark

[Already bookmarked]
Title: Exploring Gaze Data for Determining User Learning with an Interactive Simulation
Authors: Samad Kardan, Cristina Conati
Tags: eye tracking
Bookmarked by: P Brusilovsky D Zapata-Rivera E Herder M Liang
Location: Open in browser...

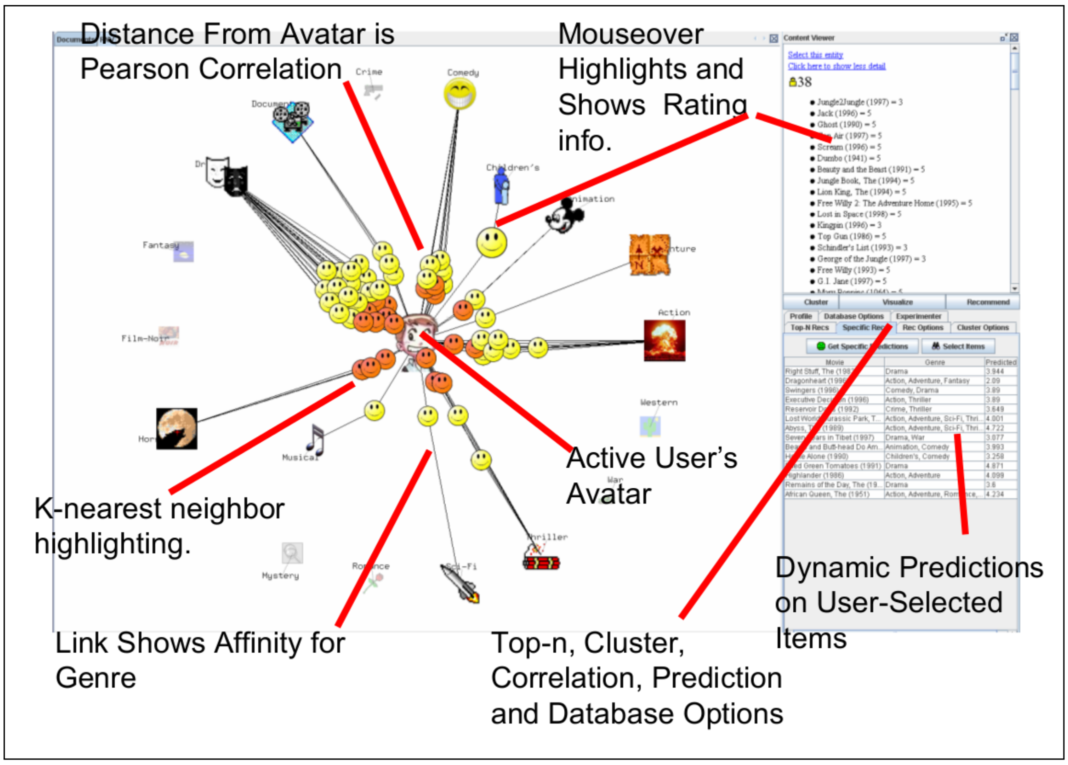
[Already bookmarked]
Title: Adapting performance feedback to a learner's conscientiousness
Authors: Matt Dennis, Judith Masthoff, Chris Mellish
Tags: learning
Bookmarked by: P Brusilovsky D

Figure 2.1: *TalkExplorer* (Verbert et al., 2013, p. 5)

finding other similar users by visualizing the mentioned connections and by finding user-groups with similar interests, so called peer groups.

The visualization works within a force-directed graph where the actual user is in the center, and the movie-genres are positioned around in a circle. Each other user is visualized through an icon, whose position depends on that user's preferences. Thus, multiple users with similar interests are clustered in similar positions. Figure 2.2 gives an example of how such relations would look in *PeerChooser*.

PeerChooser also supports interaction: the user has the possibility to manipulate the visualized graph by deleting or moving elements. For example, if the user moves a genre closer to the graph's center its importance increases and the positions of other users connected to the genre also change. Finally the k-nearest users (orange icons in Figure 2.2) are used to calculate actual recommendations of movies for the user.



Select this entity
Click here to show less detail
38

- Jungle2Jungle (1997) = 3
- Jack (1996) = 5
- Ghost (1990) = 5
- Con Air (1997) = 5
- Scream (1996) = 5
- Dumbo (1941) = 5
- Beauty and the Beast (1991) = 5
- Jungle Book, The (1994) = 5
- Lion King, The (1994) = 5
- Free Willy 2: The Adventure Home (1995) = 5
- Lost in Space (1998) = 5
- Kingpin (1996) = 3
- Top Gun (1986) = 5
- Schindler's List (1993) = 3
- George of the Jungle (1997) = 3
- Free Willy (1993) = 5
- G.I. Jane (1997) = 5

Movie	Genre	Predicted
Right Stuff, The (198	Drama	3.944
Dragonheart (1996)	Action, Adventure, Fantasy	2.09
Swingers (1996)	Comedy, Drama	3.89
Executive Decision (1996)	Action, Thriller	3.89
Reservoir Dogs (1992)	Crime, Thriller	3.649
Lost World: Jurassic Park, T...	Action, Adventure, Sci-Fi, Thri...	4.001
Abyss, The (1989)	Action, Adventure, Sci-Fi, Thri...	4.722
Seven Years in Tibet (1997)	Drama, War	3.077
Beavis and Butt-head Do Am...	Animation, Comedy	3.993
Home Alone (1990)	Children's, Comedy	3.258
Fried Green Tomatoes (1991)	Drama	4.871
Highlander (1986)	Action, Adventure	4.099
Remains of the Day, The (19...	Drama	3.6
African Queen, The (1951)	Action, Adventure, Romance,...	4.234

Figure 2.2: Visualizing similar users of a movie-recommender with *PeerChooser* (O'Donovan et al., 2008, p. 2)

2.2 Graph Visualizations

As this work focuses on visualizing multiple document collections and providing interactive comparison mechanisms, the following section gives an overview of different graph comparison techniques and comparison of document collections particularly as well as background on visualizing time oriented graphs.

2.2.1 Comparing Graphs

This section should provide an overview of some work on comparing graphs in general. As every research and implementation focuses on domain-specific data from different sources with different visualization goals, the approaches vary greatly.

2.2.1.1 TreeJuxtaposer

An early work in the field of graph comparison was done by (Munzner et al., 2003). It describes the *TreeJuxtaposer* tool, which was implemented to compare huge trees with hundreds of thousands of nodes in a scalable way. The tool faces the following challenges:

- **Automatic identification of structural differences**: Finding appropriate algorithms for finding differences in trees.
- **Differences characterization**: It is not only necessary to know that trees are different but also how they differ.
- **Scalability in tree and display size**: The work should support millions of nodes in the future. Thus, it is necessary to consider such scales in navigation and comparison.

To manage these problems, the authors made use of the following techniques:

- **Structural comparison**: *TreeJuxtaposer* tries to find the *most similar* node in another tree for each node. This makes it possible to highlight the corresponding node when hovering over a node with the mouse.

The algorithm runs almost in linear preprocessing time and in almost constant time during lookup.
- **Guaranteed visibility**: It is important that the user always sees the currently highlighted tree, regardless of the current performed navigation.
- ***AccordionTree* navigation**: *TreeJuxtaposer* uses algorithms "based on global rectangular Focus+Context distortion" (Munzner et al., 2003, p. 2) adapted to phylogenetic trees to provide the user an appropriate navigation.

The compared trees are visualized in two columns where similar subtrees of both sides are highlighted in the same color. Trees are rendered in a rectilinear layout where the root lays on the left side and the tree develops to the right side of the column. *TreeJuxtaposer* supports several mechanisms of interactions to identify nodes. The user can highlight a specific node by mouse-over where the best matching node of the other tree is also highlighted. Furthermore, it is possible to perform a search query to get a node by its name. Highlighted nodes can be expanded by clicking on them.

The system's visualization underlies the *Focus+Context* "approach of showing an area of distorted aggregate context around an easily changeable focus point to allow a large overview integrated with details in limited screen real estate." (Munzner et al., 2003, p. 2)

Figure 2.3 shows a comparison-process of two different genetic trees using *TreeJuxtaposer*.

2.2.1.2 Semantic Graph Visualiser

Another graph comparison tool is described by (Andrews, Wohlfahrt, and Wurzinger, 2009), as *Semantic Graph Visualiser* (SGV) where the focus lies on comparing two graphs which both visualize similar business models. Aside from the comparison, the two graphs should be able to be merged into a single graph.

The comparison using SGV works in six steps (Andrews, Wohlfahrt, and Wurzinger, 2009, p. 3):

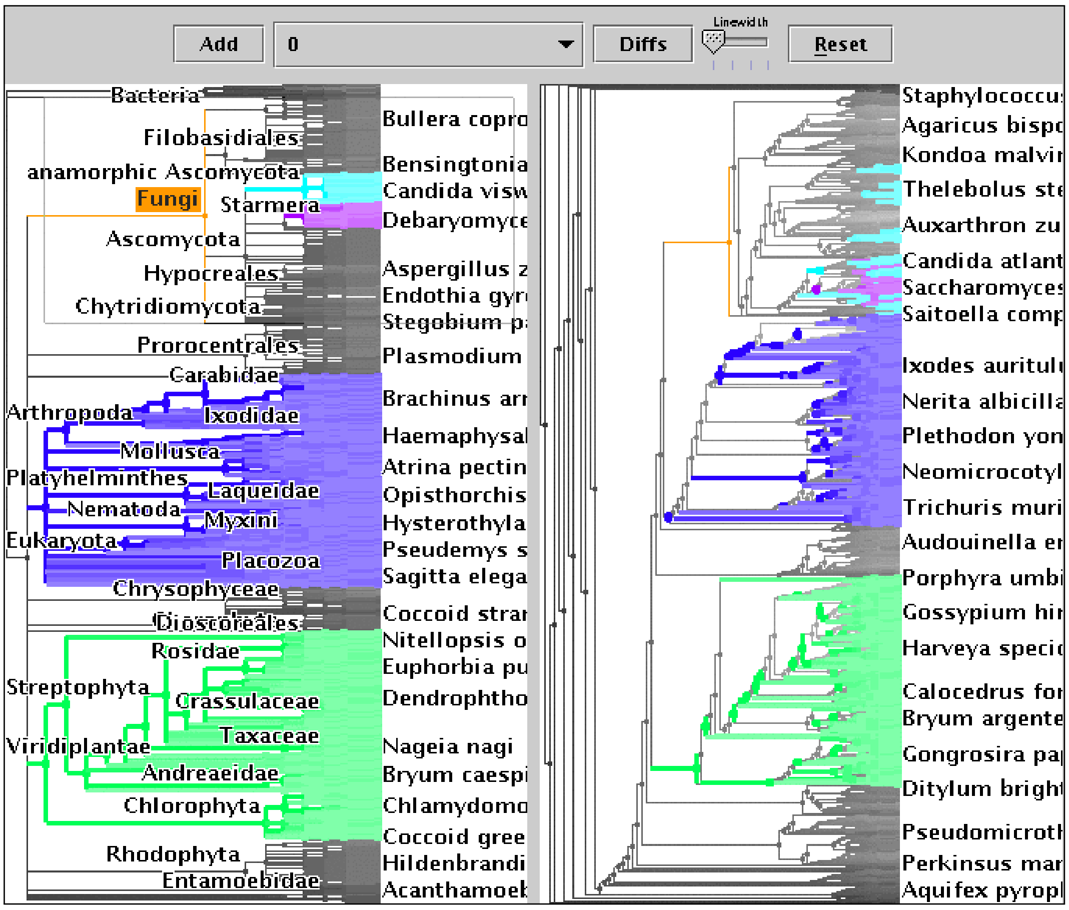


Figure 2.3: Usage sample of the *TreeJuxtaposer* comparing different genetic trees (Munzner et al., 2003, p. 6).

- **Reading two input graphs**: Two graphs described in GraphML[2] (Brandes, Eiglsperger, et al., 2010) are read and a similarity matrix for each pair of nodes in the graphs is created.
- **Finding matching nodes**: Using the *Hungarian Algorithm* (Kuhn, 1955) for each node, a matching node from the other graph is searched.
- **Creating the merged graph**: A new graph containing the nodes from the first graph and the not-matching nodes from the second graph is created and connected.
- **Layouting**: Either a layout-algorithm is applied to the new graph or the user makes use of the possibility to adjust the layout by hand.
- **Applying the layout to the old graphs**: The newly calculated layout of the merged graph is applied to both old graphs to make side-by-side comparison easier.
- **Manual editing**: Finally, the user has the chance to edit the positions or labels of the merged graph.

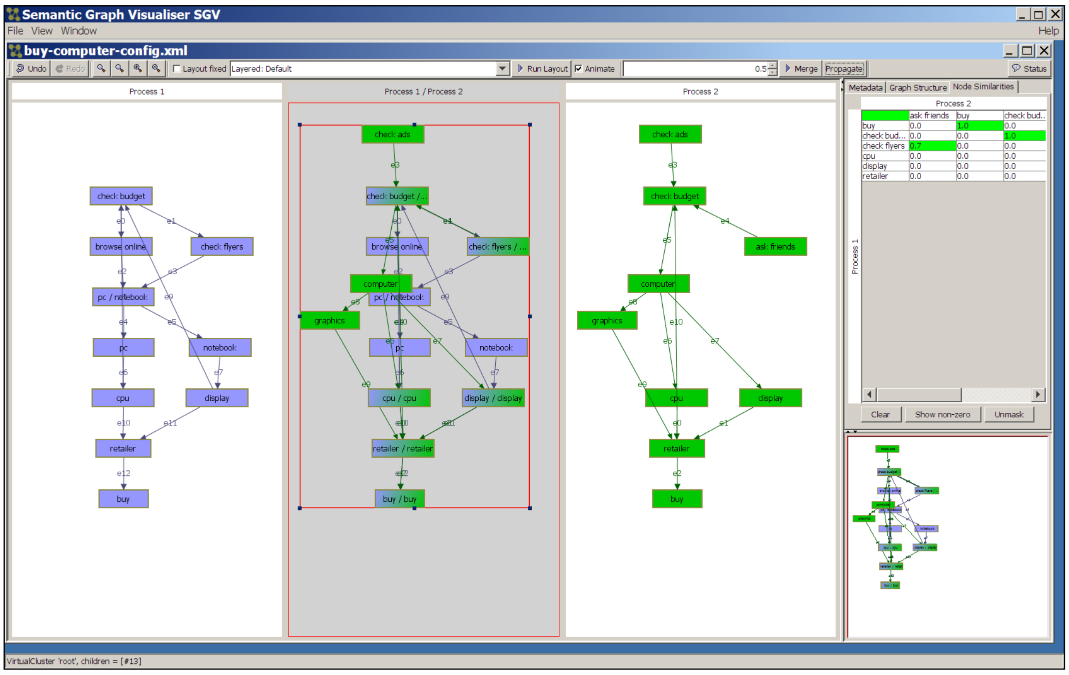
Figure 2.4 shows the usage of SGV in a sample of two graphs describing a computer-buying-process.

2.2.1.3 Difference maps With Hierarchies

An interesting approach for comparing graphs is described by (Archambault, 2009). The underlying idea is to display graphs that contain the difference of two graphs to compare (Figure 2.5). Those differences concern both edges and vertices. Such ***difference maps*** may become very big and may result in large computation effort and visualization problems. The authors tried to solve that problem by introducing **graph hierarchies**. They allow a *difference map* "to scale to larger graphs, as areas of the graph that the user is not interested in are abstracted away, reducing visual complexity" (Archambault, 2009, p. 1).

Graph hierarchies, which are defined as the recursive grouping of the graph's node, contain so-called *Metanodes*. They hold a subset of nodes and connecting edges. It is important that hierarchies are *path-preserving* which is fulfilled "if any path in a graph hierarchy level corresponds to one or

[2]http://graphml.graphdrawing.org/



Process 2

Process 1

🟩	ask friends	buy	check bud..
buy	0.0	1.0	0.0
check bud...	0.0	0.0	1.0
check flyers	0.7	0.0	0.0
cpu	0.0	0.0	0.0
display	0.0	0.0	0.0
retailer	0.0	0.0	0.0

Figure 2.4: Comparing two graphs describing a buying-workflow in *SGV* (Andrews, Wohlfahrt, and Wurzinger, 2009, p. 2)

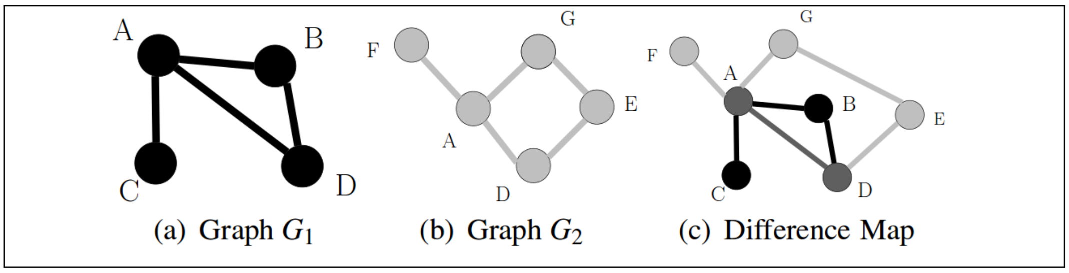


Figure 2.5: Example of two simple graphs (a),(b) and their *difference map* (c) (Archambault, 2009, p. 1)

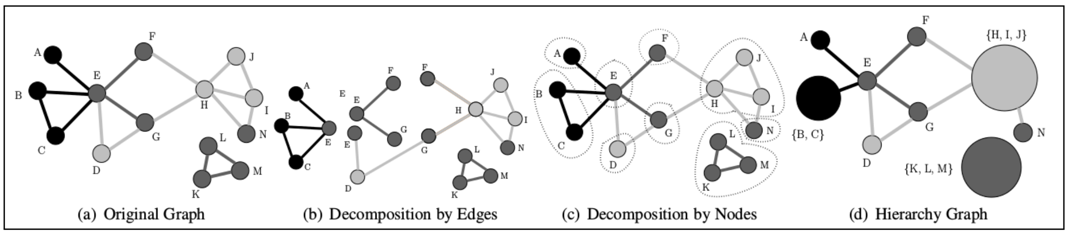


Figure 2.6: Creating hierarchies by applying node- and edges-decompositions on the graph (Archambault, 2009, p. 3)

more paths in the underlying difference map" (Archambault, 2009, p. 1). To create hierarchies, decompositions by nodes and edges are applied to the (combined) graph. This process is illustrated in Figure 2.6.

Finally, a technique called *coarsening* is applied to merge degree-1 nodes (*degree-1 coarsing*) and to combine nodes with low betweenness-centrality-difference within single meta nodes (*betweenness coarsening*) (Archambault, 2009, p. 3). Thus, only major differences in the graph are visualized.

Figure 2.7 shows *difference maps* of a graph with and without hierarchies and coarsening.

2.2.1.4 Summary graph Visualization

One approach to handling multiple graphs, their differences, and similar sub-graphs is to sum them up in *summary graphs* as (Koop, Freire, and Silva, 2013) introduced in their work. They described ideas to merge the data

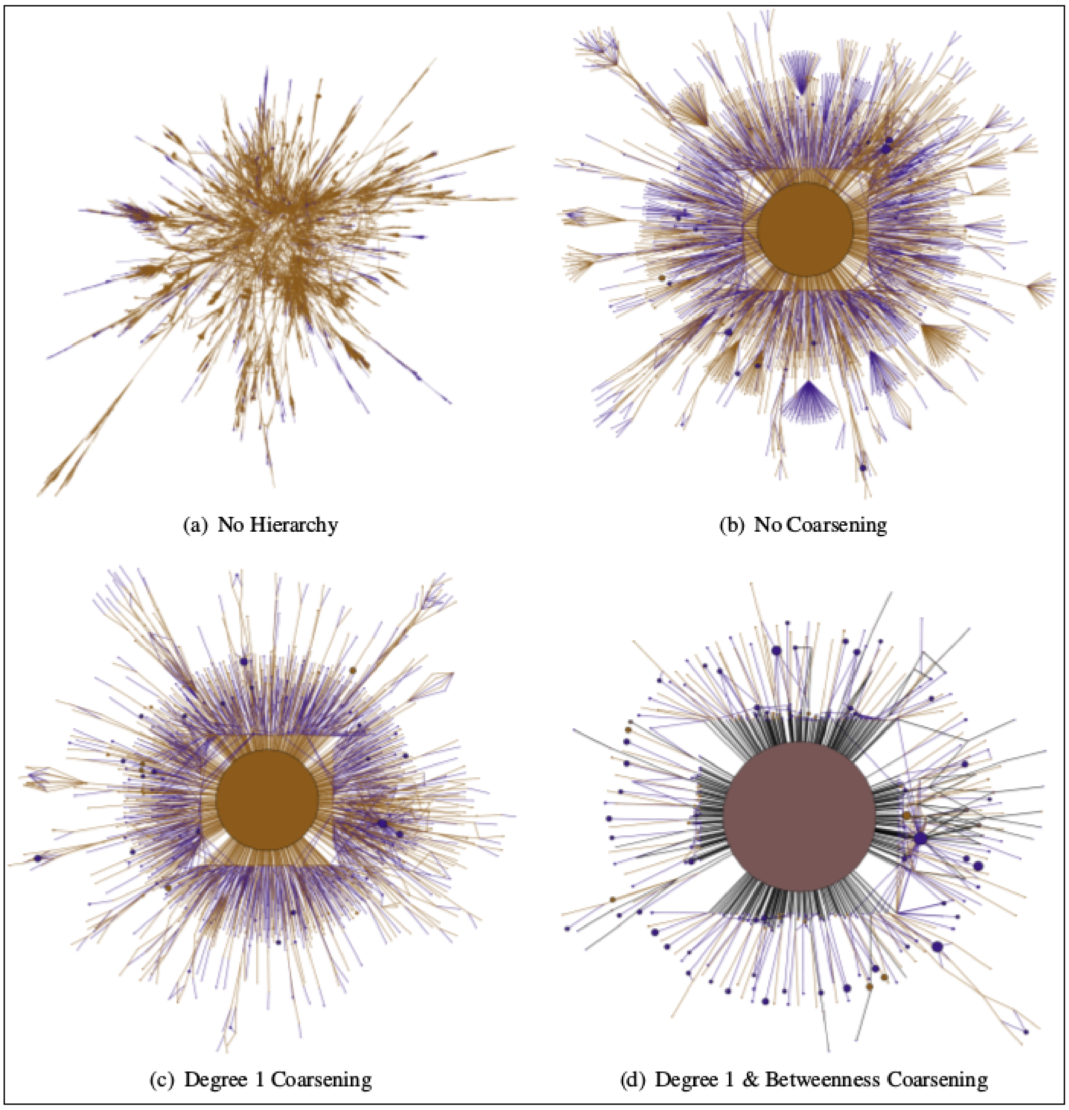


Figure 2.7: *Difference maps* without a hierarchy (a), a hierarchy without coarsening (b), Degree 1 coarsening (c) and Degree 1 coarsening & Betweenness coarsening (d) (Archambault, 2009, p. 7)

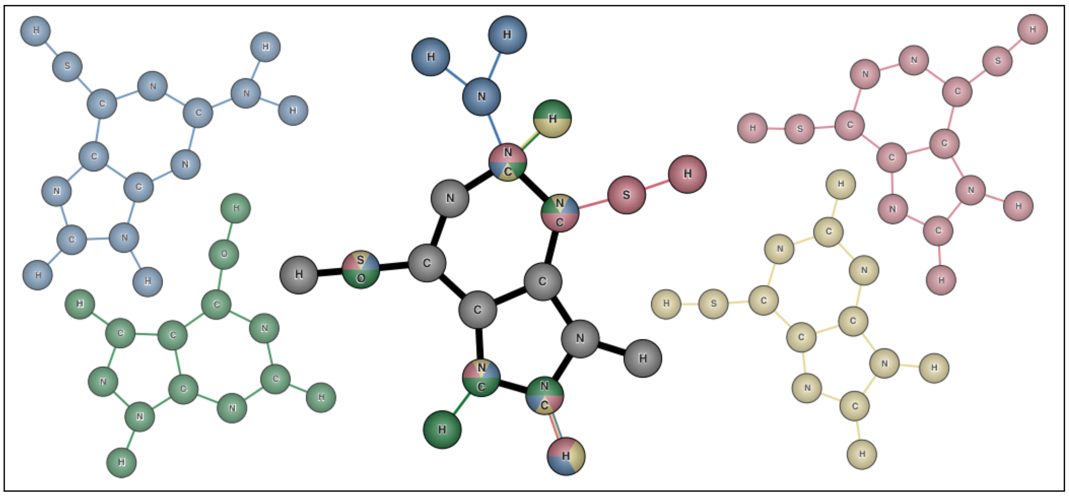


Figure 2.8: A summary graph merging data and structures from four similar graphs (Koop, Freire, and Silva, 2013, p. 3)

of more than two graphs but also tried to preserve the graphs' structures as well as possible. The user should have the possibility to influence the changes of the merged structures by using *break and join operations*. This allows them to bring out structures more evenly, if the corresponding nodes of the graphs do not match perfectly.

Another focus of the authors was to provide good interaction methods for easy exploring and manipulating of the summary graph. A *summary graph* is calculated by computing pair wise graph matching using the *Hungarian Algorithm* combined with *similarity flooding* (Melnik, Garcia-Molina, and Rahm, 2002) to calculate the initial node substitution costs for improving the result's connectivity (Koop, Freire, and Silva, 2013, p. 4).

Figure 2.8 gives an example on how a *summary graph* may look if four similar graphs were merged. To let the user recognize each source graph in a *summary graph*, the authors make use of different colors if a node is not the same on all graphs. The *summary graph* is editable, which means that the user can influence the structure of the graph by *breaking* (splitting) and *joining* nodes. This may be necessary if the nodes match but too much crossing edges messes up the layout, for example.

2.2.2 Comparing Document Collections

One of the major goals of this work is to visualize several document collections: either the search history or bookmark-collections. The user should have the possibility to easily find similarities and differences in those collections. This section will provide a short overview of previous work on visualizing the differences of document collections.

2.2.2.1 Donatien

A very interesting approach can be found in the work by (Hascoët and Dragicevic, 2011), which introduces a tool called *Donatien*. It was implemented to demonstrate mechanisms for the visual comparison of document collections and query histories which parallels the motivations of this work. The tool is meant to fulfill the following needs:

- Combining collections
- Comparing collections
- Allowing interactions

To satisfy all three of these conditions, the authors decided to use different layers for comparison. Each layer holds one collection and a layout is applied. Layouts can be used on more layers to simplify comparing the underlying collections. To make the comparison easier, the user can zoom and pan on each layer. The nodes' positions aren't calculated by optimization based algorithms like force-directed layouts, but are rather defined by so called *node signatures*. A node's signature is a property that may be calculated through semantic data (e.g. the document's content) or through structural information, like its complexity-degree. The latter type of node signatures is equivalent to node invariants. This type of positioning has several benefits:

- The layout is deterministic and predictive: the same set of documents will produce the same layout every time it is calculated.
- The layout is stable: modifications of the data mostly lead to only slight changes of the layout.

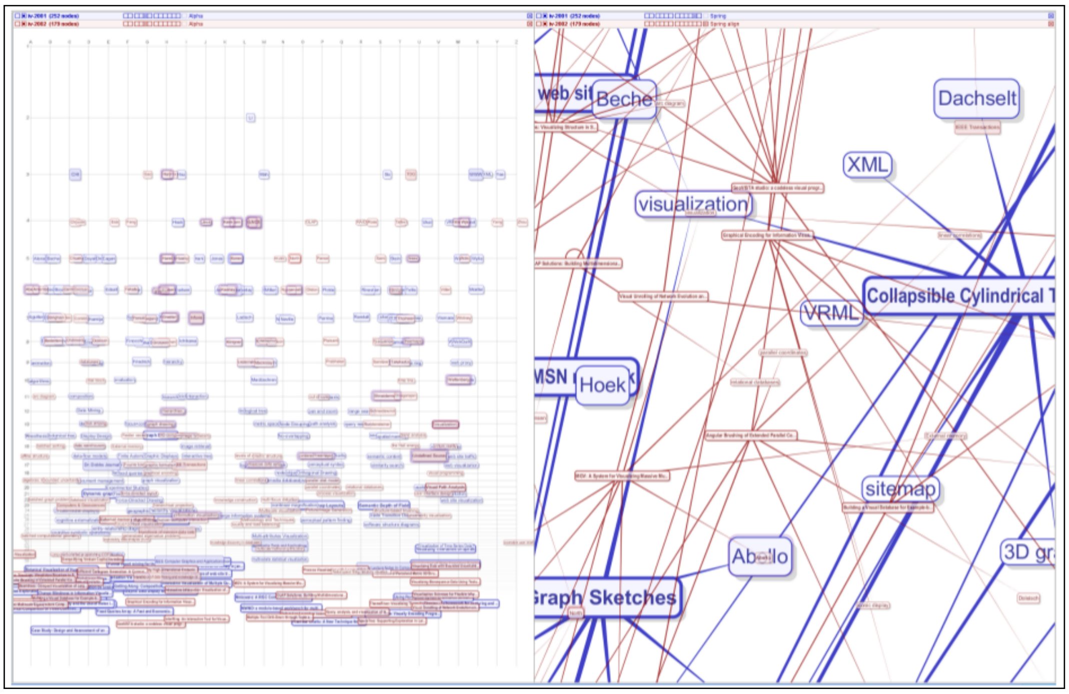


Figure 2.9: Comparing collections via layers in a label based layout (left) and multiple zoom-factors on different layers (right) in *Donatien* (Hascoët and Dragicevic, 2011, p. 9)

- Ideal for superimposed comparing: documents with the same data will have the same position in each layer. That means that if they are the same in both graphs, they share the same positions.

Donatien supports two types of comparisons. As mentioned, users can superimpose the layers. Due to the node signature layout, the same documents share the same position and thus the user can easily identify them as the same. Due to the possibility of interaction, errors can be corrected by drag and drop. Such errors can occur due to little differences in metadata like author, keywords, etc. A sample of superimposed comparing can be seen in Figure 2.9. The second possibility to compare multiple collections is by using a split-window view, where the graphs can be compared side by side.

2.2.3 Time-Oriented Graphs

Sometimes a single set of data changes over time. To visualize those changes is a challenge that also needs to be addressed in this work because different collections of the query history are compared.

2.2.3.1 GraphAEL

One work that deals with changing graphs is (Erten et al., 2003). The authors developed a software written in Java called *GraphAEL* with the goal of visualizing evolving graphs. In particular, they provide an interface to query results from a relational database, which stores documents related together through their citations, topics, etc. When requesting a query, the user can set a specific time-granularity. It results in citation-graphs, topic-graphs, or co-authorship-graphs.

Once the data was collected, the graph can be visualized in *GraphAEL*. The force-directed graph-drawing algorithm that is used to layout the graphs uses a modification of the *GRIP*-algorithm (Gajer and Kobourov, 2001). Each vertex has a timestamp attribute called *timeslice*. Only vertices with the same *timeslice* attribute are connected within one force-directed layout.

Corresponding nodes with different *timeslices* need to be connected to visualize the evolution of the graph. Those connections are made between vertices with the same label on different *timeslices*. It is important to preserve similar structures of the different connected groups to conserve the mental map of the graph's evolution.

The tool makes it possible to select from several visualizations: it is possible to show a static representation of the *timeslices* in 2D or 3D where the connecting edges between the *timeslices* can be shown or hidden. Alternatively, an animated representation can be selected. The interpolated values of edges and vertices of two *timeslices* are animated instead of showing both sets.

To determine small changes within two *timeslices*, a *difference graph* can be shown instead of the classic representation. It only visualizes the differences between the two sets.

Figure 2.10 shows three time slices of a topic-graph visualized with *GraphAEL*.

2.2.3.2 Evolving Discourse Networks

(Brandes and Corman, 2003) also tried to find a good representation of graphs, containing different states at a specific point of time. Although the authors' approach works on every kind of time-evolving graph, they developed their tool to visualize information of different discourses, so that the evolution of one or more persons' discourse is visualized.

Typically, dynamic, animated graph drawing is used for visualizing changes in graphs. This has the advantage that user can see the actual change and his mental map is preserved. The problem is that when viewing a graph at a specific state of time, the applied changes can't be recognized (*change blindness* (Nowell, Hetzler, and Tanasse, 2001)). This is the reason why the authors introduced their layer-based model: they make use of semi-transparent layers, each containing a graph of the current time stamp in a 3D space.

The tool was developed in JAVA using the Data Structures Library (JDSL) and VRML for the 3D visualization. Using this approach has the advantage of seeing changes on the graph at each state. Old states can be seen as semitransparent in the top-view and fade out over time when as layers are attached.

If a layer becomes populated new elements, they can be set on a suitable position depending on the graph-layout, whereas nodes that already existed in the layer remain in their positions. The change of the importance of a node (in the authors model the importance of a discourse's word) is visualized through changing the nodes diameter in the graph.

This approach allows the analysis of the discourses of one, two or more people, where each speaker gets their own color in the graph to help distinguish between the different speeches. If a word is used by more than one person frequently in a discourse, the overlap can be easily seen because the different colored sub-graphs are connected at this node. Figure 2.11 gives an example of a group discussion using such layered visualization.

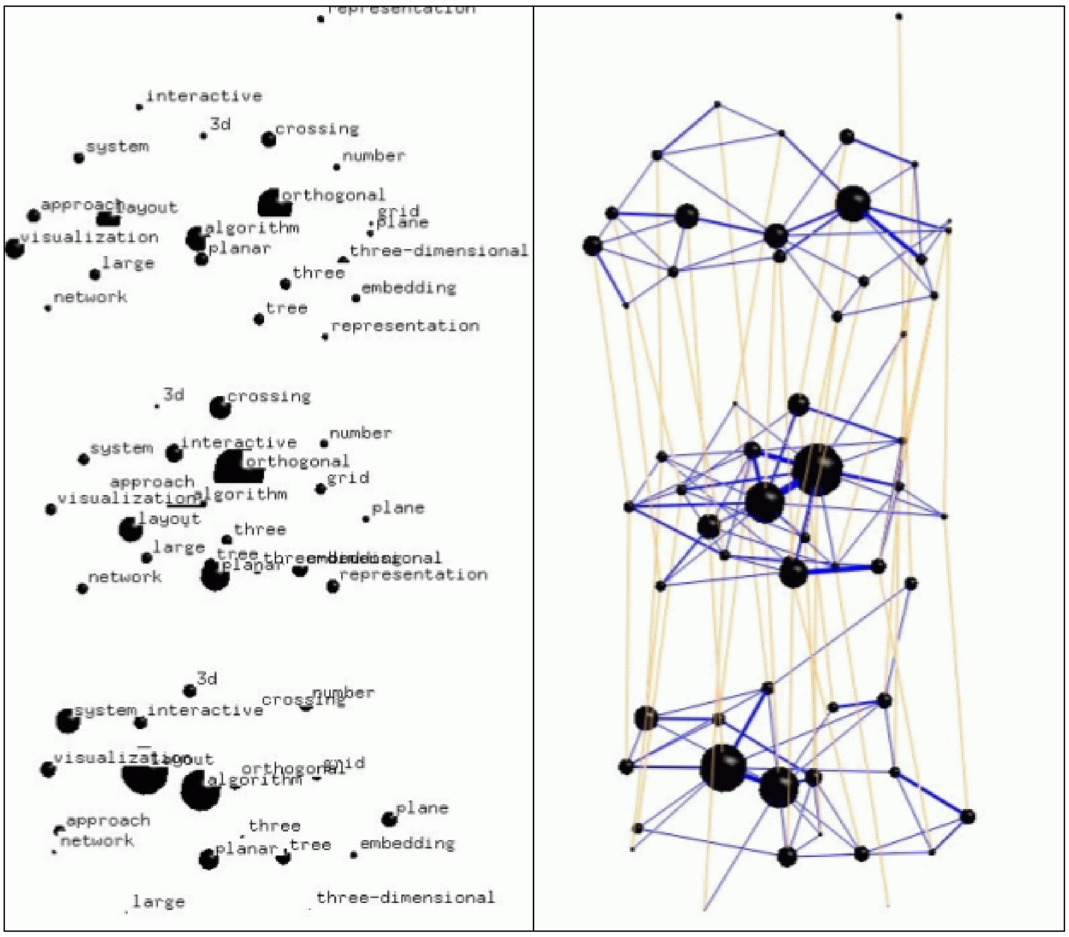


Figure 2.10: Three time slices of a topic-graph visualized with *GraphAEL* (Erten et al., 2003, p. 9)

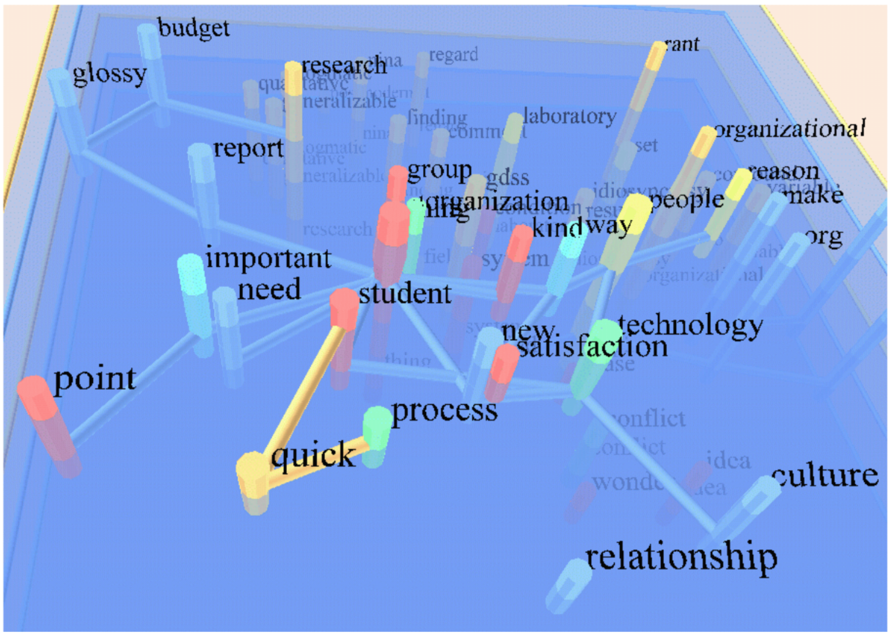


Figure 2.11: A group discussion visualized through layers (Brandes and Corman, 2003, p. 6)

2.2.3.3 GEVOL

Software is typically developed with support of *Concurrent Versions Systems* (CVS), where different authors can add, change, merge etc. shared source code. (Collberg et al., 2003) tried to visualize the evolution of software in a CVS system. They introduced *GEVOL*, a tool to show different representations of a software project's chronological development as a dynamic graph to address the following questions:

- When and where in the project were parts of the software created?
- When did heavy modifications of a software-part occur?
- Which part of the software may be unstable?

The tool is based on *GRIP* which "is designed to quickly layout graphs with tens of thousands of vertices without assuming any information about the underlying graphs" (Collberg et al., 2003, p. 3). There were two essential challenges to manage:

- The drawing must be readable.
- The drawing must preserve the mental map when changing.

Therefore *GRIP* was extended with the support of node- and edge-weights and the support of time slices.

The extracted data from a CVS are collected in time slices of one day and can be represented in several visualizations:

- control flow graph
- inheritance graph
- call graph

Each graph lets the user find out interesting facts about the software. This is sometimes necessary if, for example, a user has to extend, fix or rewrite some software that is unfamiliar, which is badly written, or hardly documented, in the worst case. Through those visualizations, the user can find out facts about the structure, functionality and history of that program. When going through the timeslices, the age of a modification is visualized through different colors: new modifications are shown red and fade to a pale blue over time. So, if a specific part of the graph is red all the time, it could mean that there bugfixes are very frequently committed and the part may have to

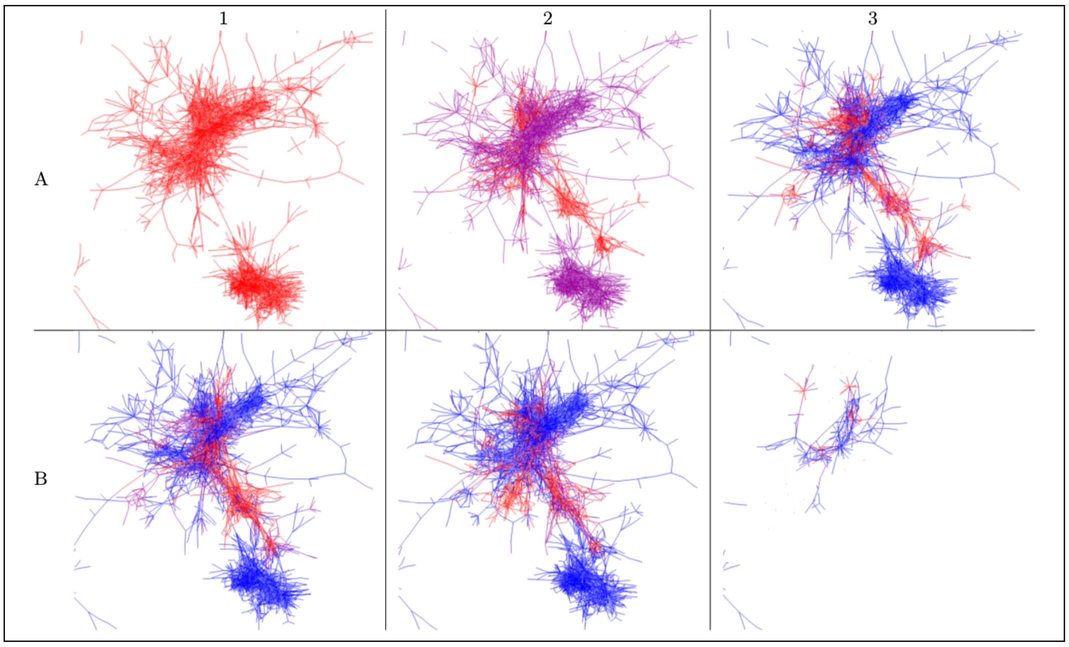


Figure 2.12: Evolution of software in a CVS visualized with *GEVOL*. "Snapshots of the SandMark call-graph. Nodes start out red. As time passes and a node does not change, it turns purple and, finally, blue. When another change is affected the node again becomes red." (Collberg et al., 2003, p. 6)

be rewritten. Figure 2.12 gives an example of a visualization in *GEVOL*. The user can click on an interesting node, and information about the authors is shown.

2.2.3.4 Further Related Work on Time-Oriented Graphs

Further research about changes in dynamic graphs can be found in (Bach, Pietriga, and Fekete, 2014). The authors used animations to help understand changes in dynamic networks. Their tool "*GraphDiaries* relies on animated transitions that highlight changes in the network between time steps, thus helping users identify and understand those changes." (Bach, Pietriga, and Fekete, 2014, p. 1)
(Fu et al., 2007) describes a *timeslice*-based visualization of dynamic graphs by using "a 2.5D visualization for temporal email networks to analyze the

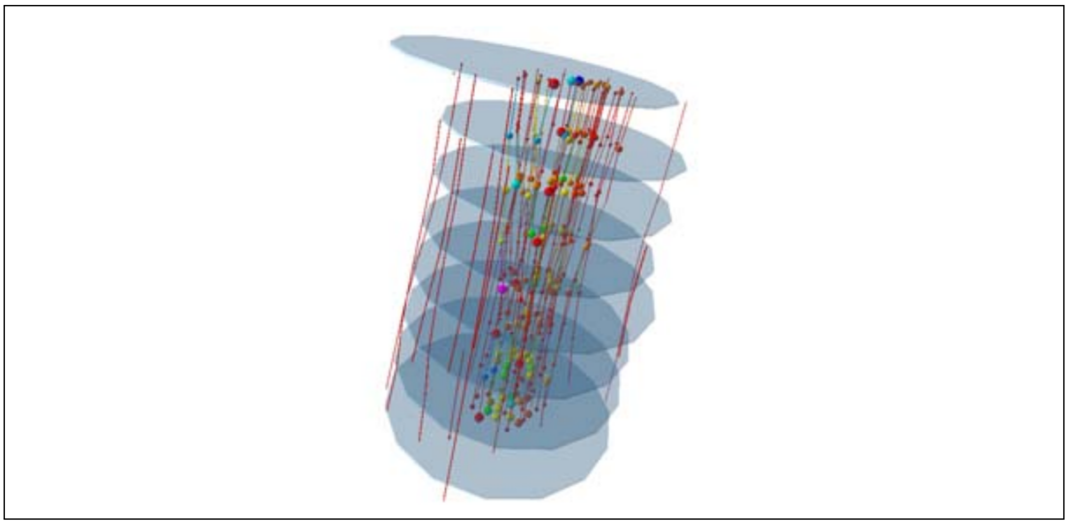


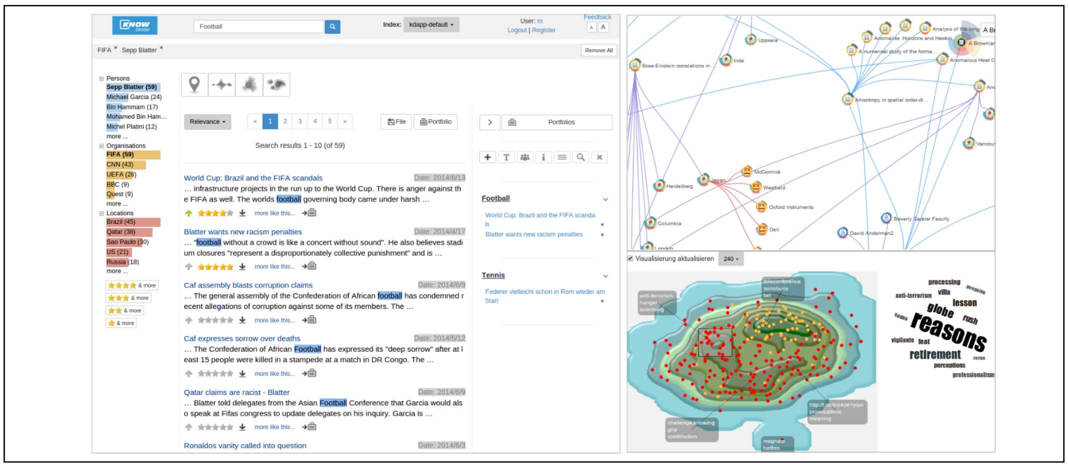
Figure 2.13: *Timeslices* visualizing the evolution of an email-network (Fu et al., 2007, p. 5).

evolution of email relationships changing over time" (Fu et al., 2007, p. 1) (see Figure 2.13).
(Ahn et al., 2011) introduce two prototype applications that propose five principles of implementing temporal visualizations including encoding time into visual properties.

(Wozelka, Kröll, and Sabol, 2015) introduced a "time-oriented graph visualisation approach which maps temporal information to visual properties such as size, transparency and position and, combined with advanced graph navigation features, facilitates the identification and exploration of temporal relationships." (Wozelka, Kröll, and Sabol, 2015, p. 1)

2.3 Further Related Work

The *Knowminer Search* (Rauch et al., 2015) is a search tool that builds up on the *faceted search approach* (Zheng et al., 2013), combined with interactive visualizations and possibilities to organize search results (see Figure 2.14). It shares similar ideas of visualizing results with the *EEXCESS - Recommender*



Search results 1 - 10 (of 59)

World Cup: Brazil and the FIFA scandals Date: 2014/6/13
… infrastructure projects in the run up to the World Cup. There is anger against th e FIFA as well. The worlds football governing body came under harsh …
more like this...

Blatter wants new racism penalties Date: 2014/4/17
… "football without a crowd is like a concert without sound". He also believes stadi um closures "represent a disproportionately collective punishment" and is …
more like this...

Caf assembly blasts corruption claims Date: 2014/6/9
… The general assembly of the Confederation of African football has condemned r ecent allegations of corruption against some of its members. The …
more like this...

Caf expresses sorrow over deaths Date: 2014/5/12
… The Confederation of African Football has expressed its "deep sorrow" after at l east 15 people were killed in a stampede at a match in DR Congo. The …
more like this...

Qatar claims are racist - Blatter Date: 2014/6/9
… Blatter told delegates from the Asian Football Conference that Garcia would als o speak at Fifas congress to update delegates on his inquiry. Garcia is …
more like this...

Ronaldos vanity called into question Date: 2014/6/3

Football
World Cup: Brazil and the FIFA scandals
Blatter wants new racism penalties

Tennis
Federer vielleicht schon in Rom wieder am Start

Figure 2.14: *Knowminer*'s web-based search interface (left), the graph-view on the top-right and an information landscape (bottom-right) ((Rauch et al., 2015, p. 4)).

Dashboard (Section 3.3). Both share a representation of the results through a *Geo-visualization* and an *Information Landscape* visualization (see Section 3.3.2).

3 EEXCESS

This work contributed to the *EEXCESS* (***E**nhancing **E**urope's e**X**change in **C**ultural **E**ducational and **S**cientific Re**S**ources*)[1] recommender system project. Its goal is to support the user during the consumption and creation of content on the web (Granitzer et al., 2013, p. 2) by augmenting the current browser window with **educational, cultural and scientific resources** from the so-called *long-tail*.

This chapter gives an overview of the motivation and background of the *EEXCESS* project, and provides technical details of the software used in Section 3.5.

3.1 Motivation

With the enormous growth of content in the web over the last decades, it has become a big challenge to find and retrieve valuable results. A few big-players have emerged, like search engines or social-networks that support finding and retrieving of relevant content. It is particularly difficult to find scientific or cultural documents, as the results are often lost in the so-called **long-tail**. The long-tail which is a synonym for the mass of potentially relevant results that is rarely retrieved in searches, for instance, because they are not at the head of the demand (not popular or out of the mainstream for example) (Anderson, 2013).

The goal of *EEXCESS* was to bring these long-tail results with cultural, educational or scientific context to the user, by creating an interface that uses the current content of a browser window to extract information from

[1] http://www.eexcess.eu/

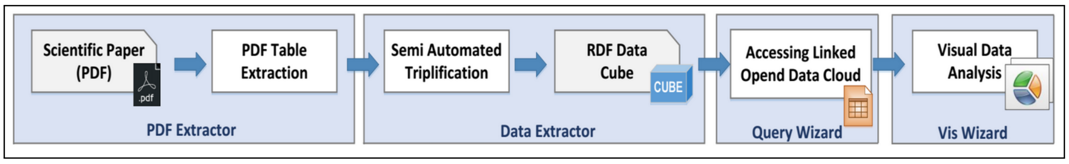


Figure 3.1: Schema of the *CODE* pipeline for analyzing and the extracting of scientific data embedded in publications (Mutlu and Sabol, 2015, p. 2)

it, infer the user's interests and use this information as input for an RS that retrieves the corresponding results.

Even if documents can be found by users, another challenge is it to extract data from the content. An approach to support the user in both extracting and visualizing this data automatically is the *CODE Visualisation-Wizard* which was designed as "a tool to automatically extract data from scientific publications and propose the appropriate means to visualise the facts and data therein" (Mutlu, Hoefler, et al., 2014, p. 1). It helps the user in fulfilling the following tasks (Tschinkel, Veas, et al., 2014, p. 1):

- selecting and configuring the visualizations
- aggregating datasets
- brushing and linking over multiple datasets

As Figure 3.1 illustrates, the *Visualisation-Wizard* is the last one of several steps in a pipeline of the *CODE* project.

Similar ideas can also be found in the objectives of the *EEXCESS Recommendation Dashboard* (see Section 3.3).

3.2 Basic Challenges

Due to the specific focus of the *EEXCESS* project, it was necessary to address several research-based challenges (Granitzer et al., 2013, p. 2):

- The system should work both when the user **consumes data**, like surfing the web but also if new **content is created**, like writing blog-posts or Wikipedia articles (See Section 3.4.2).

- It is necessary to implement *EEXCESS* within an thoughtful **user interface** design to ensure a high rate of user acceptance.
- Every time a recommendation request is initiated, data about the user's current task is sent to a server. This fact means that **privacy considerations** arise. Consequently, only a minimum of necessary user data must be sent to the servers by interpreting the content on the client locally as far as possible. The trade-off between privacy and performance is mentioned as one of the largest challenges.
- To reduce immense costs by developing one large-scale RS it was intended to build an **RS-network** instead. Each instance of this network should provide specific content to a special user group. This network was realized through a handful of **external providers**: "The recommender retrieves results from various data sources, e.g. *Europeana* for cultural heritage resources or *Mendeley* for scientific papers." (Tschinkel, Sciascio, et al., 2015, p. 2) Further providers were included like the *Deutsche Zentralbibliothek für Wirtschaftswissenschaften* (ZBW)[2].
- The communication with these providers is managed by the **Federated Recommender**. It acts as a service to send queries to the different providers through their APIs and to collect, harmonize and fill the received results with metadata before passing them back to the client. Figure 3.2 illustrates its functionality (also see Section 3.5.2 for a technical discussion).

3.3 Recommendation Dashboard

One of the core-features of *EEXCESS* is the **Recommendation Dashboard** (**RD**) (Figure 3.3). Due to the fact that the user hardly has control over the search of the RS itself, getting an overview over the received results and doing individual inspections of the documents may be very challenging for the end-user (Tschinkel, Sciascio, et al., 2015).

The *RD* was developed to avoid these problems by augmenting user interfaces with visualizations and assisting users in analysis and exploration of the recommendation space (Veas et al., 2015, p. 1). The dashboard provides

[2]http://www.zbw.eu/

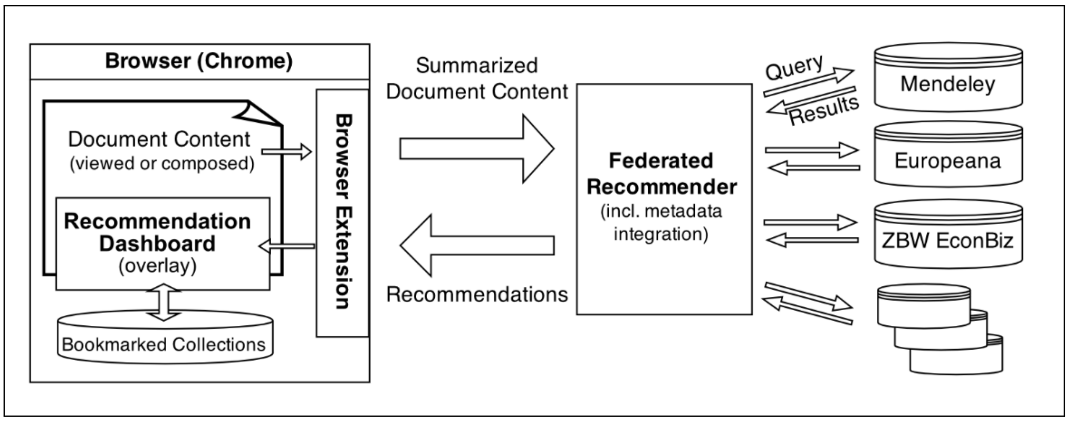


Figure 3.2: Schema describing the federated recommender's functionality (Tschinkel, Sciascio, et al., 2015, p. 2)

different visualization that take advantage of the human visual system's capabilities to deal with a lot of data simultaneously, and recognize patterns within (Tschinkel, Sciascio, et al., 2015, p. 2).

3.3.1 Functionality Of The RD

The dashboard allows to visualize **result-items** or **recommendations** holding metadata of their documents. These items can either be the result of an *EEXCESS* query (**query-collection**) or a saved **bookmark-collection** (see Section 4.2.6). Depending on the user's needs and the content of the items, a visualization can be chosen to represent the items' data.

3.3.1.1 Brushing

Since the user may be interested in finding and visualizing subsets of the items, ***brushing*** (Becker and Cleveland, 1987) can be used. Depending on the chosen visualization, a brush defines a subset of the whole dataset by using interactive methods like drawing a rectangular field on the *Geo-View*-visualization's map or by selecting a bar in the *Bar-Chart*-visualization (Tschinkel, Hafner, et al., 2016). After setting a brush all not matching items

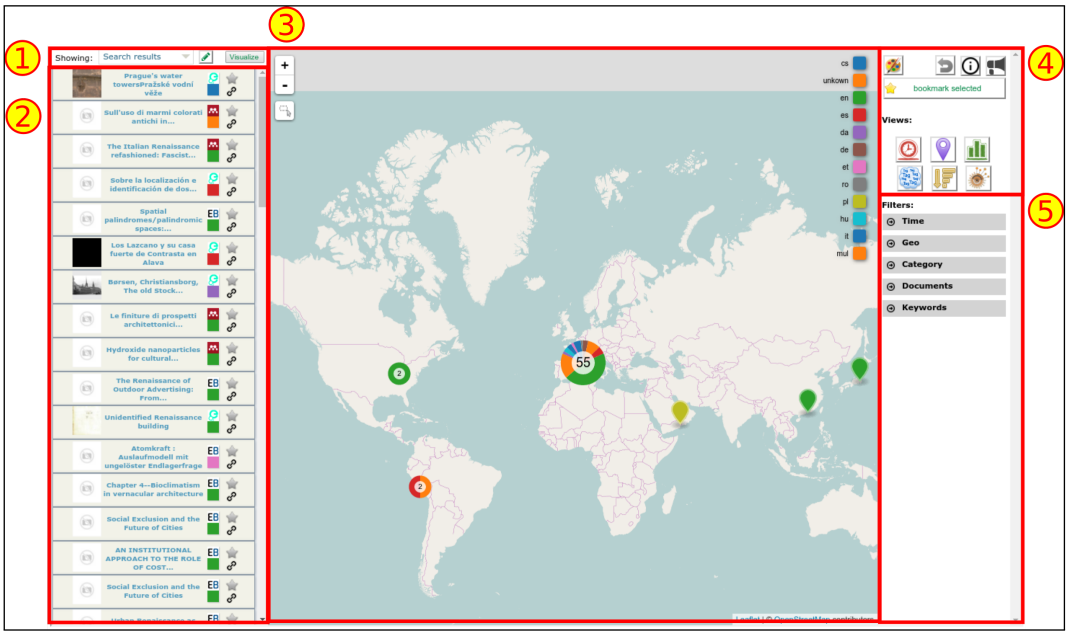

1
2
3
4
5
Showing:
Search results
Visualize
Prague's water towersPražské vodní věže
Sull'uso di marmi colorati antichi in...
The Italian Renaissance refashioned: Fascist...
Sobre la localización e identificación de dos...
Spatial palindromes/palindromic spaces:...
Los Lazcano y su casa fuerte de Contrasta en Alava
Børsen, Christiansborg, The old Stock...
Le finiture di prospetti architettonici...
Hydroxide nanoparticles for cultural...
The Renaissance of Outdoor Advertising: From...
Unidentified Renaissance building
Atomkraft : Auslaufmodell mit ungelöster Endlagerfrage
Chapter 4--Bioclimatism in vernacular architecture
Social Exclusion and the Future of Cities
AN INSTITUTIONAL APPROACH TO THE ROLE OF COST...
Social Exclusion and the Future of Cities
EB
+
-
55
2
2
cs
unkown
en
es
da
de
et
ro
pl
hu
it
mul
bookmark selected
Views:
Filters:
Time
Geo
Category
Documents
Keywords

Figure 3.3: *EEXCESS-Recommendation Dashboard*: On the top left (1) the user can choose between the current search results and bookmarked collections. In the left panel (2) the actual resources are listed. In the middle (3) the current visualization is shown. The top right corner (4) provides possibilities to change settings, to bookmark items and to change the current visualization. (5) The bottom right panel holds the active *Micro-Visualizations* (see Section 3.3.3). (Tschinkel, Sciascio, et al., 2015)

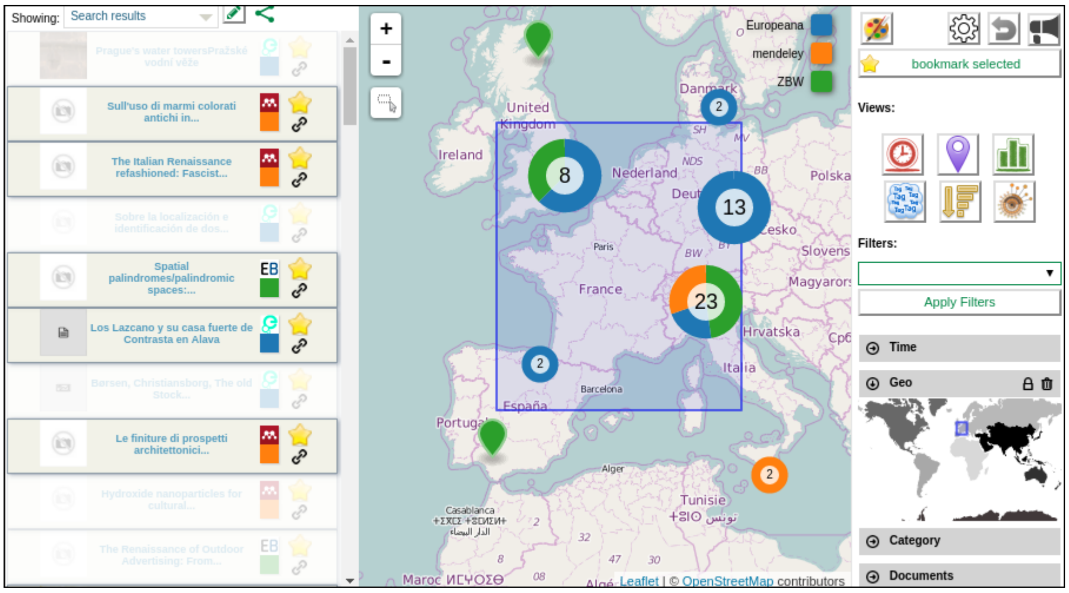

Showing: Search results
Sull'uso di marmi colorati antichi in...
The Italian Renaissance refashioned: Fascist...
Spatial palindromes/palindromic spaces:...
Los Lazcano y su casa fuerte de Contrasta en Alava
Le finiture di prospetti architettonici...
Europeana
mendeley
ZBW
bookmark selected
Views:
Filters:
Apply Filters
Time
Geo
Category
Documents
Leaflet | © OpenStreetMap contributors

Figure 3.4: *Brushing* items in the *Geo-View*-visualization: A shape is limiting the results inside the brush. Results outside the brush are greyed-out in the item list on the left side. The current brush is also shown as a *Micro-Visualization* on the bottom-right side and can be applied there as a permanent filter by clicking on the lock-icon.

are greyed-out in the list and in the current visualization to help the user recognize the chosen subset (see Figure 3.4).

3.3.1.2 Filtering

The user may want to remove irrelevant items from the current query- or bookmark-collection. Therefore a brush can be applied as a (permanent) filter by clicking the lock-icon of the corresponding *Micro-Visualization* (see Section 3.3.3) (Tschinkel, Hafner, et al., 2016). This leads to a removal of all items that do not match the current brush. If the user changes the visualization afterwards, only the filtered items are displayed. It is further possible to apply multiple filters by brushing again on any visualization. All applied filters are permanently visible in the filter-area. They can be each removed by clicking the trash-icon.

3.3.1.3 Bookmarking

An *EEXCESS* query-collection may contain interesting items that the user want to keep. Therefore the *RD* provides a bookmarking functionality. The current displayed - and maybe filtered - items (which can result from a query or an already loaded bookmark-collection) can be added to an existing collection or a completely new one can be created. Bookmark-collections are stored in the user's browser storage, which means they can be restored after closing the browser window. It is possible to select a bookmark-collection by using the dropdown in the top-left corner of the *RD* (Figure 3.3 (1)). A set of bookmark-collections also can be visualized in *ECHO* (see Section 4.2.6).

3.3.1.4 User Interface

The dashboard is visually structured as follows (see Figure 3.3)

- In the top left corner (1) the user can choose between stored bookmark-collections and current search result to display.
- On the left side (2), all results of the current query- or bookmark-collection are listed showing the result's title, a preview image, the provider, the language, a link to the resource, and a bookmark-icon. Clicking on this icon opens a bookmark-form where the user can add that item to an existing or new bookmark-collection.
- The main-view (3) is shown in the middle where the selected visualization is rendered.
- In the top-right corner (4), the control panel can be found, where the user can choose between the available visualizations, can set different options and can bookmark the current items.
- In the bottom-right corner (5), the currently applied filters are shown in the filter-area as *Micro-Visualizations*. If a an active brush exists, or a filter is applied, it is visualized as a *Micro-Visualization* (Section 3.3.3) in the corresponding filter-box. Here the user has the possibility to delete the filter or to make a brush permanent as a filter.

3.3.2 Visualizations

The current version of the *RD* supports five different visualizations. Each of those has a different way of showing information from the results by making use of the result-items' metadata: the results contain a title, description, and a thumbnail, but also geo-references, timestamps, the language, the name of the data-provider, mime-type, and the underlying license of the document (Tschinkel, Sciascio, et al., 2015, p. 2) The latter are summed up as the results' *facets* (for a detailed list, see Section 5.6.1). All of the visualizations described below are realized within the plugin system (see Section 3.5.3.2) of the *RD*.

3.3.2.1 Timeline

The **Timeline** visualization makes use of the results' timestamps (x-axis) to visualize recommendations, their timestamps and categorical metadata (provider or language) using the y-axis and a color-mapping. It is possible to switch between those two facets using the *color-mapping* button. The different values of the selected facet type can be distinguished by different color representations. The other facet is described by a result's position on the y-axis. Figure 3.5 gives an example of a timeline-visualization with the providers encoded using colors and languages positioned along the y-axis. At the bottom of this view it is possible to define a time-range and apply it as a brush.

3.3.2.2 GeoView

Many results contain geographical information, represented by coordinates. This makes it possible to visualize those results on a map using the **GeoView** visualization. As with the timeline visualization, the user again has the possibility to choose the categorical facet that describes the results via different colors on the map.

This view can also deal with rendering many results in a small area on the map: "When the density of single data items on screen becomes too

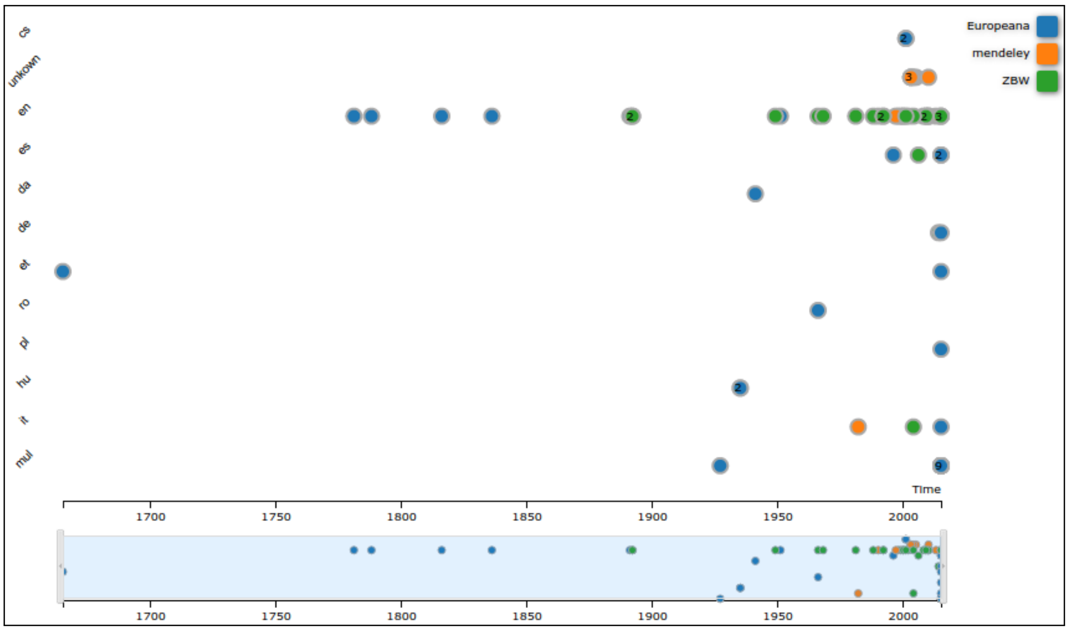


Figure 3.5: Visualizing the results' providers on a timeline in the *EEXCESS-RD*.

high for displaying them separately they will be aggregated into a single symbol – a "donut chart" – representing the distribution of the categorical metadata and displaying the number of the underlying items in the centre." (Tschinkel, Sciascio, et al., 2015, p. 3)

Figure 3.6 shows results in the GeoView with the data-provider mapped onto color. If the user wants to limit the result to a specific region, a rectangular brush tool can be used to select resources depending on their geo-reference metadata.

3.3.2.3 Bar-Chart

Another possibility for visualizing the results is to use the **Bar-Chart**. It renders all values of the selected facet on the x-axis and the number of their occurrences on the y-axis. Clicking on a bar applies this value as a brush. The Bar-Chart only supports the categorical filters *provider* and *language*. One of those facets can be chosen to be shown on the x-axis, the other one is

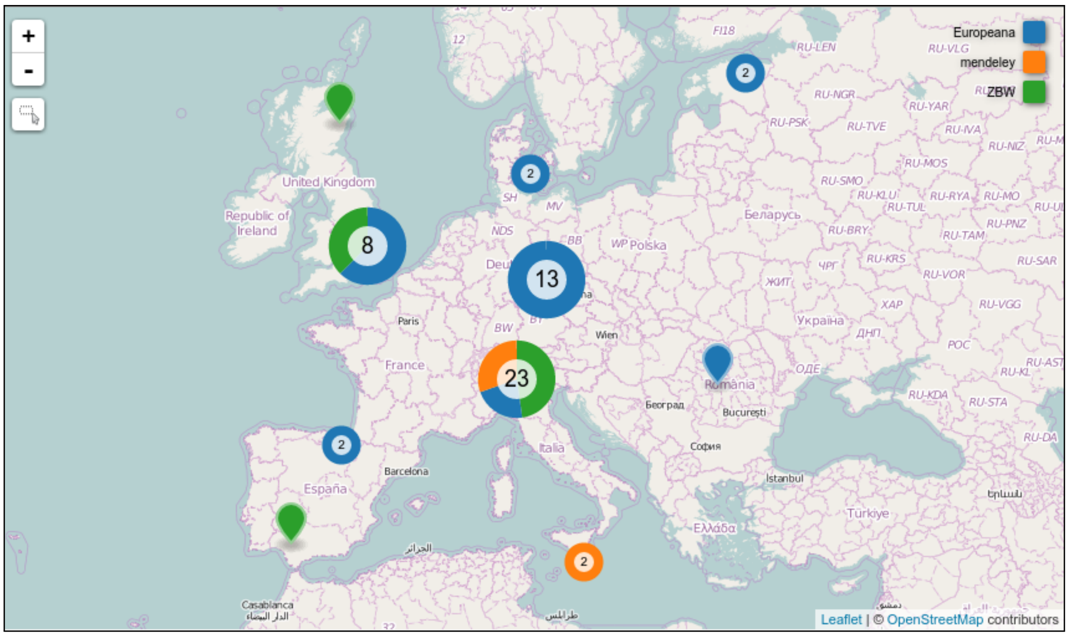


Figure 3.6: Results represented via their geographical information on a map with different colors, encoding the data-providers.

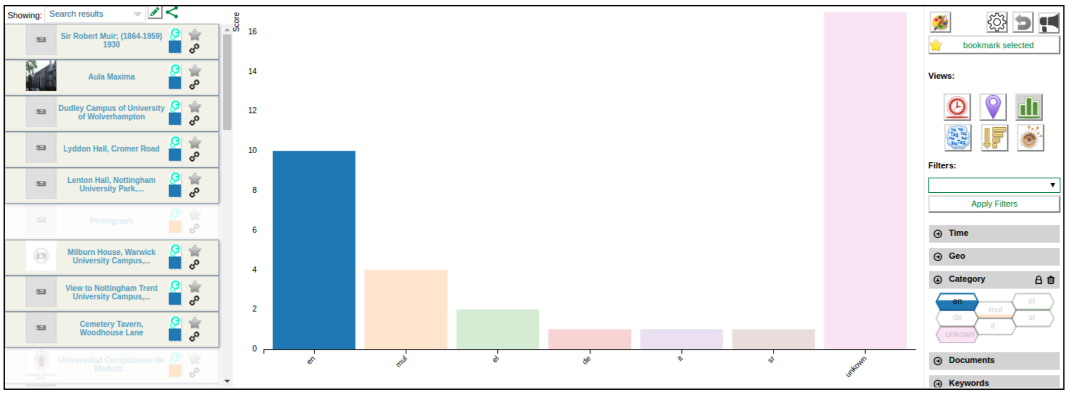

Showing: Search results
Sir Robert Muir; (1864-1959) 1930
Aula Maxima
Dudley Campus of University of Wolverhampton
Lyddon Hall, Cromer Road
Lenton Hall, Nottingham University Park,...
Photograph
Milburn House, Warwick University Campus,...
View to Nottingham Trent University Campus,...
Cemetery Tavern, Woodhouse Lane
Universidad Complutense de Madrid,...
Score
16
14
12
10
8
6
4
2
0
en
mul
el
de
it
sr
unkown
bookmark selected
Views:
Filters:
Apply Filters
Time
Geo
Category
en
el
mul
de
sr
it
unkown
Documents
Keywords

Figure 3.7: A Bar-Chart with an applied brush on a value of the language-facet, which is represented on the x-axis.

represented through a color. Figure 3.7 shows a Bar-Chart with an applied brush on a specific language-facet value. It also shows that the Bar-Chart's categorical *Micro-Visualization* is represented through hexagons (Tschinkel, Hafner, et al., 2016).

3.3.2.4 Information Landscape

The **Information Landscape visualization** provides an approach for visualizing an overview of dominant topical clusters, relationship between them and for identifying topical outliers, using a geographic map metaphor. It renders a topical landscape model and computes the x/y-position of the results, depending on their extracted keywords using *force-directed placement* (Fruchterman and Reingold, 1991). As Figure 3.8 shows, this leads to an easily recognizable representation of the resulting clustering of the different documents. Results with similar keywords are positioned closer than those with completely different keywords.

The Information Landscape makes use of metaphors based on map reading to enable visual thinking in the exploratory phase. (Ulbrich et al., 2015). "An information landscape conveys similarity of topics in a data set through spatial proximity and density of topics as elevation in the visualization. Hills represent groups (clusters) of topically related documents separated

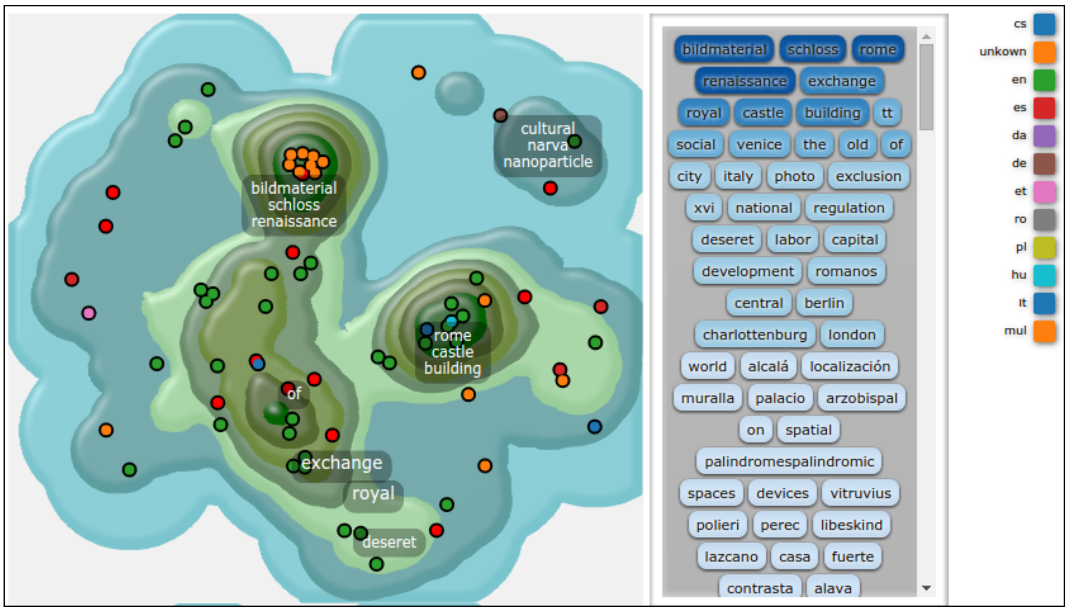


Figure 3.8: Landscape-visualization of the RS's results in the *RD*

by areas represented as sea. Higher hills group more documents than lower ones." (Ulbrich et al., 2015, p. 3)

The user has the possibility to perform zoom and pan operations on the map to navigate. On the right side of the landscape a tag-cloud can be found holding the sorted keywords of the results. Hovering a keyword leads to highlighting the matching items on the map. Brushing elements in the Information Landscape works by clicking on one or multiple keywords which hence appear in the keyword-*Micro-Visualization*.

3.3.2.5 uRank

An extensive visualization included in the *RD* is **uRank**. The plugin provides "interactive methods for understanding, refining and reorganizing documents on-the-fly as information needs evolve" (Sciascio, Sabol, and Veas, 2015, p. 1).

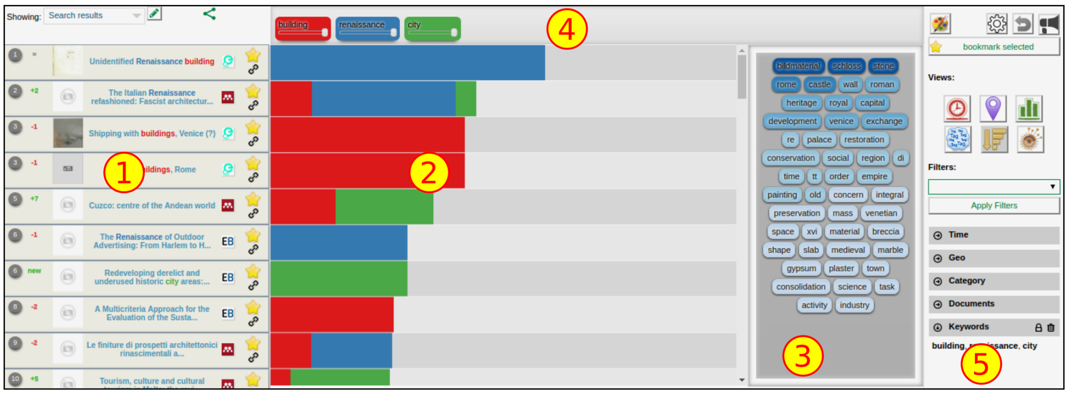

Showing: Search results
building
renaissance
city
Unidentified Renaissance building
The Italian Renaissance refashioned: Fascist architectur...
Shipping with buildings, Venice (?)
...ildings, Rome
Cuzco: centre of the Andean world
The Renaissance of Outdoor Advertising: From Harlem to H...
Redeveloping derelict and underused historic city areas:...
A Multicriteria Approach for the Evaluation of the Susta...
Le finiture di prospetti architettonici rinascimentali a...
Tourism, culture and cultural
bookmark selected
Views:
Filters:
Apply Filters
Time
Geo
Category
Documents
Keywords
building, ...ssance, city
1
2
3
4
5

Figure 3.9: *uRank* user interface integrated int to *RD*. The left side of the RD shows the (re-ranked) recommender results (1) and the left side of the visualization-area (2) shows the *Ranking View*. They both "present a list with augmented document titles and stacked bars indicating relevance scores" (Sciascio, Sabol, and Veas, 2016, p. 4). The *Tag-Box* (3) holds the extracted and ranked keywords of the results. The *Query Box* (4) holds the keywords selected by the user. The keyword-*Micro-Visualization* (5) shows the current brush and gives the user a possibility to apply the selected keywords as a filter on the current items

It is an interactive user-driven tool that supports exploration of textual document recommendations through:

- an automatically generated overview of the document collection depicted as keyword tags
- a drag-and-drop-based mechanism for refining search interests
- a transparent stacked-bar representation to convey document ranking and scores, plus the query term contribution

Figure 3.9 shows the current implementation of *uRank* inside the *RD*.

The keywords are extracted from the title and descriptions of the retrieved recommendations using a combination of *part-of-speech tagging* (Brill, 1992), the *Porter Stemmer method* (Porter, 2006) and TF-IDF (*term frequency - inverse document frequency*) (Salton and McGill, 1986). The keywords provide a topical overview of the result set and also makes it possible to refine the ranking (Sciascio, Sabol, and Veas, 2015, p. 2). To manipulate the list, the user can make use of the *tag-box* which can be found on the top. Possible manipulations are the additions of new keywords, their removal, or the change of weight. The latter can be done by using the built in weight slider of each keyword. Adding keywords works through a simple drag-and-drop mechanism. Hovering over the keywords highlights the corresponding documents. Next to the keyword-list, the *ranking visualization* can be found. It is divided into a list of document titles and matching keywords, represented as stacked bars, which visualize the importance of the selected keywords to the corresponding recommendations.

3.3.3 Micro-Visualizations

The *RD* further provides so called *Micro-Visualizations* (Tschinkel, Hafner, et al., 2016). Since the user has the possibility to filter results on each visualization through brushing, the *Micro-Visualizations* give an overview of the current set filters on the right side of the *RD*.

"The visual design is data type-specific to convey the filter information in an adequate, natural manner. As we focus on reducing the user load and minimising the use of screen area, the design includes just enough information to represent the filters." (Tschinkel, Hafner, et al., 2016, pp. 4-5) Figure 3.10 provides a sample of different *Micro-Visualizations*. It can be seen that only the *Keywords* provides the lock-icon due to already applied filters on the other visualizations.

3.3.4 VizRec

The user has the possibility to visualize the *EEXCESS* results with any of the visualizations above. The *RD* itself does not support the user in finding the most meaningful visualization depending on the underlying data. Therefore a visualization recommender called *VizRec* was integrated in the *RD*, that suggests the best visualization for the given data. *VizRec* has two ways to recommend proper visualizations (Mutlu et al., 2015a, p. 2):

- It automatically identifies the set of appropriate visualizations using a rule-based algorithm to analyze compatibility between visuals and input data.

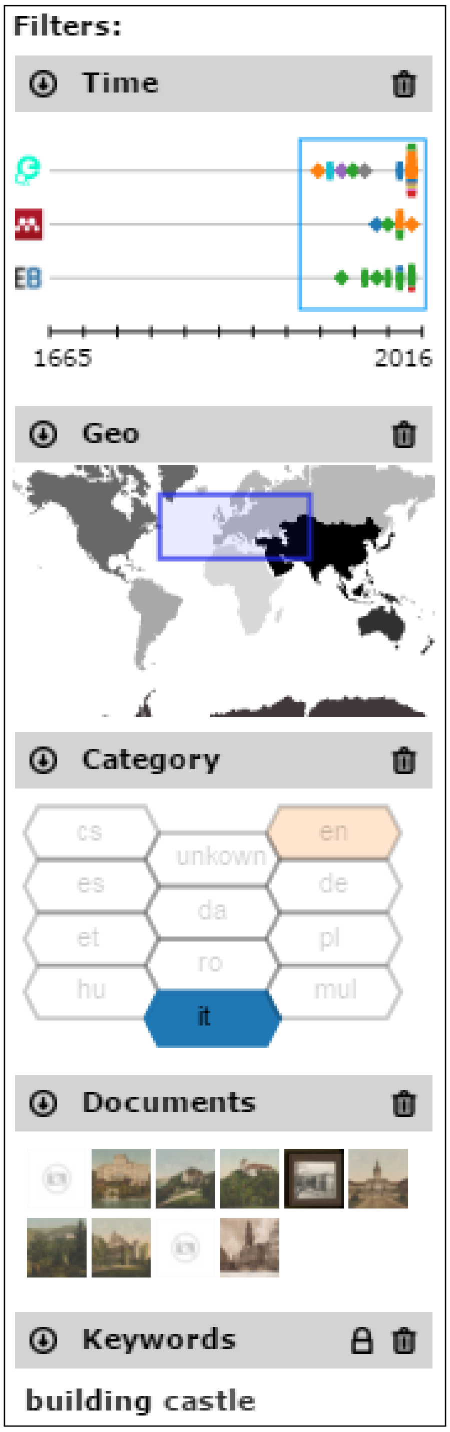


Figure 3.10: The different *Micro-Visualizations* of the *RD*

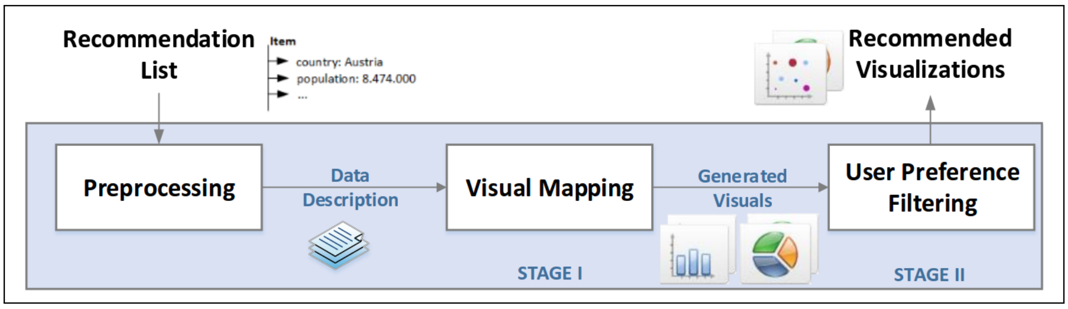


Figure 3.11: *VizRec* workflow for recommending visualizations (Mutlu et al., 2015b, p. 3)

- It filters and ranks a subset based on user's preferences to be recommended as the list of top-n visualizations that best reflect the user's information needs.

Figure 3.11 illustrates *VizRec*'s general workflow (Mutlu et al., 2015b, p. 3).

If the *VizRec* feature, which can be activated optionally, is turned on, the received *EEXCESS* recommender results and a unique User-ID are sent to a *VizRec* server. Depending on the user's preferences and the received data *VizRec* replies with a ranked list of visualizations. Each recommended visualization also holds the most relevant mapping (a Bar-Chart with the providers on the x-axis for example).

3.4 Frontend Implementations

The development of *EEXCESS* has resulted in several prototypes over the last few years that can be used [3]. As mentioned above, it is necessary to distinguish between the usage of *EEXCESS* while reading content on the web, and while writing content.

[3]https://github.com/eexcess

3.4.1 While Reading Content

Using the *EEXCESS* **Chrome-Extension**, the user retrieves recommended documents while surfing the web. With the current version installed via *Chrome Web Store*, keywords are extracted from the content the user is currently reading. This not only means that the current website is analyzed, but also that the extension tries to identify the paragraph that is most likely being read at the moment in order to get the best fitting keywords.

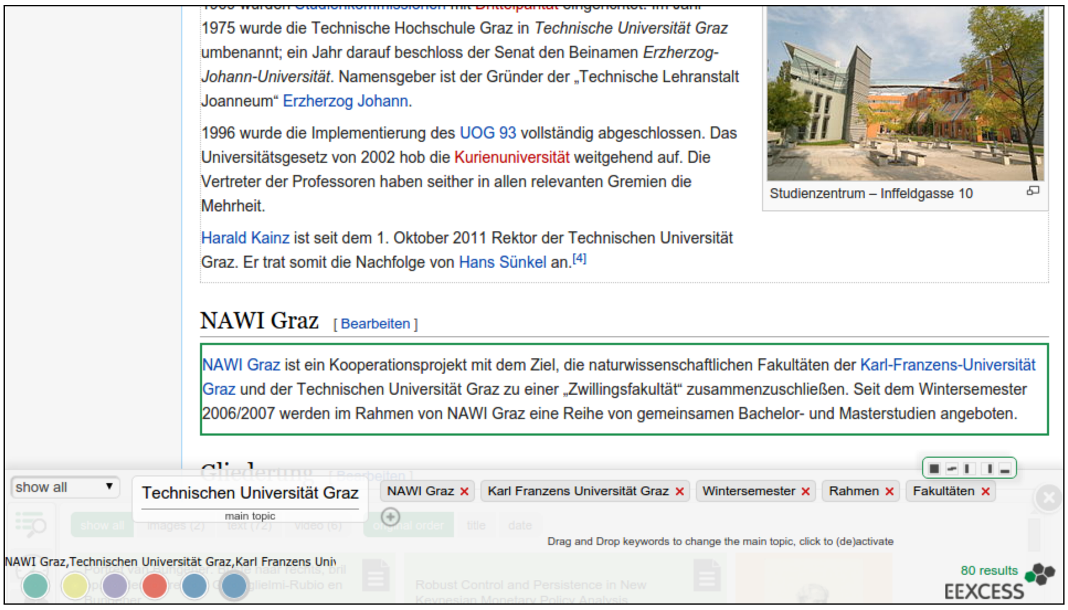
The keywords extracted are shown in a discreet bar at the bottom of the page, where they can be removed or set as the *main-topic* by dragging one keyword into a box. Every change in the keywords starts a new recommendation request on the *Federated Recommender*. While waiting for the results after requesting, the user gets visual feedback that the search is underway via the animation of the *EEXCESS*-logo at the injected *EEXCESS*-bar at the bottom of the page. After retrieving results, the user can open the extension's result list where the documents are listed in a grid view. The user now has the option to open the *RD*.

Figure 3.12 shows the *EEXCESS* Chrome-Plugin while browsing a *Wikipedia* article. The current paragraph the user is focusing on is detected and is highlighted by a green frame. At the bottom, the extracted keywords are listed with *Technische Universität Graz* manually set as *main-topic*.

3.4.2 While Creating Content

As mentioned above, one of the goals of *EEXCESS* was not only to provide results when reading content but also to allow their injection while creating content. Therefore plugins for several browser-based systems were created, that allow the user to enrich content during writing, by getting additional cultural, educational or scientific resources from *EEXCESS* providers for use as reference in their written content.

Plugins are currently available for **Wordpress**, **Moodle** and **GoogleDocs**. An example of use with the *GoogleDocs* Plugin can be seen in Figure 3.13.


1975 wurde die Technische Hochschule Graz in *Technische Universität Graz* umbenannt; ein Jahr darauf beschloss der Senat den Beinamen *Erzherzog-Johann-Universität*. Namensgeber ist der Gründer der „Technische Lehranstalt Joanneum" Erzherzog Johann.

1996 wurde die Implementierung des UOG 93 vollständig abgeschlossen. Das Universitätsgesetz von 2002 hob die Kurienuniversität weitgehend auf. Die Vertreter der Professoren haben seither in allen relevanten Gremien die Mehrheit.

Harald Kainz ist seit dem 1. Oktober 2011 Rektor der Technischen Universität Graz. Er trat somit die Nachfolge von Hans Sünkel an.[4]

Studienzentrum – Inffeldgasse 10

NAWI Graz [Bearbeiten]

NAWI Graz ist ein Kooperationsprojekt mit dem Ziel, die naturwissenschaftlichen Fakultäten der Karl-Franzens-Universität Graz und der Technischen Universität Graz zu einer „Zwillingsfakultät" zusammenzuschließen. Seit dem Wintersemester 2006/2007 werden im Rahmen von NAWI Graz eine Reihe von gemeinsamen Bachelor- und Masterstudien angeboten.

show all
Technischen Universität Graz
main topic
NAWI Graz ×
Karl Franzens Universität Graz ×
Wintersemester ×
Rahmen ×
Fakultäten ×
Drag and Drop keywords to change the main topic, click to (de)activate
NAWI Graz,Technischen Universität Graz,Karl Franzens Univ
80 results
EEXCESS

Figure 3.12: Screenshot of the *EEXCESS*-Chrome-Plugin used while browsing a *Wikipedia* article

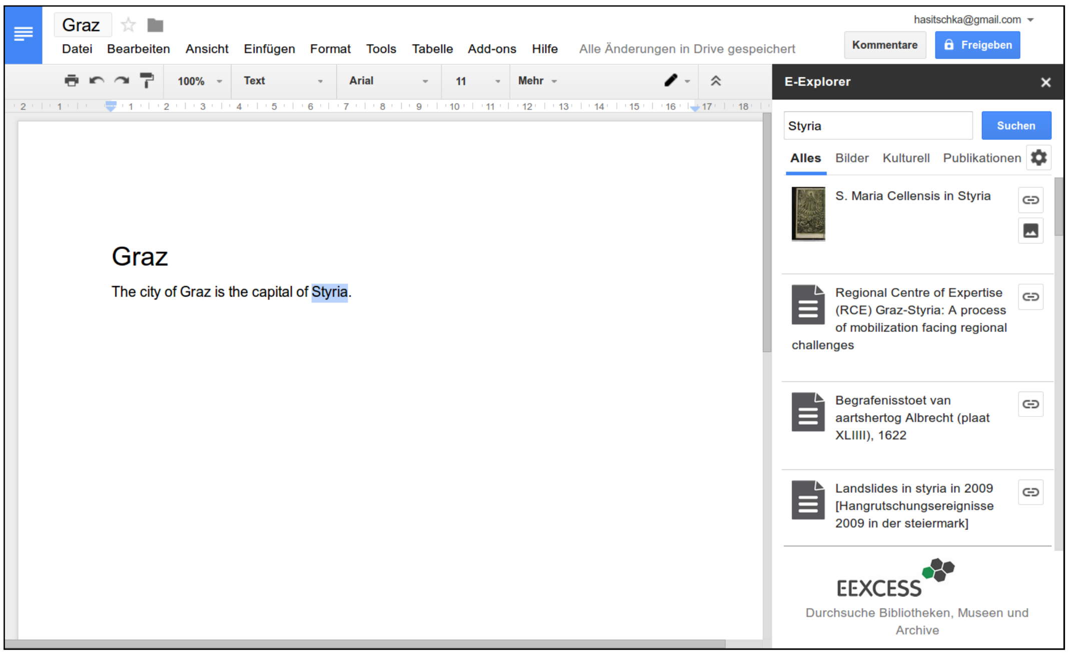
3.5 Technical Background

As can be seen in Figure 3.14, *EEXCESS* consists of a lot of sub projects, used on the server- or client side. The following sections describe structures, modules and interfaces to provide some technical background about their usage that is necessary for further understanding.

3.5.1 Code Structure

The code of each *EEXCESS* sub project is managed via *GitHub*[4]. To connect the modules, GIT-Submodules are used where only a commit of a foreign repository is stored to prevent redundant copying and updating from foreign repositories. This work was implemented as a submodule inside a fork of the `EEXCESS/visualization-widgets` repository.

[4]https://github.com/EEXCESS/


Graz
Datei Bearbeiten Ansicht Einfügen Format Tools Tabelle Add-ons Hilfe Alle Änderungen in Drive gespeichert
hasitschka@gmail.com
Kommentare Freigeben
100% Text Arial 11 Mehr

Graz

The city of Graz is the capital of Styria.

E-Explorer

Styria Suchen

Alles Bilder Kulturell Publikationen

S. Maria Cellensis in Styria

Regional Centre of Expertise (RCE) Graz-Styria: A process of mobilization facing regional challenges

Begrafenisstoet van aartshertog Albrecht (plaat XLIIII), 1622

Landslides in styria in 2009 [Hangrutschungsereignisse 2009 in der steiermark]

EEXCESS
Durchsuche Bibliotheken, Museen und Archive

Figure 3.13: Usage example of the *EXCESS GoogleDocs*-Plugin

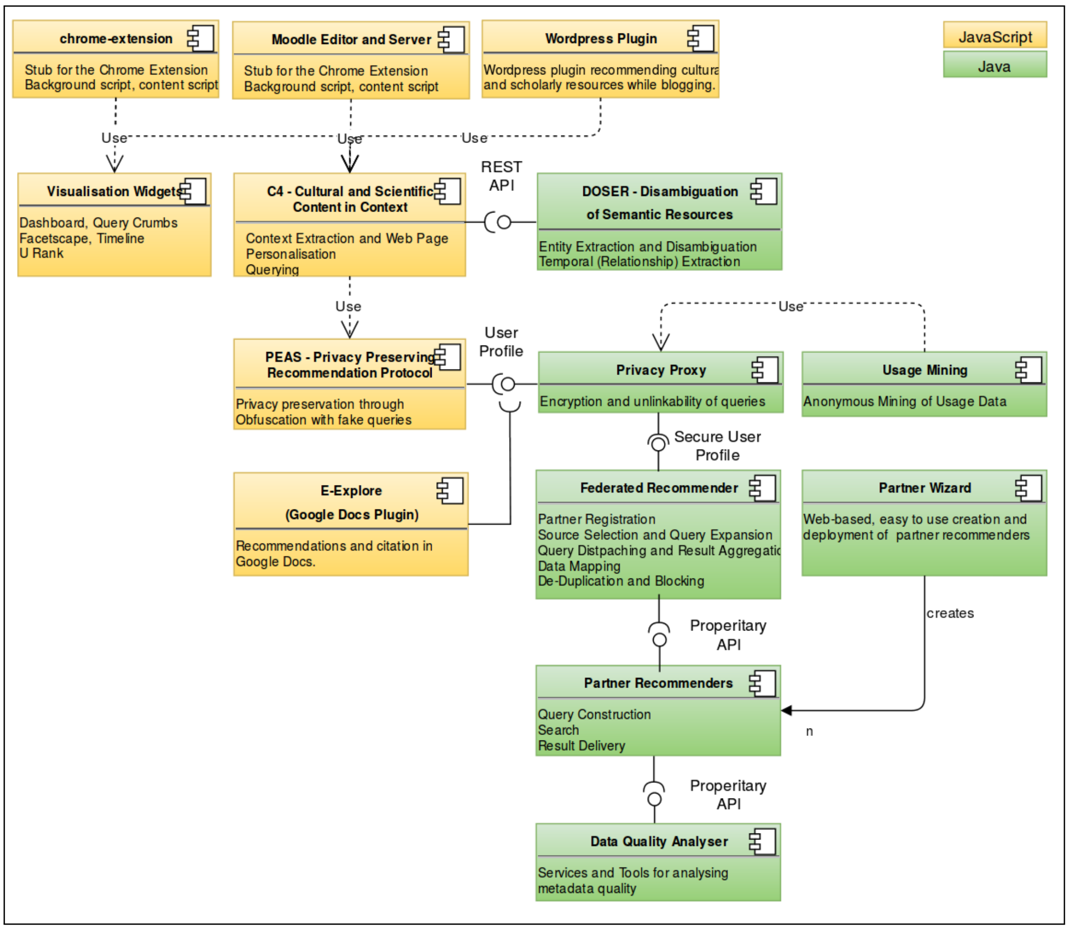

chrome-extension
Stub for the Chrome Extension Background script, content script
Moodle Editor and Server
Stub for the Chrome Extension Background script, content script
Wordpress Plugin
Wordpress plugin recommending cultura and scholarly resources while blogging.
JavaScript
Java
Use
Use
Use
Visualisation Widgets
Dashboard, Query Crumbs Facetscape, Timeline U Rank
C4 - Cultural and Scientific Content in Context
Context Extraction and Web Page Personalisation Querying
REST API
DOSER - Disambiguation of Semantic Resources
Entity Extraction and Disambiguation Temporal (Relationship) Extraction
Use
Use
PEAS - Privacy Preserving Recommendation Protocol
Privacy preservation through Obfuscation with fake queries
User Profile
Privacy Proxy
Encryption and unlinkability of queries
Usage Mining
Anonymous Mining of Usage Data
Secure User Profile
E-Explore (Google Docs Plugin)
Recommendations and citation in Google Docs.
Federated Recommender
Partner Registration Source Selection and Query Expansion Query Distpaching and Result Aggregatic Data Mapping De-Duplication and Blocking
Partner Wizard
Web-based, easy to use creation and deployment of partner recommenders
creates
Properitary API
Partner Recommenders
Query Construction Search Result Delivery
n
Properitary API
Data Quality Analyser
Services and Tools for analysing metadata quality

Figure 3.14: The architecture of *EEXCESS* (Source: *https://github.com/EEXCESS/eexcess*)

3.5.2 EEXCESS API

One core-functionality of *EEXCESS* is its recommender-service, the so-called ***Federated Recommender Service***. The API[5] can be called up through a URL, whereas the POST-method is used and keywords are sent within the POST Payload.

The JSON-response contains data of the received results from the different providers. A single result contains a unique ID, the URL to the document, a date, a preview-image and metadata like the provider, the license, its language and the media type. The latter are called *facets* in the following chapters.

A second possibility to get the results from the *EEXCESS*-API is using the ***Privacy Proxy Services***: Its request- and response structure is nearly the same as the one of the *Federated Recommender Service*. There are two main reasons why the *Privacy Proxy Service* should be used instead of calling the *Federated Recommender Service* directly:

- **Privacy**: As mentioned in Section 3.2, privacy was one of the basic challenges to address. To send as little data as possible from the client's browser to the *Federated Recommender Service*, a proxy service was implemented to be set between the client and the recommender server. Thus, the recommender does not get information about who sent the request containing sensitive data like keywords.
- **Logging**: The privacy proxy allows to log information for internal usage. Each query and response is logged implicitly by the proxy server. But it also has the possibility of explicit logging through different API-calls such as *moduleOpened*, *itemRated*, *itemCitedAsText* etc.

3.5.3 Recommendation Dashboard

The *RD* (described in Section 3.3) is responsible for providing different visualizations of the recommender-results. It is mainly written in *JavaScript* with some additional *html* and *CSS*-files, typical for frontend-web-applications.

[5]https://github.com/EEXCESS/eexcess/wiki/Federated-Recommender-Service

3.5.3.1 Used Libraries

The *RD* makes use of the power of different well known *JavaScript* libraries. Some of them are provided in the following list:

- For modifying and manipulating the HTML page's DOM ***JQuery***[6] is used, since it provides an extensive set of manipulation methods (Miles, 2016).
- Since *JavaScript* does not support a native dynamic loading of source-code files, ***requireJS***[7] and ***Modernizr***[8] are used to load files on demand, because of a lot of different modules and external libraries used within the dashboard.
- The ***D3***[9]-library, which provides a lot of visualization methods is used by most of the visualization widgets that are included as plugins.
- The *GeoView* visualization uses the ***Leaflet***[10]-library to provide an interactive map based on *OpenStreetMap*[11] data.

3.5.3.2 Plugin Structure

The *EEXCESS RD* provides a plugin-system for easily including new visualizations without touching the core-code. By registering a plugin in the `plugins.js` file and after adding a button in the `index.html`, it can be accessed through the frontend.

Each plugin has to have at least the following methods in its main file located under `Dashboard/Plugins`: `initialize`, `finalize` and `draw`. The plugin's entire code can be located at an arbitrary location inside the project.

[6]https://jquery.com/
[7]http://requirejs.org/
[8]https://modernizr.com/
[9]http://d3js.org/
[10]http://leafletjs.com/
[11]http://www.openstreetmap.org

3.5.3.3 Communication Between Modules

To help different plugins and modules of the *RD* communicate, such as when a query was triggered, results arrived or were rated, they use the `window.postMessage` method of the browser, to send a message and `window.onmessage` to listen for specific messages.

4 ECHO - Explorer of Collection Histories

This chapter first elaborates on the requirements from several use cases in Section 4.1 and further discusses all features derived from them in Section 4.2.

4.1 Requirements

Before defining *ECHO*'s features, it was necessary to identify the requirements in order to identify them. They can be devided into **functional requirements** (Section 4.1.2) and **non functional requirements** (Section 4.1.3). The requirements were derived from a list of possible use cases **use cases** (Section 4.1.1)

4.1.1 Use Cases

The following use cases were conceived in order to to identify which requirements were necessary for *ECHO*.

4.1.1.1 Visualizing a Collection of Recommended Documents

ECHO should support the user while using the *EEXCESS-Recommendation Dashboard* (see Section 3.3). Thus, the user wants an easy way to visualize a **data-collection** containing representations of recommended documents. The documents' available metadata should be visible and easy to understand.

Since the collections may contain a lot of results, the user may want to see this data summarized in a way that is easy to understand. Depending on the media type, a preview image can help the user to identify and categorize the underlying content.

4.1.1.2 Finding and Using Interesting Results

The visualization of a collection should allow the user to search and find interesting results. The recommended documents may contain textual content, images, videos, or sound files. Thus, a logical conclusion would be that the user wants to discover recommended documents by their file type. However, the documents' further metadata (*language*, *license*, *provider*, *type*, *year*) may also provide a valuable opportunity to find other interesting recommendations. The user wants to interact with the visualization to select and filter recommendations by their metadata.

Furthermore, the user may want to use the documents discovered in the collection. They may simply want to open the corresponding document, or they may want to use it as a reference externally: The *RD* makes it possible to use recommended results in external tools, depending on the current underlying *EEXCESS* frontend implementation (see Section 3.4). The user might also want to make use of this functionality in *ECHO*.

4.1.1.3 Visualizing the Query History

One basic motivation behind *ECHO* is the desire to explore former queries. This feature is not available in other *RD* plugins as of yet. Thus, the user may want to use *ECHO* to visualize the entire query history. Users may also want to navigate through a timeline of queries without much effort, and also want to have the possibility to access former queries and investigate them as described before. The user wants to easily identify former queries by specific characteristics, such as the number of results or the query keywords.

Queries are represented by **query-collections**. Each of them holds results from search queries.

Finally, one motivation for *ECHO* was to allow comparisons between different query-collections (see the motivation in Section 1.1). Thus, a clever method for comparing results across multiple queries is necessary.

4.1.1.4 Organizing Recommendations

Since the user may be interested in multiple results from different query-collections, it must be necessary to bookmark them, as is possible in other *RD* plugins. These bookmarks are stored in bookmark-collections. **Bookmark-collections** are created by the user to store individual results on demand.

The user might also be very interested in visualizing them in the same way as the query-collections are visualized. This may help to refine the bookmarks by using the power of *ECHO*'s visualization techniques.

4.1.2 Functional Requirements

Having defined simple the use cases in Section 4.1.1, *ECHO*'s functional requirements can be derived. They will help to determine the proper ways of designing and implementing *ECHO* afterwards.

4.1.2.1 Visualizing a Query-Collection Containing Results of Recommended Documents

Consideration of the use cases in Section 4.1.1.1 lead to the following requirements:

- **FR1.1**: Proper visualization of an *EEXCESS* search query and its results.
- **FR1.2**: A clear layout that lets the user easily identify different results. It must be possible to deal with dozens of results without causing a loss of overview.

4.1.2.2 Visualizing Results of Cultural, Historical and Scientific Data

When focusing on the type of data of which the results consist, further requirements can be derived from the Sections 4.1.1.1 and 4.1.1.2:

- **FR2.1**: Since the underlying documents of the results have different media types (text, image, audio, video, etc.), a good preview is necessary in order easily to identify their content.
- **FR2.2**: Each result has a set of metadata values (including the file type). These facets have to be visualized in such a way that the user can easily distinguish between and recognize them.
- **FR2.3**: A summary of a collection's facets should be visible in order to see its *character* and to be able to distinguish it from other collections (see Section 4.1.2.3).
- **FR2.4**: Since the collections can contain a lot of results, a filtering method is a must. By interacting with the visualization, the user should be able to filter the results by their facets.
- **FR2.5**: The relevance of each result must be easily recognizable by the user, allowing them to find the most valuable results inside a collection.
- **FR2.6**: Results should be usable in external tools, defined in the *EEXCESS* implementation, and it should also be possible to open them externally.

4.1.2.3 Dealing with Multiple Data Collections from a Search History

The use case regarding visualizing the query history (Section 4.1.1.3) leads to the following requirements:

- **FR3.1**: *EEXCESS* search queries and their results need to be stored for later usage.
- **FR3.2**: This resulting query history needs to be visualized. Thus, multiple collections must be visible at once.
- **FR3.3**: To see how the search queries developed over time, each query-collection has to be visualized in such a way that it can clearly be

identified and distinguished from others by taking the results into account.
- **FR3.4**: The user should be allowed to show differences, relations and similarities between different query-collections. Therefore, a visualization technique must be available to compare multiple collections to show them. It should be kept in mind that documents can occur in multiple collections. Thus, the visualization must allow connections between those documents to show their occurrence in different query-collections over time.
- **FR3.5**: To allow the user to extend research on interesting results to multiple query-collections, it is important that they be able to locate them by their facets. This is possible by extending a potential filtering method (mentioned in Section 4.1.2.2) to all collections.

4.1.2.4 Navigation & LODs

Combining the requirements above, it is important to consider the navigation and representations on different Levels Of Detail (LOD) offered by *ECHO*:

- **FR4.1**: It is necessary to visualize single results (Section 4.1.2.2), single collections (Section 4.1.2.1), and the whole query history (Section 4.1.2.3) simultaneously. Thus, it is necessary to deal with different LODs, in order to prevent clutter and user overload.
- **FR4.2**: A seamless transition between those LODs is necessary.
- **FR4.3**: Furthermore, navigation is a big challenge. The user will want to navigate easily through the query history and be able to discover a collection or single results.

4.1.2.5 Organize Results & Bookmarking

As discussed in the use case above (Section 4.1.1.4), it should be possible to visualize the *EEXCESS* bookmark-collections.

Dealing with bookmark-collections leads to the following requirements:

- **FR5.1**: *ECHO* should allow users to bookmark single results. The user needs to be able to either choose an existing bookmark-collection or create a new one.
- **FR5.2**: There should be the option to switch between visualizing the query history and a selected set of bookmark-collections. The bookmark-collections should be visualized in the same way as query-collections.
- **FR5.3**: It must be possible to use different results from collections in external tools.

4.1.3 Non Functional Requirements

In addition to the functional requirements (Section 4.1.2), other important prerequisites have to be defined:

Performance In contrast to other *RD* visualizations, *ECHO* has to deal with multiple collections, which means that the performance has to be considered.

It is necessary to consider the rendering technique used. Current visualizations are based on HTML5-Canvas rendering using the 2D API, which may or may not be hardware accelerated. The potential usage of a GPU-supported rendering has to be taken into account **(NFR1.1)**. Performance has to be kept in mind during the implementation process in general, so as to avoid bottlenecks later on **(NFR1.2)**.

System Requirements *ECHO* and the *RD*, as well as *EEXCESS* in general, should be usable through a common web browser. Thus, there are no special system requirements for using *ECHO*:

- **NFR2.1**: No special software should be necessary except a modern web browser.
- **NFR2.2**: *ECHO* also has to be usable without depending on special hardware and drivers.
- **NFR2.3**: *ECHO* should work on multiple operating systems.

Extensibility The implementation design should take extensibility with further features in future into account. Thus, well-documented, and clean code is necessary, as well as well-defined object-oriented modules **(NFR3)**.

4.2 Features

After describing all functional and non-functional requirements in Section 4.1, *ECHO*'s functional features can now be derived.

The following section begins with the design of a proper visual layout (Section 4.2.1) and navigation mechanism (Section 4.2.2).

Further features derived from the requirements are worked out in the following sections:

- filtering (Section 4.2.3)
- comparing collections (Section 4.2.5)
- collection summary (Section 4.2.4)
- bookmarking functionality (Section 4.2.6)

4.2.1 Visualization Layout

As a result of the requirements in the Sections 4.1.2.1, 4.1.2.2, and 4.1.2.3 this section deals with the layout of *ECHO*'s visualization.

In this section, first, the basic concept of the layout is discussed. Next, different ideas for the visual appearance of collections and results are discussed (Sections 4.2.1.2 and 4.2.1.3. This is followed by a concept for visualizing multiple collections at once (Section 4.2.1.4).

4.2.1.1 Basic Concept

As described in Section 4.1.2.1, the results of an *EEEXCESS* search query need to be visualized.

It seems likely that *search results* may be **listed**, ranked from the top to the bottom, with a clickable title and a short abstract, as people are familiar with from search engines like Google[1].

Since the documents behind the *EEXCESS* search results may contain images and videos, a representation of the documents in a **grid**, where their **preview image** is shown, may help the user to identify their content more easily.

Both of these approaches may be suitable for a simple visualization of simple collections, but representing multiple collections, as required in Section 4.1.2.3 may be difficult. Finally, when considering comparison and analysis of the query history, other methods of showing the results have to be taken into account.

Section 2.2.3 introduced different systems and approaches that visualize time-oriented **graphs**. Even if the concepts and underlying data differ, they all show that it is possible to find proper ways of **visualizing the chronological evolution of graphs**. Especially the idea of visualizing different states of the graph as *timeslices* and connecting the same nodes (see Figures 2.13 and 2.10) may help solve problems regarding the comparison on different query-collections. Thus, the decision was made to design the visual layout of collections and the query history based on of graph visualization.

4.2.1.2 Collection Design

Possible Network Layouts One basic idea is to represent the recommended documents as nodes in a graph. Since the nodes in a graph are connected to each other, there are two different ways to represent a collection as a network of nodes:

- make connections between all result nodes
- connect the recommendations only with a center node, representing the collection as a single element

Directly linking recommendations that share same metadata could help users find similar recommendations. However, since there are multiple types

[1]https://www.google.com

of metadata, connecting the nodes could cause clutter, and the graph could confuse the user. Furthermore, when considering the requirement to allow the user to compare different collections (see Sections 4.1.2.3 and 4.2.5), this comparison could be very challenging, since the graphs may differ greatly. Finally, Section 4.1.2.2 discusses the requirement to show a summary of the collection's results' metadata and to provide methods to filter on them. An unordered graph that may vary a lot, depending on the number and content of the results, could cause problems with visualizing such features, which are further described in Section 4.2.4. This resulted in the choice of the the so-called *RingRepresentation*.

Rooted Tree To avoid these problems, a more simple method of visualizing a collection of recommended results was conceived: a graph.

- the collection itself is presented as a central node in the graph
- the recommended results are connected as sub nodes of this central node

This structure results in a tree graph with the height 1: Thus, the collection node is the root and the recommendations are its leaves.

The next challenge was to find a proper layout for the graph, in order to best fulfill the requirements. The traditional way to visualize rooted trees is to place the root node on the top and the further nodes underneath. In the case of a tree with a height of one, the graph may look like the figure in the left hand side of Figure 4.1. Such a tree visualization typically makes it possible to easily recognize hierarchy structures and other characters of the graph. However, a tree with the height 1 does not contain further hierarchy levels. Thus, another layout may be more suitable for representing a result collection and its recommendation nodes.

Circular Layout Considering the requirement of dealing with multiple collections (Section 4.1.2.3), a more space saving layout is preferable. One possibility is to place the nodes on an imaginary circle around the center node (see Figure 4.1 on the right). This approach has the following benefits:

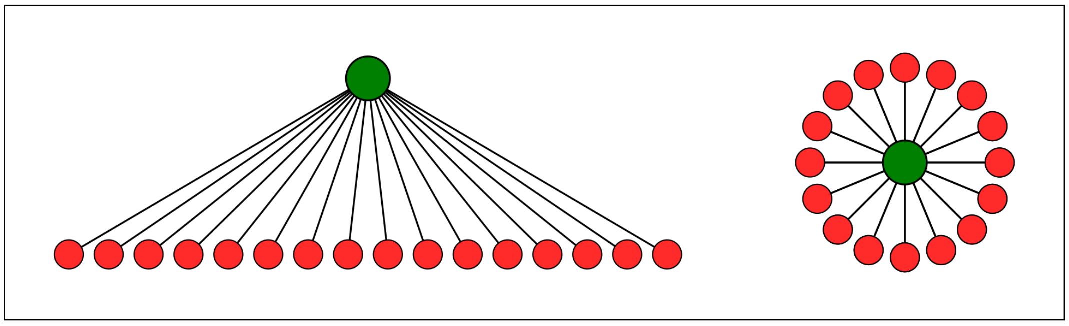


Figure 4.1: Comparing a common tree layout with a circular layout structure both representing a rooted tree with the height 1. On the left the root node is on the top and the leaves on the bottom. The right visualization orders the leaf nodes around the root

- It saves space compared to a top-down layout. In general, placing the nodes on a circle around the root node is a very efficient way to approach space usage.
- The space occupied by a collection is independent from the number of results, which helps when dealing with multiple collections.
- The basic length of the edges is the same for all nodes in contrast to the top-down layout (see Figure 4.1). This makes it possible to encode a visual variable into the edge length (as described in Section 4.2.4).
- The space between the two node levels can be used to represent different kinds of information regarding the collection and its nodes (see also Section 4.2.4 for further details).

These arguments led to the decision to use the **circular layout** to represent a single collection, which contains a center node and shows the recommended results as sub nodes evenly distributed around it.

Further Collection Elements The collection graph is augmented with the collection ID in the top left. In the case of a query-collection, its search keywords are listed on the right side. When visualizing bookmark-collections, a given name is shown (see Section 4.2.6). These keywords provide the possibility to identify matching results: when hovering over a single keyword, all recommendation nodes which have those keywords in their document's

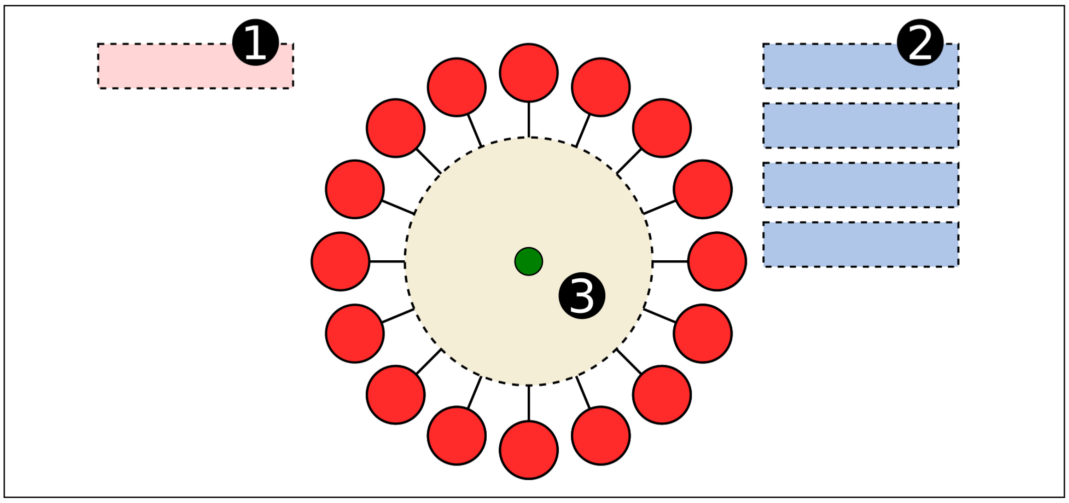


Figure 4.2: Schema of further (query-)collection elements: On the top left the collection ID is displayed (1). On the right side the keywords are listed (2) . The space in between the results and the center node is used for the *RingRepresentation* (see Section 4.2.4) (3).

title are highlighted in a specific color.

The place between the recommendation nodes and the collection node is further used by the *RingRepresentation* (Section 4.2.4), which shows a summary of the collection and allows the results to be filtered. Figure 4.2 outlines a collection containing the mentioned elements. A collection containing 20 recommendations and 10 search keywords can be seen in Figure 4.8.

4.2.1.3 Recommendation Nodes

Section 4.2.1.2 discussed the fact that the results are represented as nodes around the collection center. Those nodes are called **recommendation nodes** or just **recommendations** in the further text. This section discusses the layout and features of the recommendations with a focus on fulfilling the following requirements:

- defined in Section 4.1.2.2:

  - the user should be able to find relevant recommendations in the collection
  - depending on the document type, a preview may be necessary
  - each recommendation's underlying document must be usable, e.g. as a reference
  - the results must be easily distinguishable by their facet values

- defined in Section 4.1.2.5:
  - a bookmark function must be available for the results
- defined in Section 4.1.2.3
  - the user should be allowed to use the result's facet values as filters in order to find other similar recommendations

Position Around the Collection Given the fact that the recommendation nodes result from a ranked list of results from the *EEXCESS* recommender, it is crucial that this ranking be also easily recognizable for the user.

Anticipating the concept of the *RingRepresentation* in Section 4.2.4, the recommendations are not ordered by their relevance when a collection is in focus, since their position depends on their facet combinations.

However, if a collection is not focused, the recommendation nodes are ordered by their relevance around the collection. They are ordered clockwise, beginning with the most relevant recommendation at the top. For further details on the different navigation levels, see Section 4.2.2.

Since the nodes are not ordered by their relevance on the collection focus level, this value needs to be represented in another way: The **distance** of the recommendation node to the collection center thus correlates with its relevance. The higher the distance, the higher the relevance. Distance as a metaphor for relevance is further underpinned by another representation of the relevance used on the recommendation navigation level, where a greater relevance results in a bigger node. This second representation is necessary, since the distance of the node is not visible on this navigation level.

Layout Since it is typical to represent single nodes in a graph, recommendations are represented as filled circles. This makes it possible to use the same representation, independent of the current navigation level (outer navigation, collection focus, recommendation focus - see Section 4.2.2), since other representations, such as complex visualizations, encode different values in it, which may only work on specific navigation levels due to the different sizes of the node on the user's screen.

The circle representation also provides the opportunity to fill it with other important information about the node when the user zooms in. As defined in the requirements (Section 4.1.2.2), the **preview** should help the user identify recommendations by their content. Depending on the media type, *EEXCESS* results already contain a URI to a preview image. If this image exists, it is loaded by *ECHO* and used inside the node's circle to allow the user to get an idea of the recommendation's content (see Figure 4.4). If no preview image exists, the *EEXCESS* logo is used as a placeholder. On the outer navigation level, where the user navigates through the query history, the recommendation nodes appear too small to show a meaningful preview image. Thus, on this level, the preview image is not shown, and only the node as a circle, appears.

According to Section 4.1.2.2, the user should be able to find relevant results in a query collection. As described above, a result's **relevance** is already stored in the *EEXCESS* results, since it is also used in the *RD*. The relevance is represented as the distance between the recommendation node and the collection center. When focusing on a recommendation, this connection is not visible anymore, and thus, another representation of the result's relevance is necessary. For this purpose, the **size of the node** is used. The more relevant a node is, the bigger it is, whereas the preview image always remains the same size. The variable area between the outer rim of the node and the preview image can thus easily be recognized, and the user can identify the relevance of the node even when only the recommendation node is visible on the screen. Figure 4.3 sketches the two representation of the recommendation nodes' relevance.

In addition to the relevance, another variable of the recommendation nodes is encoded in their visual appearance: the node's **color** represents a value of a specific facet. The facet can be configured in *ECHO*'s configuration file

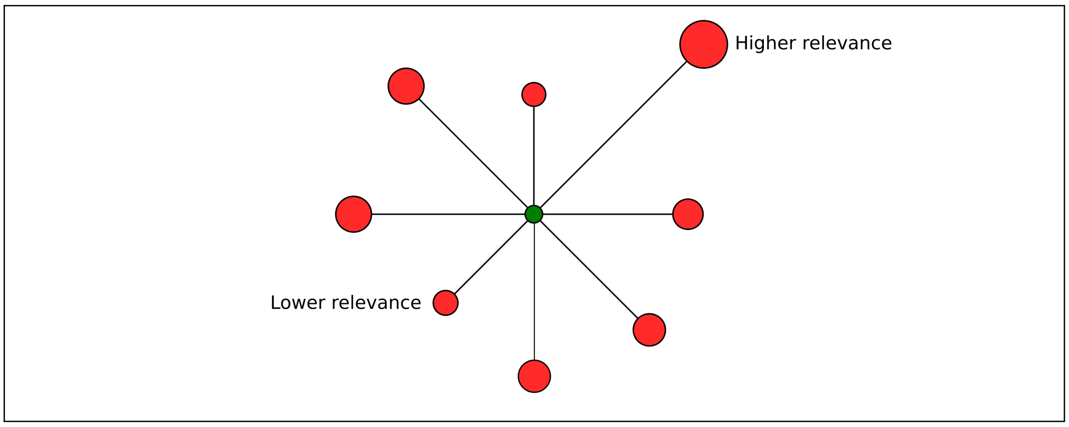


Figure 4.3: Relevances in *ECHO*: A higher relevance results in bigger recommendation nodes and a higher distance to the collection center.

(see Section 5.10). The color value itself is defined in the *RingRepresentation* settings. For further information on the *RingRepresentation* and its color encoding see Section 4.2.4.

Recommendation Details Looking at Figure 4.4, several buttons can be seen around the preview image: These appear when a single recommendation is in focus. The other two navigation levels (see Section 4.2.2) do not show them. The buttons are equally distributed between the outer rim of the node and the preview image. Their purpose is to fulfill the requirements of the **external usage** of the underlying document (see Section 4.1.2.2) and the need for a **bookmarking** functionality (see Section 4.1.2.5).

- **Star**: Gives the user the possibility to **bookmark** this document. The *RD*'s bookmark window opens to select an existing or a new bookmark-collection to add. (See Section 4.2.6)
- **Arrow**: Lets the user **open the link**, guiding them to the document in a new browser-tab.
- ***i***: This buttons opens the **info-panel**, which is described below in detail.
- **Chain**: As the *RD* allows integration into content creating applications like *Wordpress* or *Google-Drive* (see Chapter 3.4.2), this button was

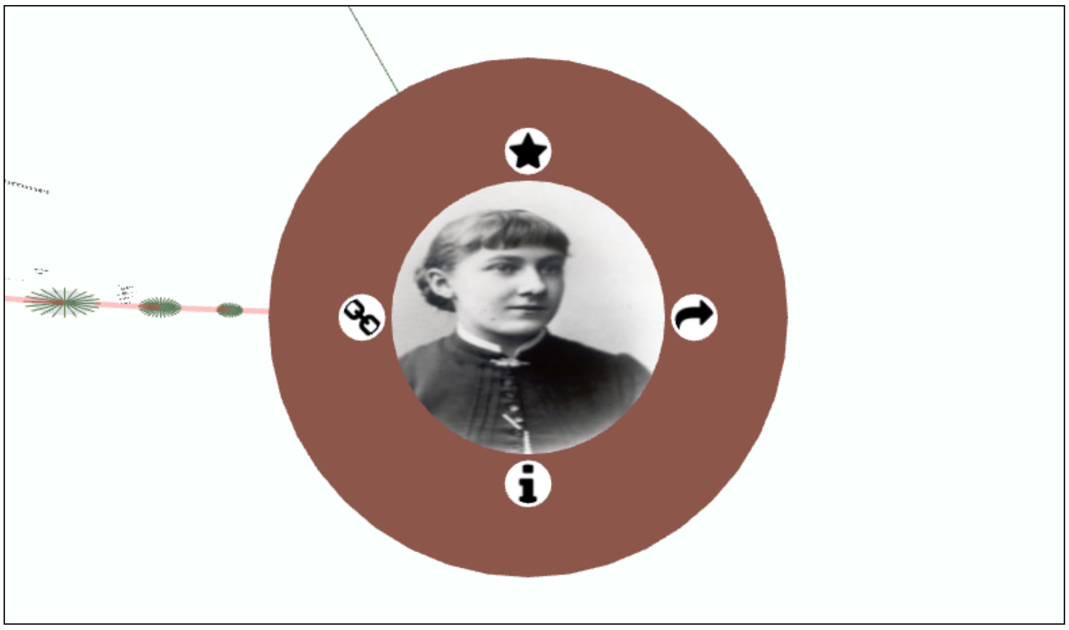


Figure 4.4: A focused recommendation node, showing its content's preview and four interaction buttons.

included to enable an external callback that integrates this document in such a tool (e.g. insert link) by clicking on it.

To help the user understand the buttons' functionality, a mouse-hover shows a short text below the button.

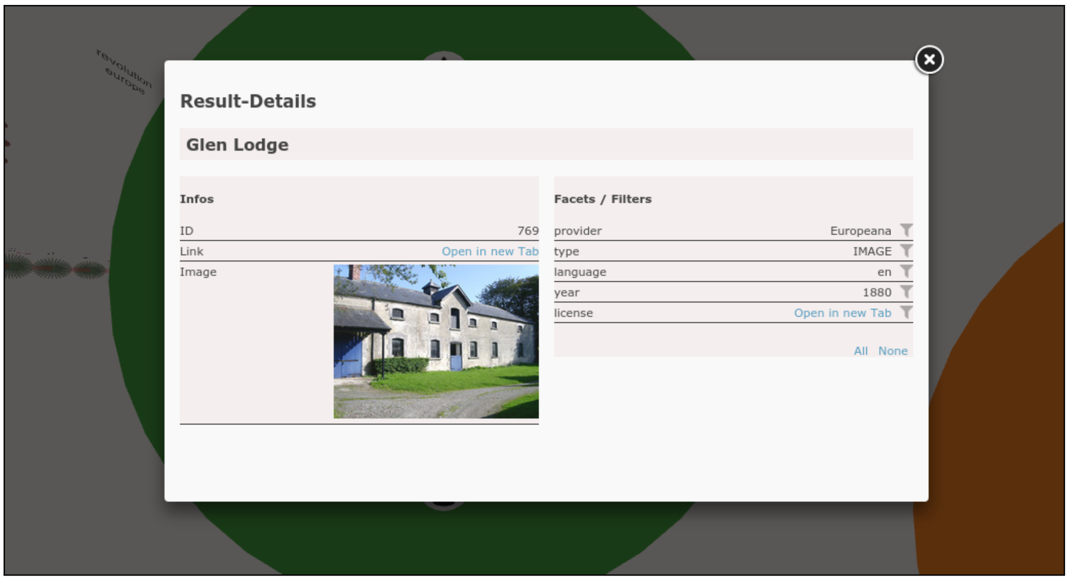
After clicking the ***i***-button of a recommendation node, its **info-panel** opens. This *window* is visualized through a superimposing HTML-element (see Figure 4.5) that can be closed by clicking anywhere outside of the window or by using the *X*-Button in the upper-right corner. It provides more details about the recommendation node's document. Its title, the preview-image and the facets can be seen. The user has the option to open the document in a new browser-tab. The facets are listed on the right side of the panel. If the document's license contains a link it can be opened in a new tab right from the panel, in order to get more information about it.

The info-panel not only summarizes a document's data, it also provides methods to apply filters (which are described in detail in Section 4.2.3) on the scene. Each row describing a facet and its value contains a funnel-symbol. Clicking on it adds or removes a filter for that facet, containing the corresponding value. Further below the facet- table the user can apply or remove all facet-values as filters with a single click. Applying all facet-values also means that only those recommendations that contain exactly the same facet-values remain totally visible.

4.2.1.4 Query History

ECHO's layout is finalized by specifying how multiple collections from the query history should be represented in the entire visualization.

As defined in the requirements (Section 4.1.2.3) it must be possible not only to show multiple collections, but also to allow the user to easily recognize a specific search query and its results when navigating through the history. For that reason, the collections contain their search query keywords and an ID (see Section 4.2.1.2).



Result-Details

Glen Lodge

Infos	
ID	769
Link	Open in new Tab
Image	

Facets / Filters	
provider	Europeana
type	IMAGE
language	en
year	1880
license	Open in new Tab

All None

Figure 4.5: A recommendation's info-panel overlays the scene and provides further information and interaction possibilities

Layout A layout has to be chosen that is able to show the collections in such a way that they also fulfill the requirement of being comparable (which is described in detail in Section 4.2.5). It is possible that a lot of search queries will be made that all have to be visualized. When trying to visualize them all, the following challenges have to be considered:

- **multiple collections** at **any position** of the query history need to be comparable.
- Comparison means that the corresponding elements **need to be visible**.
- Thus, it must be possible to show different collections of the history at the same time, **independent** of their **amount** or **position** in the query history.

To allow the user to recognize the chronological order of the query-collections, a basic approach is to visualize them from left to right. This is a typical way of showing a chronological graph (Harris, 2000, p. 417). However, just putting the collections on a line from left to right would increase the challenge to have multiple collections visible at the same time.

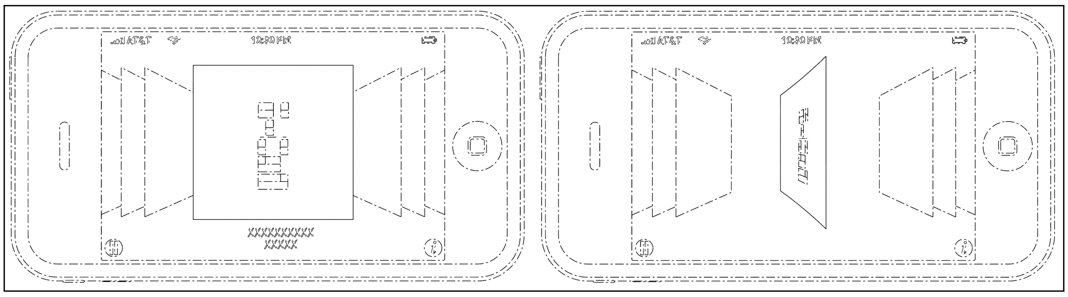


Figure 4.6: Apple's *CoverFlow* visualization of items used on mobile devices (Chaudhri, 2010)

CoverFlow One famous approach to deal with a visualization of ordered items, where previous and next elements should be visible is Apple's ***CoverFlow*** (Chaudhri, 2010) (see Figure 4.6, left). Its basic idea is to show the current element in the center and to stack the other ones behind it, so they are visible, but do not take up too much space.

As mentioned, elements hiding each other may cause problems with regard to their comparability. However, since *CoverFlow* may solve problems when dealing with a lot of collections, it is worth trying to modify its layout to fulfill all requirements.

Circular Arc Layout A first solution to deal with the space problem could be to change the collections' orientation. If they are positioned on an imaginary circle and all face away from its center, they use less space in the field of view, the further they are from the center collection. As Figure 4.7 (left) illustrates, all collections can be positioned in an area that is relatively narrow, but allows the user to see them all. To avoid hiding collections that are on the opposite side of the circle, it remains open. This also helps to prevent annoyance when viewing the closest collection by not positioning others behind it.

Thus, the concept for a layout that allows the user to get an overview of the collections in the query history can be summarized as follows:

- All collections should be positioned on a horizontal **circular arc**. The first and the last collection are not directly connected to allow the user

to easily identify the beginning and end of the chronological ordered elements.
- Not closing the ring may also help prevent disturbing collections in the background when viewing one in the front. However, this depends on the camera's angle of view and distance.
- All collections face to the outside of the ring. Thus, a collection directly in front of the user's view is oriented directly towards the camera. Other collections face away from the camera. This has the effect that collections that are farther away need less space in the field of view.
- The user is allowed to navigate around this ring (for details on navigation, see Section 4.2.2).

Field of View Problem These definitions provide a clean solution for visualizing the entire query history but the ability to view all collections at the same time results in a relatively small representation of each individual collection.

Since, in *WebGL*, the user's view is defined by the camera, which has a position, a zoom-factor, and an angle of view, several experiments regarding the combination of these three factors were done to both gain a good overview of the collections as well as to show a single, selected collection as big as possible so as to allow the visualization of many details. Solutions that met these requirements led to a trade-off that resulted in a very strong *fish-eye* effect, known from real life cameras with a very short focal length. More details were visible due to the larger field of view, and elements in the center appeared larger than those near the edges, but the whole scene was heavily distorted.

This challenge led to the idea of splitting up the query history view and the collection view. Thus, the camera changes its variables, mentioned above, to allow the detailed visualization of a single collection. This idea was also combined with changes to the collections' rendering, and thus, different **Levels Of Detail** (LODs) are provided, depending on the current focus. These LODs and the entire navigation are described in detail in Chapter 4.2.2.

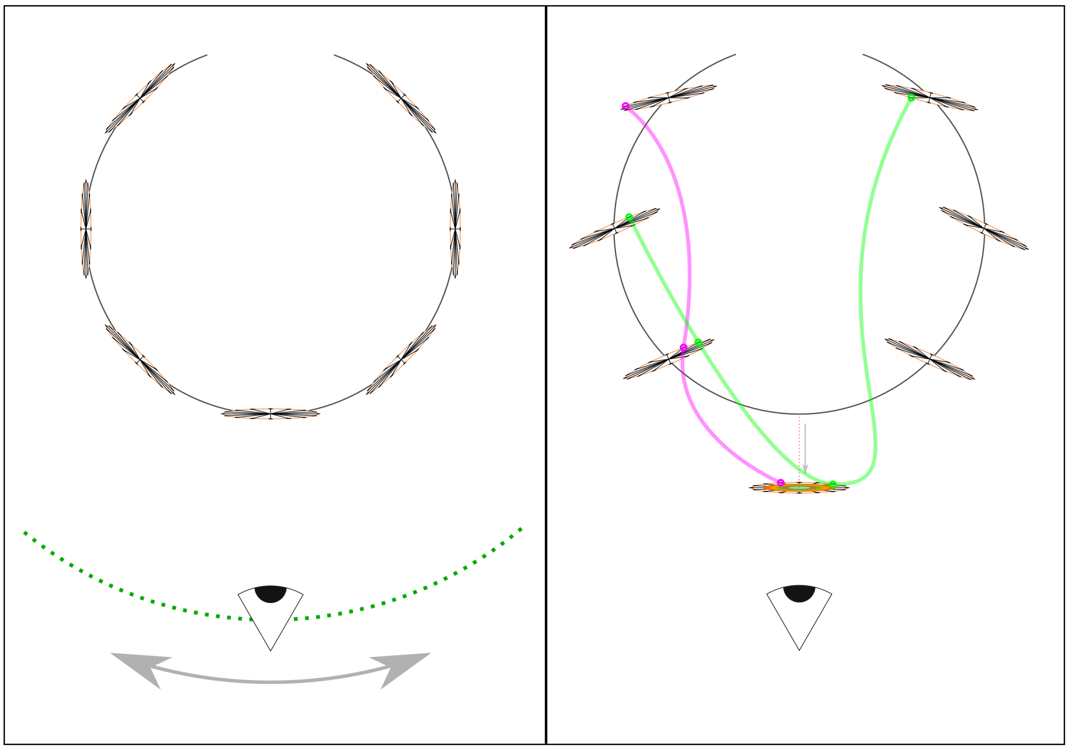


Figure 4.7: Query history Layouts depending on the current focus. Left: No focus set (overview). Right: The focus lies on a specific collection and comparing is activated (see Section 4.2.5)

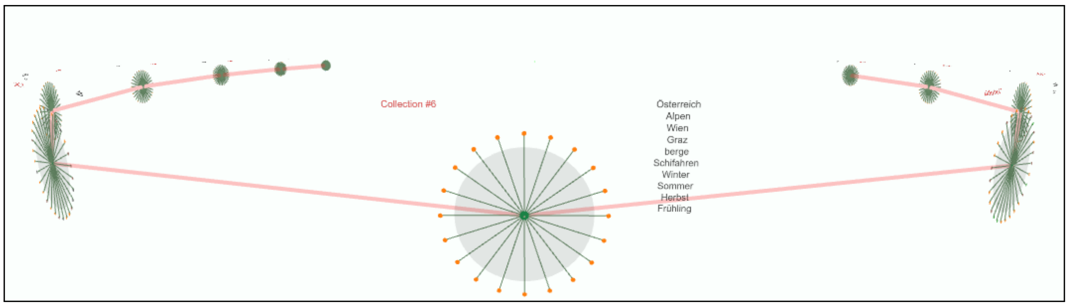


Figure 4.8: Overview over a query-collection in the query history

4.2.2 Navigation & Levels Of Detail

After discussing different solutions that were able to fulfill the requirements for the combination of **navigating on three different levels**, the final approach used for *ECHO* is described in detail in the current section. In addition to the navigation in the scene and between the different LODs, different input methods are also discussed below.

Initial Challenges According to Section 4.1.2.4, three challenges need to be managed:

- the user needs to be able to operate on three different levels:
  - in the query history
  - on a single collection
  - on a collection's result
- Single collections may show a lot of details and the recommended results' nodes may go into detail even more. Thus, different LODs that show, remove, or replace elements, depending on the current user's navigational needs may be possible. This helps to remove clutter and to improve the performance.
- The different levels lead to different requirements regarding the navigation. It must be possible to navigate easily through the query history. However, investigating a collection or recommendation node may need a more *fixed* behavior, that helps keep the focus on the element.

Basic Navigation Concept Considering a case of typical use, the user mostly starts by searching a collection in the history. According to Section 4.2.1.4, all collections are positioned equally on a circular arc from left to right (see Figure 4.8).

Two scenarios have to be kept in mind before deciding on a navigation approach:

- The user wants to scroll through the collections before selecting one.
- The user knows exactly which collection is the desired one and wants to select it, regardless of its position relative to the camera.

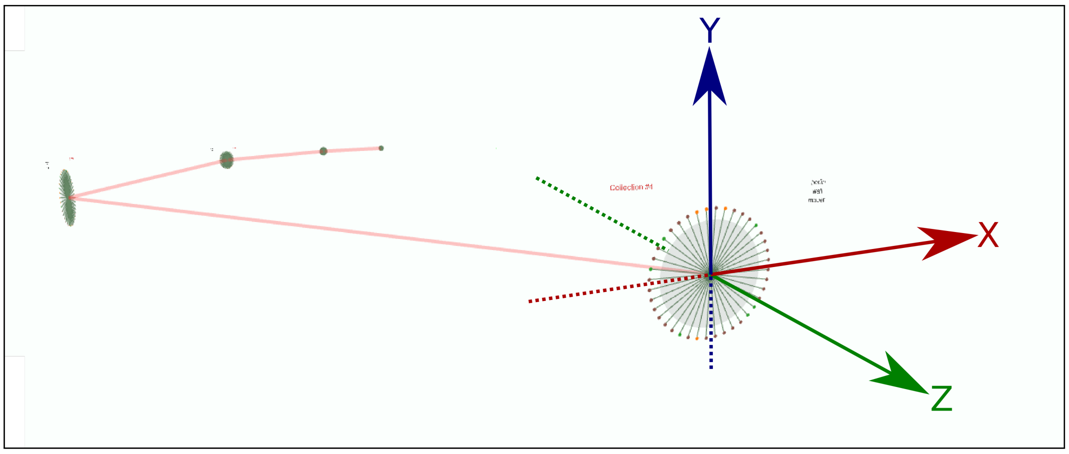


Figure 4.9: The X-, Y-, and Z-axis in *ECHO*'s three dimensional space

Thus, navigation by means of clicking on a collection must be possible, as well as *free navigation*.

Since the scene lies in a three dimensional space, navigation is not as simple as it would be on a two dimensional plane. One basic idea is to allow the user (or, more precisely, the camera) to move on the outer side of the circular arc, keeping the view on its center (see Figure 4.7, left). This allows access to every collection in the history. However, the following questions are still open:

- The navigation happens in the X-Z plane (see Figure 4.9), where the circular arc lies. Which Y-position should be chosen? Should the user be allowed to change it?
- Which input devices can be used to navigate (mouse, keyboard) and how does the interaction with the scene work exactly (fixed steps, or free movement around the arc)?
- Is it useful to allow the user to navigate freely in the three dimensional space in order to view the query history from any position or distance?

A clear way to design the navigation would be to restrict the Y-position completely. The user only could navigate around the circle containing the collections, navigating left and right. This concept would be the most intuitive and easy to use since there is only one axis. However, according to requirements of Section 4.1.2.3, documents that occur in different collections should be connected. Since these connections, which are described in detail in Section 4.2.5, should be visible, this solution is not optimal, due to the fact that the connections on the inner side of the circular arc are hidden by the collections in front of the camera. Figure 4.7 shows a sketch of those connections on the right.

The Navigation Sphere (Level 1) To solve that problem, a fixed Y-position could help the user to *look* beyond the collections into the circular arc, as shown in Figure 4.8. It is hard to find an optimal Y-value, since the connections and number of collections can differ. Thus, the following solution was designed and chosen for implementation:

- The user can navigate around the circular arc from a fixed distance.
- There is no initial Y-value that allows the user to look beyond the collections in the inner part of the circular arc.
- However, the user can also navigate up and down on an **imaginary sphere**.
- This means that the camera also can reach the highest point on that sphere directly on the top of the circular arc's center, but can also move below the collections to watch them from the bottom of the arc.
- This gives the user a **high level of non-restricted navigation** on the one hand, but also prevents them from getting irritated by a too free navigation and *getting lost* in the scene.

Focusing a Collection (Level 2) The user can navigate through the collections, as discussed in the last paragraphs. If an interesting collection is found, it can be selected for further exploration.

A click on a collection's graph should set a focus on it. Therefore, a change in the camera's attributes is necessary to allow the visualization of more details of the collection.

There are two basic ways to modify the camera to allow a collection to take more space in the field of view:

- move the camera towards the collection

- increase the zoom factor (focal length)

Both possibilities lead to similar results: the collection in focus takes more space on the screen, while other collections disappear outside the canvas' edges. However, another problem also arose: since collections should be comparable, especially when one of them is in focus (see Section 4.2.5), a maximum number of them should be visible, even on this navigation level. Since focusing on the collection, by zooming in or moving the camera, moves other collections out of the field of view, comparison becomes very difficult.

To circumvent this problem, a third method was conceived which provides a better view of the collection, while also showing as much of other collections as possible:

- The camera does not change its distance to the circular arc or its focal length when the user triggers a collection focus.
- The camera moves directly in front of the collection center.
- The focused **collection moves toward the camera**.
- This method keeps the field of view on the other collections.
- Nevertheless more details of the collection can be seen due to the increased proximity of the camera.
- *Unfocusing* the collection results in it moving back to its initial position.

According to the *CoverFlow* idea, only the focused collection should be seen from the front. The other collections have their orientation in another direction, depending on their position in the circular arc. However, since the selected collection finally moves when it is in focus, a further improvement is possible by making further changes to the layout:

After solving the problem regarding the visibility of other collections when performing a collection focus, the following approach helps make the content of these collections more visible:

- When a collection focus is triggered, the camera moves to a position directly in front of the focused one.
- All other collections' front sides turn to the direction of the camera (see Figure 4.8 on the right). This allows the user to see all collections from the front, regardless if in focus or not.

- Since a lot of changes occur simultaneously, all of those including the movement of the focused collection are **animated** to allow the user to recognize all the current changes.

Focusing a Recommendation (Level 3) Recommendation nodes, which are described in detail in Section 4.2.1.3, also can be focused (see Figure 4.4). This results in the third navigation level, after the query history navigation and the collection focus.

Instead of moving the camera on the virtual sphere as is done when focusing on a collection, it is positioned directly in front of the recommendation. Figure 4.10 illustrates the position of the camera in this navigation level on its right side. Since the collection's plane is tangent to the collection-circle and the recommendation node is also displayed as a plane, moving the camera on to the intersection of the connection between the sphere-center and the node with the sphere itself would result in a view of the node slightly from the side. Furthermore, the camera also changes its distance to the node. Thus, it appears larger on the screen.

Input Methods and Navigation Level Transition After having described the different navigation levels above, this section explains different input methods, how the user can interact with the scene, and which possibilities exist to focus on collections or recommendations and easily change the navigation level.

ECHO allows the user to navigate through the graph by using the keyboard or the mouse. Both have their own advantages and peculiarities:

- Typically, the user intuitively tries to use the **mouse** in the browser to navigate through interactive two- or three-dimensional visualizations, such as maps, for example. Thus, mouse-navigation was necessary to implement.
  The user can move around the camera-sphere by dragging the mouse, which means that the horizontal and vertical distances between a mouse-down-event and a mouse-up-event affect the camera's movement.

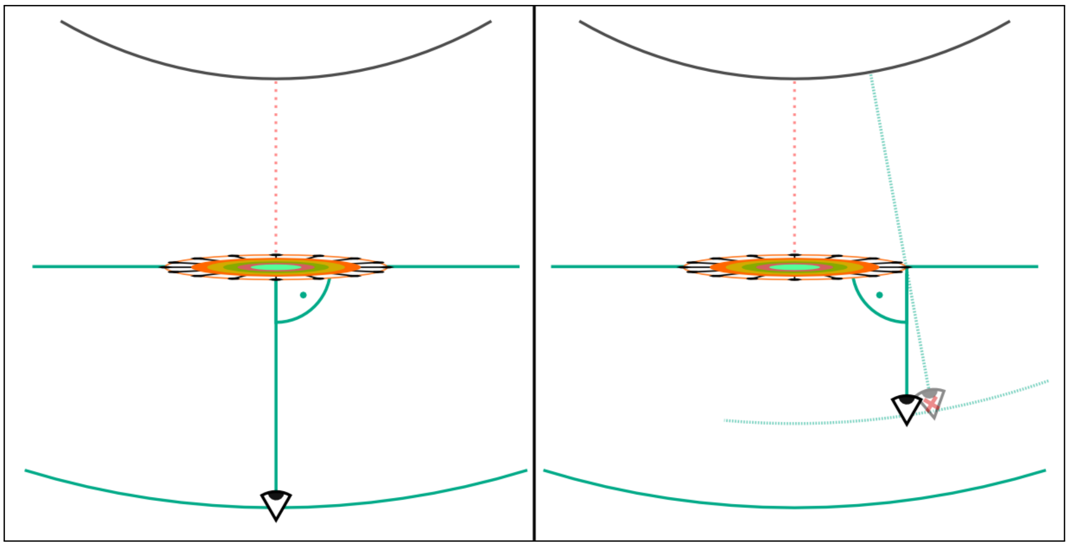


Figure 4.10: Illustration of the camera positions when focusing on a collection (left) and when focusing on a recommendation (right) where the camera leaves its path on the virtual sphere to ensure an orthogonal view on the recommendation node

- Further, the user can use the **cursor keys** of the keyboard to move the camera. This may be necessary if a comparison (see Section 4.2.5) is active and the user wants to avoid losing the splines between the collections. In this case, using the cursor-keys allows them to move around the scene and to investigate the compared collections.
  Moving the camera to the top of the scene gives a good overview of the comparison splines' paths.
- To easily change the navigation-level, the **mouse-wheel** can be used. While it is possible to go deeper into details by clicking on a collection or recommendation, it is not possible to return to the previous navigation level through mouse-clicks.
  Scrolling down with the mouse-wheel changes the current navigation level if a collection or a recommendation is active, by *zooming out* to the next outer level.
  Otherwise, scrolling up has the same effect as clicking the left mouse-button. Thus, this operation only works when the mouse is hovering over a collection or a recommendation. (Depending on the current

level).
Summarizing these two navigation options, combined with the two possibilities of moving the camera, the user can navigate through the three-dimensional graph and its navigation levels easily and quickly.

Except for the movement on the navigation sphere on the outer navigation level, all changes to the camera's position and navigation level are **animated**. This helps the user to keep track of changes that have been initiated by the interaction. For example, if a click on a collection occurs while navigating through the query history, the following happens:

- *ECHO* calculates the shortest path from the camera's current position to the point in front of the selected collection.
- The camera starts to move until this point is reached, while always pointing to the center of the circular arc that holds the collections. The movement slows down at the end to provide the user the feedback that the desired collection has been reached.
- At the moment the collection focus is initiated, the collections begin to change their orientation and the focused collection begins to move towards the camera to reveal more detail.

If a collection focus is initiated while another collection was focused, the process is completely the same, with the additional movement of the previously selected collection back to the collection arc.

Different Levels Of Detail As mentioned above, the three navigation levels correlate with different Levels Of Detail (LODs). They are necessary to fulfill the following non-functional requirements mentioned in Section 4.1.3.

- **performance**:
  - Each object, visible in the scene consists at least of one *WebGL* primitive, which is stored in the client's RAM.
  - Especially when a lot of collections with many recommendation nodes exist, the number of vertices used to represent the objects increases.

  - Therefore, since the *THREE.js* framework makes it possible to set the number of nodes that define a circle, the recommendation nodes are visualized as simple circle primitives with only a few vertices in the first navigation level, where the user can navigate through the query history, and the appearance of the small recommendation nodes is insignificant.
  - In the case of a collection focus (level 2) or a recommendation focus (level 3), the objects representing them in the focused element are destroyed and replaced with circles that have more vertices, to prevent a polygonal visualization of the nodes. This results in smooth circles. Furthermore they contain the preview image in their center.
  - Similar to the recommendation nodes, the *RingRepresentation* (see Section 4.2.4) only appears when a collection is in focus. Upon unfocusing it, all elements are deleted and recreated again on demand to save memory.
- **clarity**:
  - It is important to prevent visual overload that causes irritation and a loss of focus on important scene elements.
  - Thus, the recommendation buttons are only shown on level 3, since they are too small to use in the outer levels.
  - The preview images are only shown while a collection or recommendation is in focus (see Figure 4.14 as an example).
  - As mentioned above, the *RingRepresentation* would also be too small to identify specific facet values in the outer level. Thus, it is only shown on both inner levels.

4.2.3 Filtering

One crucial requirement was to allow the user to filter the recommendations of all visualized collections (see Section 4.1.2.2). For that purpose, the *facets*, a set of properties each result has, are used. The different facets (*language*, *license*, *provider*, *type*, *year*) are described in detail in Section 5.6.1.

Filtering the recommendation nodes can help the user to find similar recommendations over multiple collections easily. The following section first describes the data used for filtering and then discusses the visual appearance of filters and how filtering works in *ECHO*. Finally, possible connections between *ECHO*'s filters and those of the *RD* and its *Micro-Visualizations* are described.

4.2.3.1 Filtered Data

As mentioned above, every recommended document contains a set of properties summarized as *facets*. Since all recommendations share those properties and contain interesting information that characterize the documents, they are ideal for filtering.

Data Quality As mentioned in Section 3.2, *EEXCESS* retrieves its results from different providers. This fact can lead to variations in the results' facet-values. Especially when visualizing all values of the results' facets through the *Ring-Representation*, these differences are clearly visible:

- **Different date-formats**: the values of the *year*-facet often differ between simple years ("`2016`") and full dates ("`2016-02-03`"). Also timestamps containing hours and minutes are retrieved sometimes ("`2016-02-03-1950`"). Even completely internal time-codes of specific providers or completely corrupted strings can be retrieved. Improvements were made during *ECHO*'s implementation inside the *RD* to prevent most of the malformed values. The date should be displayed as a four digit year-string. Since the problem is significant, it may be taken up by a future work (see Section 7.3). Figure 4.11 shows a screenshot of an earlier state of *ECHO* that did not yet include mechanisms to identify and fix malformed values.
- **Interpretation of the date**: some dates refer to the creation of the underlying original document. Others describe the date that it was added to the library. However, we currently do not receive the exact semantics from the *EEXCSS* recommender.
- **No value**: some values are not even set. They are displayed as `unknown` values.

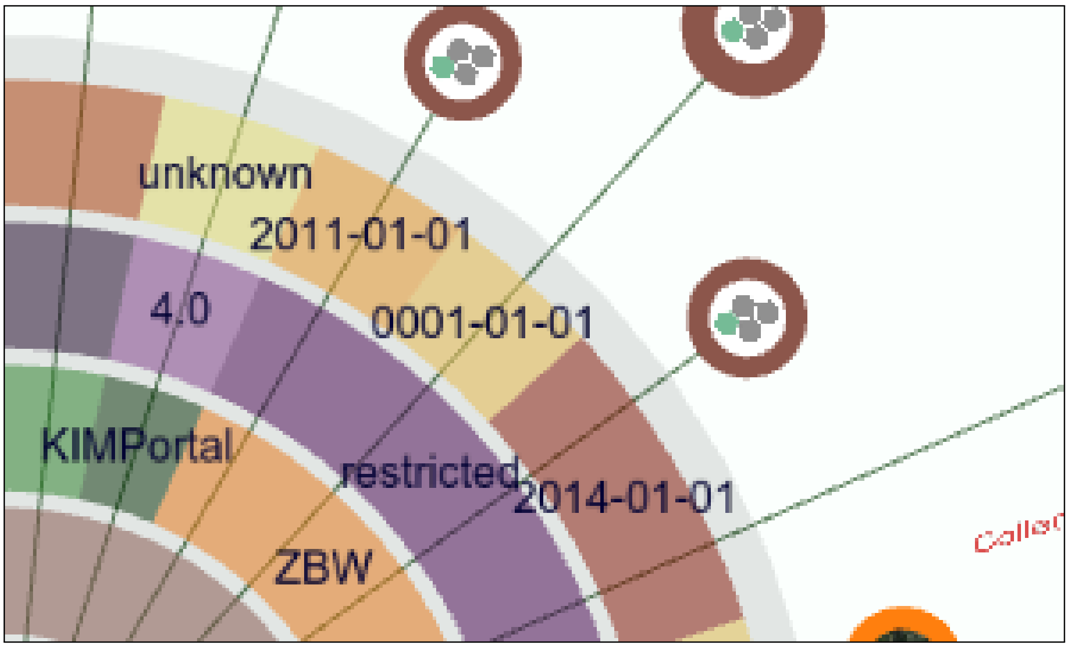


Figure 4.11: Different quality of facet-values visible inside the *Ring-Representation* in an earlier version of *ECHO*. The *year*-facet contains wrong ("`0001-01-01`") and not-set ("`unknown`") values.

4.2.3.2 Visual Appearance Of Filters

All recommendation nodes are positioned around their collection (see Section 4.2.1.3). Depending on a potential focus on the collection, their distance to the collection-center may vary due to their relevance.

If one or more filters are applied, all recommendations that do not match the values of the filters are *faded* so that they cannot be mixed up with the matching results, but are also visible and can be clicked. In detail, a *faded* recommendation node has the following characteristics:

- **The nodes are *collapsed***: a node that was filtered out ("negative" node) decreases its distance to the collection-center to a minimum. Thus, it can easily be distinguished from the positive nodes.
- **Lower opacity**: the nodes are faded out until they are semi-transparent. Background-color and preview-image are still visible but not as bright and noticeable as the "positive" nodes.

4.2.3.3 Filtering-Methods in ECHO

Inside *ECHO*, the user has the possibility of using two methods to set or modify filters:

- **Using the *Ring-Representation***: The *Ring-Representation* is the easiest way to set, modify or identify filters. Each possible value of a collection's results' facets is represented in a *sunburst-diagram* and is clickable to set or unset the filters. More details about using the *Ring-Representation* for filtering can be found in Section 4.2.4.
- **Using the info-panel**: After focusing on a recommendation node, the user has the possibility to show more information about the document in a separate panel. There it is possible to set or unset the document's facet-values as filters. See Section 4.2.1.3 for further details.

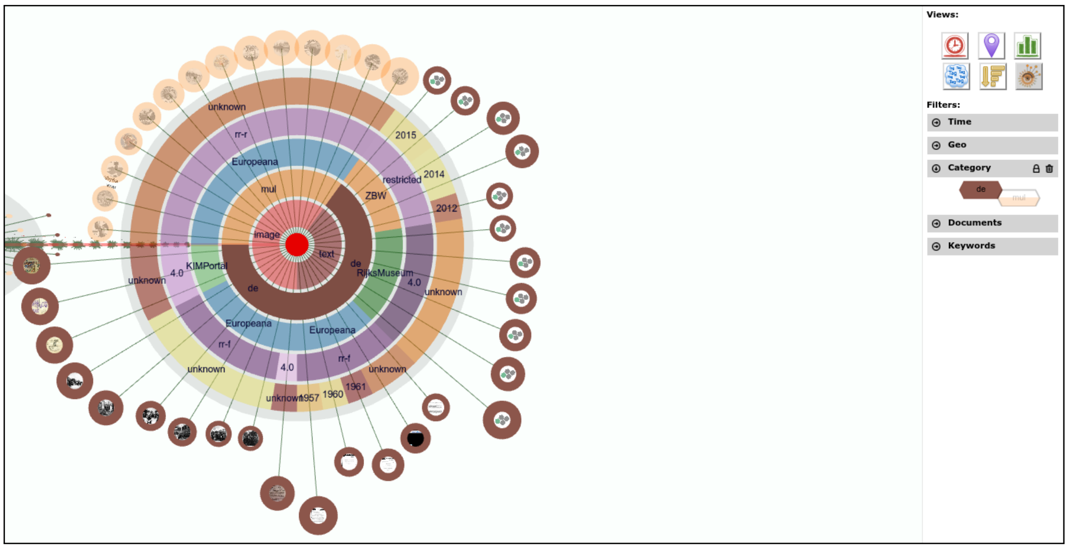


Figure 4.12: Micro-Visualizations with applied filter on the "*language*"-facet

4.2.3.4 Using the Micro-Visualizations

Depending on the plugin, the *RD* already supports brushing and filtering in different ways (see Section 3.3). The brushed or filtered values are displayed inside the *Micro-Visualizations* (see Section 3.3.3) on the right side of the dashboard. The big difference between the existing filtering & brushing mechanisms and the filtering in *ECHO* is that the existing ones are only applied on the current collection loaded in the *RD*.

This led to a challenge to connect existing filters from the *RD* with those from *ECHO*. Applying an *ECHO*-filter, like the language, for example, does not make sense if those filtered languages do not occur in the actual collection of the *RD*. Thus, a direct connection has not been implemented so far, and may be a task for future work (see Section 7.3.4).

On the other hand, this fact does not mean that the *Micro-Visualization* was not usable inside *ECHO*. The *Categorical Metadata Micro-Visualization* allows users to visualize filters, set on the *language*-facet or the *provider*-facet, depending on the setting made in the *RD*. It also visualizes the ratio between the different values of the corresponding facet. If the facet, currently represented within that *Micro-Visualization*, is used as a filter inside *ECHO*, the filter is also applied within that visualization. Figure 4.12 shows the "*category*"-Micro-Visualization with applied filters.

4.2.4 Collection Summary (RingRepresentation)

The following requirements were defined in Sections 4.1.2.2 and 4.1.2.3:

- (1) the recommendations' facets need to be visualized in such a way that the user can easily interpret and distinguish them.
- (2) a collection's *character*, derived from its results' facets should be visualized to allow an appreciation on first sight, and to show differences to other collections in an intuitive way.
- (3) further, filtering the facets (see Section 4.2.3) has to be possible to find recommendations of interest easily.

These three requirements lead to considerations of a single functionality, which realizes a combined solution. At first, the ideas of the *RD* were picked up by providing an interactive tool in the sidebar that shows data, but also allows the user to interact with the visualization by filtering it.

However, during the process of designing the collection's layout (see Section 4.2.1.2), the idea arose of using the space in the graph between the collection's center node and its recommendation sub nodes.

The resulting approach (see Figure 4.14) underlies the following basic concept:

- Each of the facets is visualized as a ring around the collection's center node.
- All values are visualized as labeled ring segments.
- The recommendation nodes' connections to the center node cross the ring segments of each facet. Each crossing represents the exact value of the recommendation node's facet data, which fulfills requirement (1).
- The ring segments need to be clickable to enable a filtering (see Section 4.2.3) of the results, which satisfies requirement (2).

- The overall visualization of the rings and their segments should allow the distribution and diversity of the facets' values (the collection's *character*) to be visible to fulfill requirement (3).

As one of the most eye-catching and outstanding features of *ECHO*, the *RingRepresentation* not only makes it possible to get the results' facet information, but also offers a way to filter the recommendations of all collections through simple mouse-clicks. It's background, usage, and features are described in detail in this section.

4.2.4.1 Creation & Appearance

When focusing a collection (see Section 4.2.2), the camera navigates in front of that collection and gives it the whole space on the canvas. As soon as the camera starts focusing on the collection, the *Ring-Representation* gets initialized.

Depending on a configuration order, all facet-values of the current collection's recommendations are collected to build up an internal **tree-structure**. Beginning with the first facet, the recommendations are grouped by their values. Each of these groups is sub-divided again by the second facet in the next step of the algorithm. This step is repeated until the separation is completed through all facets. The resulting tree-structure holds the facet-values as inner nodes and the recommendations as leaves. While each level of the tree describes the different values of a specific facet, a value could occur multiple times due to different parents.

Finally the tree is visualized as a *sunburst-diagram*: beginning with the center of the collection, a first ring consisting of ring-segments is created. Depending on the configuration, this ring describes the first facet. Each ring-segment corresponds to one of its values. The more recommendations that exist with this value, the longer the arc of the segment. After finishing creating the first facet-ring, the algorithm is repeated on each of the segments, where the sub-segments fill the space of their parent-segment.

The recommendations are attached as leaves in the last level of the tree. Each recommendation can be found by going through a unique path from the root to the bottom. The length of the *Ring-Representation*'s segments

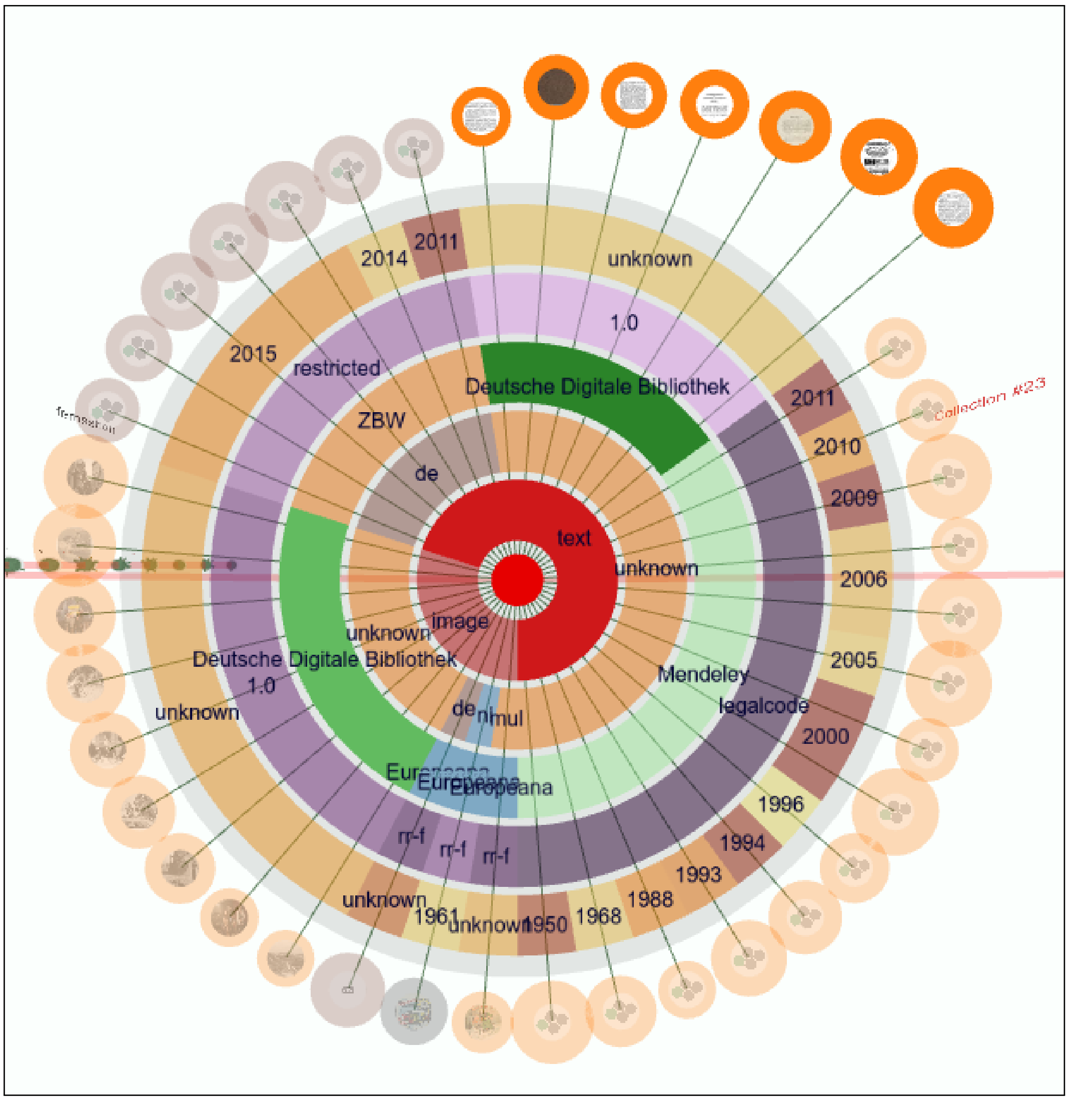


Figure 4.13: A typical *Ring-Representation* in *ECHO* with the facets *type*, *language*, *provider*, *license* and *year*. Filters are applied to get results that only contain "`text`" and have their source in the "`Deutsche Digitale Bibliothek`".

correlate with the number of elements they hold, and the sub-elements share the same angle of their parent-segment. Thus, it is possible to create a path to a recommendation by following a straight line from the center to an outer segment.

Therefore, the recommendations need to be moved to the matching position. This **reordering** is performed directly after creating the rings. Since the user needs to recognize that the order of the recommendation nodes has changed, this process is animated, and the nodes are moved along the shortest path around the collection.

However, the length of the recommendations' connections to the center node still vary according to their relevance, described in Section 4.2.1.3.

4.2.4.2 Colors & Labels

Each ring-segment has a unique background-color. There are two ways to set these **colors** using *ECHO*'s configuration:

- Each facet has a base color where its values vary in the saturation and brightness. Thus, the whole ring shares the same tint but the segments are still easily distinguishable.
- The colors for a specific value of a facet are set explicitly in the configuration. This option allows the user to follow the color-schemes of other *RD*-plugins, for example.

Each ring-segment also holds a **label** to allow an identification of the underlying facet-value. These labels are rendered using `IHQN.Text`-objects (see Section 5.7 for details about their implementation). Their center is positioned at half of the ring-segment's length. Since the **space is limited** and especially combinations between long facet-values and short ring-segments could cause overlaps, some preprocessing of specific labels are necessary:

The *license*-facet's value varies between simple names of the document's license and long URLs that refer to their license-description. Those **URLs are too long** to display, since they often need the whole width of the collection's visualization. Since there is a large number of different licenses containing different versions it is nearly impossible to map all those URLs to short names. The current solution is to use only the last part of the license's URL, separated by its slash-characters. As this short string often just describes the version of a license, it does not provide a lot of information to the user. Thus, this problem still has to be addressed in future work (7.3). Varying data from different providers is discussed in Section 4.2.3.1.

4.2.4.3 Filtering & Interactions

While the *Ring-Representation* allows the user to easily analyze the collection's data, it is also possible to use it to apply filters on the whole scene. Filters, which are described in detail in Section 4.2.3, are responsible for showing or hiding recommendations, depending on their facet-values.

While it is possible to set filters from the recommendation's detail info-panel, the *Ring-Representation* provides another opportunity to set and unset them. This is possible by clicking on the ring-segments, which each of represent a single facet-value. A click on such a segment leads to an immediate change of the filter and all affected collections in the whole graph. If a filter containing exactly the clicked value, was already applied, the filter is removed from the global filter-list. This means that the user can toggle a specific filter by clicking on a ring-segment.

The *Ring-Representation* can also help the user identify already applied filters. Even if the collection was just focused, matching filters can be recognized by a more saturated color of the corresponding ring-segment's background color. If the user defocuses the collection and the *Ring-Representation* is removed, the filters stay applied on the whole scene.

Figure 4.13 gives a sample of a collection with two filters set: One on the "*type*"-facet, with the value "`text`", and another on the "*provider*"-facet. This results in filtering documents that are text-files and have their source in the "`Deutsche Digitale Bibliothek`".

4.2.4.4 Benefits of the Ring-Representation

The full power of the *Ring-Representation* can be summarized as follows:

- **Finding results by facet-value-combinations**: by going through a path from the center to the outside of the *Ring-Representation* it is possible to find results easily. Navigating through that tree on a path of interesting values results in matching recommendations. Since the order of the facets is important in this case, the success of finding results depends on the facet-order in *ECHO*'s configuration.
- **Discovering a result's values**: the user might want to know details about a recommendation very quickly without changing the scope. Following the recommendation's edge, all of the document's facet-values can easily be found since the edge cuts each of them.
- **Filter-Feedback**: as described below, the *Ring-Representation* can be used to apply and remove filters easily. It also can provide visual feedback about the currently applied filters. If a facet-value that also occurs in the current collection is set as a filter, its ring-segment is highlighted. Thus, especially if filtering was applied in the current collection, the current filter can easily be recognized - and manipulated. Recommendations that are filtered out are shown as semi-transparent and *collapsed* (see Section 4.2.3).
- **The collection's character**: considering the *Ring-Representation* as the collection's overview, different characteristics can be read out of it. A first look already shows the user the **homogeneity** of the collection's facet-values. The longer the ring-segments are, the less different the recommendations.
  Furthermore, **relations between the facets** are also easily discoverable: for example, connections between the results' media-type and their language distribution can be found out without much effort.

4.2.5 Comparison Mechanism

Section 4.1.2.3 defines the requirement to allow the comparison of collections in the query history. The user already gets support in finding similar recommendations by the usage of filters (see Section 4.2.3). However, it also should be possible to compare whole collections to see if different search queries have resulted in similar results.

The concept of representing the results and different collections as a graph

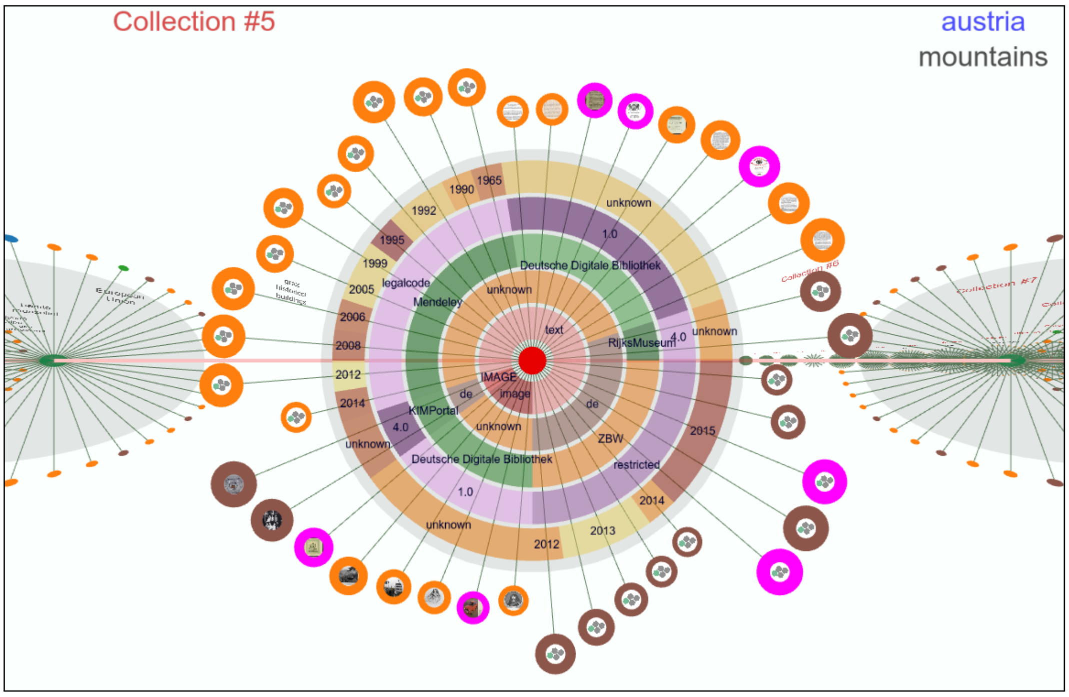


Figure 4.14: Detailed view of a selected query-collection. Results containing the keyword "austria" in their title are highlighted as purple nodes (see Section 4.2.1.2).

structure (see Section 4.2.1.1) led to the idea of showing similarities between multiple collections as connections inside the graph.

As mentioned in the requirements, the graph can contain the same document, represented by a recommendation node, multiple times. Thus, the concept of showing similarities between the collections is based on multiple occurrences of results. This section describes *ECHO*'s comparison approach in detail.

4.2.5.1 Comparing Multiple Collections

ECHO's comparison mechanism allows the user to compare all collections in the entire query-collection history or in a selected set of bookmark-collections. In particular, the user can choose a single collection to be compared with all the other ones.

Every single recommendation of this source-collection is compared with the results of the other collections. If one of them contains the same document as the one to which it is compared, they get connected. The same recommendations are not directly connected with the recommendation in the comparing collection, but they are linked to each other in a historical order. This means that if a result appears in two collections before the comparing one, those results are connected in a row: the *older* one is connected to the *newer* one which finally is linked to the recommendation of the collection that triggered the comparison. Figure 4.7 illustrates this mechanism.

In the case of visualizing search queries, this method allows the user to see **how the queries develop over the time** relating to the current collection. Depending on the number of connections, the user can intuitively recognize similarities between collections. As an example, the scene in 4.15 allows the user to understand that there is one single collection, that has a lot of the same results as the one currently being compared. To further investigate that collection, the user could focus on it with a single click (see Section 4.2.2).

The same results are connected through extruded splines in different colors. Using splines has the advantage of preventing sharp corners if more than two recommendation nodes are connected. This allows the user to follow the spline on a comprehensible path. A static color for each recommendation node also helps to distinguish different paths.

The splines have a relatively high diameter. This is caused by the fact that other collections may be far away. Combined with the fish-eye-effect of the camera (see Section 4.2.2) a thin line would not be visible in the distance.

4.2.5.2 Functionality

The comparison-method in *ECHO* can theoretically be used in every navigation level. Practically, it is only useful in the both outer levels that let the user navigate through the collections and investigate a single collection.

A comparison to a specific collection is activated by moving the mouse over the collection's center node. As long as the mouse is left hovering over the node, the splines are visible. To move across the scene while having a comparison activated, the user can use the keyboard's cursor-keys to navigate.

4.2.5.3 Future Work

Since comparing the collections and their results was one of the main motivations of *ECHO*, future implementations may focus on further comparison methods and improvements to the current mechanism. Section 7.3.3 discusses ideas for useful implementations in the future.

4.2.6 Bookmark-Collections

Section 4.1.2.5 defines the requirement of allowing the user to organize interesting results in their own collections, so-called bookmark-collections. The user has the option to build individual collections, stored as bookmarks, which contain selected results from different queries and a bookmark name. It should also be possible to visualize bookmark-collections, whereas it is not relevant if the collection was created inside *ECHO* or in the *RD*.

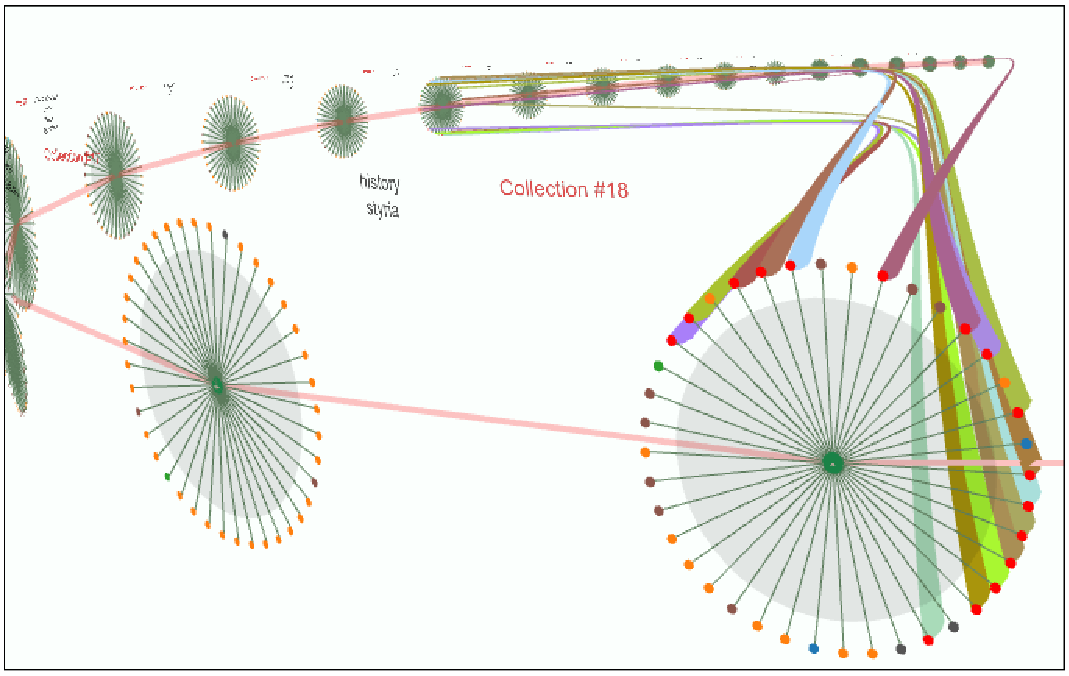
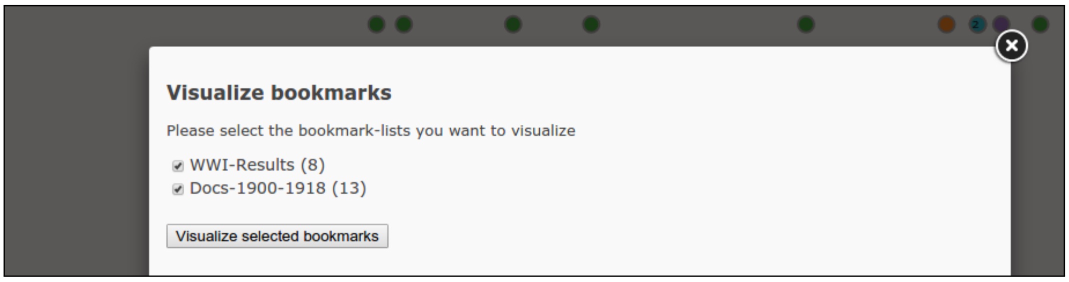


Figure 4.15: Comparing collections


Visualize bookmarks

Please select the bookmark-lists you want to visualize

- [✓] WWI-Results (8)
- [✓] Docs-1900-1918 (13)

Figure 4.16: Dialog for visualizing bookmark-collections

The bookmark-collections differ from query-collections in the following way: Typically, the *RD* shows results from a search query. Thus, this type of collection contains a **list of query keywords** beside the results, and can be found in the stored search query history. In contrast to a query-collection, a bookmark-collection does not contain a list of query keywords, but can be identified by a **unique name**.

4.2.6.1 Bookmarking a result

When focusing a recommendation, multiple buttons appear (see Section 4.2.2). One of them (the *star*-button), opens the native bookmark dialog of the *RD* (see Figure 6.12). The user then has the option to either select one of the existing bookmark-collections or to create a new one to store the active result.

4.2.6.2 Visualizing existing bookmark-collections

In the *RD*, the user has the choice to select whether the query history or multiple bookmark-collections should be displayed. In the latter case, a popup appears (see Figure 4.16) that allows the user to select a subset of bookmark-collections.

After selecting and pressing the "*Visualize selected bookmarks*" button, the collections are visualized in the same way the query history is shown. The order of the collections depends on the order in which they were created in the *RD* or in *ECHO*.

4.2.7 Fulfilling the Non Functional Requirements

In Section 5.2, several non functional requirements were defined. This section discusses how they were solved in *ECHO*.

4.2.7.1 Performance

Since both the *RD* and *ECHO* run as *JavaScript* Web Applications, performance is a big challenge. The following possible bottlenecks have to be considered:

- *JavaScript* code gets optimized through sophisticated compiling technologies in modern browsers. Nevertheless it does not approach the speed of C/C++ code.
- Several items are loaded through asynchronous calls from web resources. Possible delays must be kept in mind when designing the implementation.
- Rendering graphs may cause performance issues, since it may be quite CPU intensive.

To improve the performance related to rendering *ECHO*'s graph, *WebGL* technology was used that calculates the rendering process on the user's graphic card. Further details regarding *WebGL* and the used framework *THREE.js* can be found in Section 5.2.

Further improvements could be made by using a dirty-flag based design, which prevents calculations on unchanged objects during rendering (see Section 5.8 for details).

4.2.7.2 Low System Requirements

To allow as many users as possible to use *ECHO*, it should rely on existing state-of-the-art web technologies. Since every modern web browser supports *JavaScript* execution, and the *RD* in which *ECHO* was implemented as a sub project was written in *JavaScript*, this problems seems trivial. Only the support of *WebGL* in the user's browser is not assured. However, most of modern desktop browsers do support it nowadays.

4.2.7.3 Extensibility

It should be possible to extend *ECHO* for future work. As described in Chapter 5, *ECHO*'s code structure makes it possible to add further features and modify existing code easily through well documented classes and methods.

4.2.7.4 Usability

Animations help to track changes to elements and the navigation and thus improve the tool's usability.

Animations *ECHO*'s scene is not a static graph. Focusing a collection reorders the results around the collection's circle, for example, and applying a filter changes the result nodes' appearance. If some content of a scene changes, the user must be able to follow those changes. Therefore, an **animation-framework** was built for *ECHO*, that allows value-changes such as node positions, opacities, camera positions, etc. to be smoothly animated. Every kind of float-value and even objects containing multiple values can be animated through some simple parameters. This leads to a smooth user experience and improved usability when navigating and interacting with the graph, and thus helps to fulfill the requirement defined in Section 4.1.3.

4.2.8 Summary

As a result of the use cases in Section 4.1.1, the requirements of Section 4.1.2 and 4.1.3 were evaluated, and a detailed graph layout was designed (Section 4.2.1). The tool allows the user to navigate through the scene in three different levels (Section 4.2.2) and to filter (Section 4.2.3) and compare collections (Section 4.2.5). The *RingRepresentation* (Section 4.2.4) helps the user understand a collection's *character* and further allows them to apply filters. The bookmarking function (Section 4.2.6) allows to organize and visualize interesting results.

ECHO makes it possible to visualize a complex three dimensional graphs in the user's browser by means of support from the user's graphic card, through the *WebGL* technology.

5 Implementation-Details

This chapter provides a deeper insight into *ECHO*'s technical background and functionality.

The beginning of this chapter is structured as follows:

- Section 5.1 discusses the *RD*'s plugin-system, the used programming language and the tool's file structure, the used libraries and technical requirements.
- Since *WebGL* and its framework *THREE.js* are used extensively in *ECHO*'s implementation, they are discussed in the separate Section 5.2.

The next sections describe *ECHO*'s implementation in detail. They are ordered by the point of their occurrence in the process from starting *ECHO* up to its detailed usage. Beginning with the **Initialization** in Section 5.3 where *ECHO*'s loading as a *RD* plugin is described, similarities in the **code-structure** of *ECHO*'s scene elements (container behavior, dirty-flags etc.) is introduced in Section 5.4. Next, the **Collection** and *Recommendation* classes are described in the Sections 5.5 and 5.6. After a short introduction of the **Text-Elements** (Section 5.7), the **updating and rendering** process is introduced in Section 5.8 followed by discussing *ECHO*'s interactions in Section 5.9. The chapter concludes with an introduction of the configuration system used in *ECHO* (Section 5.10).

5.1 Environment

This section will give a short abstract of the used programming language and technology, the tool's file-structure, and the libraries used.

5.1.1 Plugin-System

ECHO was implemented within the *Recommendation-Dashboard* (*RD*), which represents different views on results of the *EEXCESS* recommender in a web-browser (see Section 3.5.3) as a plugin. Different to other existing *RD* plugins, the *ECHO* not only handles the current collection, but focuses on visualizing the entire *EEXCESS* search history or multiple bookmark-collections at the same time. For details on how the *ECHO* relates to the Plugin-System see Section 5.3.1.

5.1.2 Programming-Language & OOP-Design

ECHO was built for use within the *RD*, which is written in *JavaScript*, and can be accessed through different frontends in a client's web-browser. Thus, *ECHO* is written in *JavaScript* as well, since it is integrated directly into the *RD* as a plugin. Typical for web-applications written in *JavaScript*, *ECHO* also contains HTML and CSS-Files. Both are necessary to provide structure and a visual layout.

JavaScript was initiated as a Script-Language but nevertheless provides the possibility to perform object-oriented-programming (OOP) nowadays. Since *JavaScript* does not support classes in general, **prototyping** can be used to define, create and use objects. For easier understanding the term *class* is also used in this work for objects used in an OOP way. The usage of **namespaces** is also possible, although they are not explicitly specified in *JavaScript*. Therefore, simple objects are used for encapsulating code, as in a namespace known from other OOP-languages.

The following code-snippet demonstrates the usage of namespaces as they are used in *ECHO*, and a simple definition of an object with a method.

```
var ECHO = ECHO || {};
ECHO.MyObject = function () {
        //Constructor
};

ECHO.MyObject.prototype.someMethod = function(param){
        //Method implementation
};
```

5.1.3 File Structure

The code used to launch *ECHO* is stored in two locations inside the *RD*'s root folder: the plugin file for initializing *ECHO* is located within the `Dashboard/Plugins` folder. The functionality of the plugin is described in Section 5.3.1. The entire code of *ECHO*, however, can be found within the `WebGlVisualization` folder. Its structure is as follows:

- `css`: *ECHO*'s CSS file for layouting the used canvas and several HTML-Elements.
- `js`: containing *ECHO*'s *JavaScript* code in several subfolders holding about 50 files.
- `lib`: folder for the libraries used by *ECHO* (see Section 5.1.4 below).
- `media`: images, mostly icons, used within *ECHO*.

5.1.4 Libraries

ECHO makes use of some well-known *JavaScript*-libraries:

- **THREE.js**: The *WebGL*-API itself does not provide a lot of high-level operations on primitives, cameras, the entire scene etc. Thus, *THREE.js* is there to help perform manipulations and interactions on the scene much more easily by making methods for movements and transformations available, instead of using complex mathematical operations (also see Section 5.2).
- **jQuery**: this library is well known for easy manipulations of a web-page's HTML-DOM. *ECHO* makes use of some of its tools and also loads the *jQuery-Fancybox*-Plugin, which makes it possible to superimpose the current web-page with a window-like element, containing specific content (info-panel; see Section 4.2.1.3).

- **Underscore.js** is a powerful library with a lot of useful tools for such as filtering lists or providing `foreach`-like loops. Even if `_.each()` loops seem to be more aesthetic they should be used with caution because they are slower than typical native *JavaScript* iterations over arrays.
- **Modernizr**: although this library was originally built to test and load files depending on the user's browser, it is used both in the *RD* and in *ECHO* to load files dynamically when needed. Originally *Require.js* was used to support dynamic loading, but was replaced by *Modernizr* due to problems running the library in both systems simultaneously.
- **lz-string**: this library is responsible for compressing the query-result data before storing it inside the (size-limited) *localStorage* with the goal of being able to store more query-results (see Section 5.3.3).

5.1.5 Technical Requirements

To be able to run *ECHO*, it is necessary to use a web-browser which supports both *WebGL* and *localStorage*. While the latter is provided by all modern browsers nowadays, the support of *WebGL* also depends on the user's hardware. Older PCs or cheaper notebooks often do not have a graphic card offering Hardware-Acceleration.

Even if a lot of calculations are performed by the GPU, *ECHO* consumes more CPU than other visualizations. Thus, the user's hardware should at least be *average* for smooth usage of *ECHO*. More discussion of the tool's performance can be found in Section 7.3.1.

5.2 3D Rendering

Existing *RD*-plugins use 2D-visualization libraries which are sufficient for visualizing a single collection. But, *ECHO* should be able to handle dozens of collections at the same time, while preserving the overview and preventing a **visual overload**. Furthermore, each of the history's collections may contain up to 100 results. That fact may lead to **performance issues** if

2D-libraries were used for the visualization, because they typically just *paint* on an HTML-Canvas. Thus, they do not use the graphic card's hardware-acceleration.

These two challenges resulted in the usage of ***WebGL*** (Congote et al., 2011), a web-browser implementation of the well-known *OpenGL*-API (Woo et al., 1999). The possibility to easily use three-dimensional renderings offers more freedom in visualizing information, and the accessibility of the hardware-acceleration helps to prevent performance problems. Similar to *OpenGL*, good frameworks for *WebGL* exist that make the usage of primitives, cameras, scenes etc., much easier. Finally, the decision was made to make use of ***THREE.js*** (Danchilla, 2012) - one of the most used *WebGL*-frameworks.

THREE.js holds all objects like primitives, cameras, lights, etc. in a instance of `THREE.Scene`. This object is held by the `ECHO.WebGlHandler` which is responsible for the basic *THREE.js* objects.

5.2.1 Camera

WebGL allows the use of different kinds of cameras to provide the right projection and view of the scene. Cameras can be orthographic, for instance, but a perspective projection is also possible. Since *ECHO* makes use of a three-dimensional graph, a perspective camera could visualize the depth of the scene in a better way. Thus, a `THREE.PerspectiveCamera` is added to the `THREE.Scene` object.

When creating a perspective camera, some parameters must be set (also see Figure 5.1):

- **fov**: The camera's (vertical) field of view in degrees. To provide a wide camera-angle in order to see most of the scene at once, a relatively large value of 120° was taken for *ECHO*.
- **aspect**: The aspect-ratio of the camera. Thus, the scene's width divided by its height is taken.
- **near**: Distance of the *Near*-Plane which is close to the camera's *eye*.

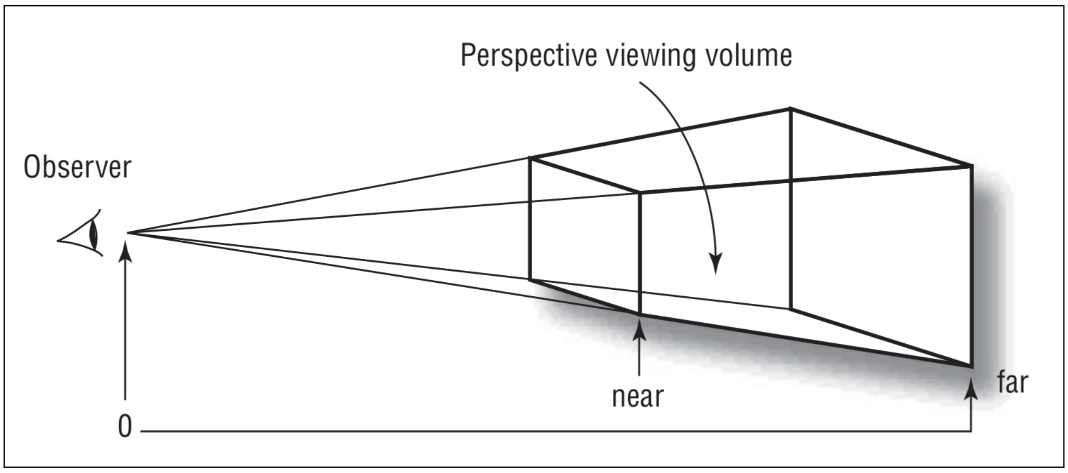


Figure 5.1: Schema of a perspective projection in *WebGL* (Wright, Lipchak, and Haemel, 2007, p. 86)

- **far**: Distance of the *Far*-Plane. A too large value could cause flickering or other unwanted effects due to inaccuracy. A too small value would cut objects that are too far away.

5.2.2 WebGl-Renderer

The entire rendering process in *WebGL* is performed by an instance of the `THREE.WebGLRenderer` class. Every time the `ECHO.WebGlHandler.preRender` method is called, the `preRender` method of the `THREE.WebGLRenderer` starts, taking the `THREE.Scene` and the `THREE.Camera` as parameters (further details in Section 5.8). At each render-step, the power of *WebGl*'s Hardware-Acceleration comes up by letting the GPU calculate the graphical output that is finally painted on the canvas element.

5.3 Initialization Of ECHO

Using the *RD* allows the user to analyze the current results of an *EEXCESS* query through clicking on one of the Visualization-buttons on the right hand-side of the main window. After clicking on the button for *ECHO*, the search-history is loaded from the database. Another method to access *ECHO* is to select bookmark-collections to visualize. This can be performed by using a button in the left column of the *RD* to further select one or more existing bookmark-collections to visualize. The following section gives an overview of how the initialization of *ECHO* as a plugin works.

5.3.1 RD Plugin Architecture

As mentioned in Section 3.5.3.2, *RD*-plugins are configured in a single file inside the `plugin`-folder and contain a class with requiring the following methods:

- `initialize`: is called directly after the *RD* was loaded. For example, *ECHO* creates the button for visualizing the bookmarks at this stage.
- `finalize`: used to clean up the visualization space after unloading the current visualization by removing some CSS-classes or resetting some flags, for instance and to restore the *RD*'s result-list.
- `draw`: this method is the link between the *RD* and *ECHO* when loading the plugin. It calls the static method `ECHO.InitHandler.init()` that is responsible for loading all necessary files and creating the scene.

After calling the `ECHO.InitHandler.init()` method, a container for adding the rendering-canvas is added to the HTML-root-element provided of the *RD*'s plugin-system. This container also holds a loading-GIF and a message to communicate the loading-status to the user.

5.3.2 Dynamic File Loading

Before creating the entire scene that contains the graph, several internal and external *JavaScript* files need to be loaded within the `ECHO.InitHandler.init()` method. Since each *RD* plugin requires a lot of different files, it loads them on demand when it is initialized the first time. *ECHO* requests all its internal classes and external libraries right before rendering. At first all external libraries (see Section 5.1.4) are loaded, followed by internal configurations, tools, the database handler etc. up to the entire file containing the `ECHO.Scene` class.

All necessary files are loaded using the ***Modernizr***-plugin (Watson, 2012), mentioned above. The files are loaded in an order that fulfills their internal dependencies. *Modernizr* provides possibilities to define callbacks that are performed after the correspondent file is loaded. Nevertheless, dependency-errors when initializing the *JavaScript* classes may occur even if the next file is not loaded until the `complete`-callback is performed. This is caused by the fact that a file is indeed loaded, but the contained *JavaScript* class still may not be initialized by the browser's *JavaScript*-engine. Adding a minimal timeout of a few milliseconds during the `complete`-callback avoids such problems since the client's browser has enough time to interpret the file before loading the next one.

5.3.3 Storing Query-Results

Since *ECHO* visualizes the entire query history, all queries and their results from the *EEXCESS*-recommender need to be stored in a database. If the user triggers a new query and the results arrive, the `onDataReceived` method inside the `starter.js` is getting called. Inside this method, the `saveReceivedData` was placed which uses the `QueryResultDb` class to save the results in the browser's *localStorage* database. `QueryResultDb` is the only class in this project, besides the plugin-class, which was placed outside the `WebGlVisualization` folder because it is not called within the plugin, but rather directly inside the `starter.js`.

Before being up to store the results in a database, it was necessary to decide which technology should be used. Since the *RD* is a client-side application, there are no meaningful possibilities for saving the data inside a server-side *SQL* database, as would be common for server-side web-applications.

It was necessary to choose between the following APIs, used for client-side data-storing within the browser:

- **Web SQL**: The *Web SQL* database was made for storing data using a variant of SQL. Since it is not supported by *InternetExplorer* or *Firefox*, and work on its *W3C* specification was stopped years ago, it was not considered for use in *ECHO*.
- **LocalStorage**: *LocalStorage*, or *DOM Storage*, makes it possible to store data within a simple key-value system. A big advantage over other systems is that it has been widely implemented in all modern browsers for years. But, *localStorage* also has a big limitation: the size of data that can be stored by one domain is usually limited to about 5-10 Megabyte depending on the browser used.
- **IndexedDB**: *IndexedDB* is a more complex and powerful database API. In contrast to *localStorage* its usage is more complicated due to its asynchronous design, but it offers indexation and much more space. *IndexedDB* is relatively new. Thus, it is not yet supported by all browsers, and is only available in newer versions.

Due to the fact that *localStorage* was already used in the *RD* for storing the user's bookmark-collections, the decision was made to use this API. It was easier to handle and has the big advantage of its wide support.

To avoid problems with the API's storage-limit, the idea of compressing the data came up. Therefore, the *lz-string*-library[1] was included, which uses a *LZW*-compression algorithm to reduce the size of a given string. Finally it was possible to compress the *JSON*-string of the result-data before storing it. While it was possible to save the results of a few dozens of queries without any compression, the usage of *lz-string* allowed hundreds of queries and its results to be stored, and hence solved the big disadvantage of *localStorage*.

Compressing the results is no guarantee of completely avoiding space problems inside the *localStorage* database. The decision was made, that it was better if an old entry was deleted to free space, instead of not being able to save new query-results. Thus, when catching `QuotaExceededError` Exceptions while storing the results, the oldest set of results in the localStorage is deleted. That means that a whole collection and its result do not appear in *ECHO* anymore. But, as mentioned, that only occurs if there are already hundreds of collections stored.

[1]https://github.com/pieroxy/lz-string

When reading the results from the *localStorage*, the stored data can easily be decompressed by *lz-string* again to get back the *JSON*-object. Finally, there were no significant performance issues measurable while using this compression-method.

5.3.4 Creating the Scene

After all files are were successfully loaded, the static `ECHO.InitHandler.initScene` method is responsible for creating the scene, and for adding the collections to the graph, depending on whether they come from the search-history-database or from bookmark-collections.

If the scene should visualize a set of bookmark-collections, their keys are passed to the `ECHO.BookmarkHandler` which loads the data from the browser's *localStorage*, where the *RD* saves the bookmarks and creates collection nodes and recommendation nodes that are injected into the scene.

Otherwise, if the query-collection-history needs to be visualized, the `ECHO.DbHandlerLocalStorage` has to load the stored search-results from the *localStorage*.

Independent from their data sources, both methods underlie the same procedure to successfully create a scene with collections and results. The procedure to visualize a collection is as follows:

At first, an object of the type `ECHO.Collection` is created and some metadata, such as the title, are set. The title may be the query-string or the name of the bookmark. This object is called ***Collection*** (see Section 5.5) in *ECHO*. Each *Collection* has a parent-collection, except if it is the very first one to visualize. When visualizing query-collections, the last query-collection before the actual one is set as parent. When showing bookmark-collections, the parent does not have such a significance and just describes the position inside the graph. Thus, the order appears as the selected bookmarks are stored within the *localStorage*.

After creating a collection, objects representing the query's results are created. Each of the results is stored as objects of the type `ECHO.Recommendation`. Each ***Recommendation*** (see Section 5.6) holds information about the entire result including the *facets* (*provider*, *language*, *license*, *date*), title, preview-image, and URL but also its weight inside the collection and its keywords. At the end, a *Recommendation* is added to its *Collection*.

Finally, the *Collections* are added to the *ECHO.Scene* object which not only holds the graph, but also references every other component used by *ECHO*. When creating the *Scene* object, instances of the following classes are created. Details are described in further Sections:

- `ECHO.NavigationHandler`: responsible for navigation within the graph.
- `ECHO.FilterHandler`: manages the application of filters on the collections.
- `ECHO.RecDashboardHandler`: link between the *Scene* and (outer) HTML-elements in the *RD*.
- `ECHO.WebGLHandler`: holds the *THREE.js* scene, camera and renderer. Responsible for the basic *WebGL*-setup. For further details see Section 5.2.2.
- `ECHO.InteractionHandler`: used for handling different interactions like mouse-clicking and dragging or keyboard actions.
- `ECHO.CollectionPosCircular`: calculates the *Collections'* positions.
- `ECHO.Forms`: creates HTML-Forms, used for filters and bookmarks for example.
- `ECHO.Animation`: animation-Framework used by registering animations with different parameters, defining callbacks etc.
- `ECHO.DirectCompare`: used for comparing similarities of neighboring collections.
- `ECHO.RecConnector`: connects *Recommendations* that represent the same result by rendering splines over the whole scene.

5.3.4.1 Animation-Loop

After creating all those components, the graph needs to be initialized by calling the `ECHO.Scene.initCollectionNetwork`.
The `ECHO.CollectionPosCircular` calculates the positions of the collections and their connections. Finally, the whole scene is shown, and the *ECHO.Scene.animation*-method is triggered for the first time. This method

performs a loop as long as *ECHO* is running. For more details on the animation-loop, see Section 5.8.2.

5.4 Code-Structure Of Scene-Elements

There are a lot of classes that represent a visual element inside the scene, like collections, recommendations, different connections, the *Ring-Representation*, its sub-elements, etc. All of these classes share a similar structure and some principles. Those are summarized in the following section.

5.4.1 Container Behaviour

Many objects in *ECHO* work as a logical container for one or more visual components (*THREE.js*-primitives). Values like the position or rotation-degrees, are stored outside the primitives, and are applied during the rendering call if necessary. All of these values, responsible for the visual appearance including the *THREE.js*-primitives, are encapsulated in a container object, called `vis_data_`. This encapsulation prevents from mixing up logical/meta/semantic-data with information necessary for the visualization, and from misusing the *THREE.js*-primitives with wrong values. Another benefit of holding the values outside the primitives is that sometimes reproducing values like a rotation or a relative position on a circle is impossible or at least hard to calculate precisely. If the values are stored outside the primitives, easier access to them is possible. As a sample, in Section 5.5 the `vis_data_`-object is described in detail.

5.4.2 Encapsulating Variables

Most variables inside the `ECHO`-classes are defined as private, and thus, when inspecting the code, a lot of getter- and setter-methods appear. Even if *JavaScript* does not support a strict separation between public and private members inside the *ECHO* project, direct access to the variables of most objects was avoided as much as possible and the usage of getter and setter was preferred to guarantee a good code quality, by ensuring control over the values when they are accessed from outside.

5.4.3 Dirty-Flags And Update

Instances of the *ECHO*-classes that represent visual elements, may have a lot of values that could change and then affect the *THREE.js*-objects, but also cause computations, each time the own `preRender`-method was called. To prevent unnecessary computations, each of those objects holds a dirty-flag that ensures that the `preRender` method only performs if the flag is dirty, which happens only if specific values (positions, textures, visibilities, ...) have been changed since the last `preRender`-call of the object. This flag also guarantees that sub-objects (like `ECHO.Recommendation` objects inside a `ECHO.Collection`) are not rendered if the dirty-flag of the parent is not set, because each `preRender`-method calls the `preRender`-method of its sub-objects if necessary.

This means, that changing a sub-object's data through a setter method must ensure a dirty-flag change on the parent-object. This is usually done inside the corresponding `setIsDirty()`-method, which triggers a dirty-flag up to the object's parent's dirty-flag-setter.

5.4.4 Initialization

The visual objects in *ECHO* have a method (mostly `initGlNode`) where all *THREE.js* objects are created and initialized. Those objects are finally added directly to the `THREE.Scene` or to a container object, which groups several objects together to easily perform transformations on more than one object.

5.5 Collections

The `ECHO.Scene` object holds each loaded collection. They could be the results of the *EEXCESS* query history (query-collections) or several loaded bookmark-collections. As described in Section 5.4, the `ECHO.Collection`-class acts as a container for its primitives but also contains other sub-objects and methods described below.

5.5.1 Functionality

In the `ECHO.Collection`-class's constructor, which passes the metadata of the collection, the initialization of the sub-elements is called and the type of positioning of the recommendations is set (`ECHO.RecommendationPosDistributed`).

Since all *WebGL*-elements within the collection should be transformable at once without touching or calculating each element, a `THREE.Object3D` container is created that holds all element. To move the whole collection, for example, only the container needs to be transformed. Every sub-element stays at its relative position within that container. This container is initially filled with the following *ECHO* elements: center node, the plane, and the text-labels, including the collection-name and keywords (see Section 5.5.2).

The collection has methods that can control sub-elements on several interactions. It can:

- toggle the *Ring-Representation*
- create and visualize splines from its recommendations to other recommendations inside the scene
- focus the camera on this collection
- turn the visualization of its recommendations' relevance on and off

As can be seen, the `ECHO.Collection` has powerful possibilities to change the visualization but does not affects the logic too much. It's main purpose is to provide functions to call from outside and to hold and update its recommendations.

The `preRender`-method of the collection calls the `preRender`-method of all its elements inside the `gl_objects`-object (see below), but also on its recommendations and a possible *Ring-Representation*. The decision, if a rendering of the sub-elements is necessary, is not made by the collection. It just calls the sub-element's `preRender`-function which decides if the render-calculations should be performed on the basis of its object's dirty-flag.

5.5.2 Visualization Data

Data and objects representing the node and its visual appearance are again encapsulated in a sub-object (`vis_data_`) of `ECHO.Collection`. This data, made up of the following elements, also gives an example of how the `vis_data_`-object would look in other container-classes:

- **position**: Absolute position inside the 3D-space.
- **initial-position**: Calculated by `ECHO.CollectionPosCircular`.
- **rotation**: Degree-value of rotation around the y-axis. It needs to be stored, since it can not be reproduced after applying transformations on the objects.
- **initial-rotation**: Calculated by `ECHO.CollectionPosCircular`.
- **gl_objects**: Container holding the entire *THREE.js* objects.
  - **center_node**: `ECHO.CollectionCenterNode`, containing a circle object representing the center of the collection.
  - **parent_connection**: `ECHO.ConnectionCollectionCollection` holding a `THREE.Line`-Geometry to visualize the connection to a potential parent-collection.
  - **plane**: `ECHO.CollectionPlane`-object holding a `THREE.CircleGeometry` to show a semi-transparent plane between the inner node and the recommendation nodes.
  - **compare_bar**: Optional object (`ECHO.DirectCompare`) for displaying a bar containing the percentage of how much the current collection has the same results as the parent collection.
- **mesh_container**: A simple `THREE.Object3D`-object that acts as a parent object for the *THREE.js* objects held by the classes above (see Section 5.5.1).

Furthermore, the `vis_data_` object contains some flags used for the visualization.

5.6 Recommendations

Recommendations represent references to documents retrieved by *EEXCESS* belonging to a search query or a manually created bookmark-collection and are implemented within the `ECHO.Recommendation`-class. The following section deals with the recommendations' data, their visual appearance, and the possibility to retrieve and use their information within a separated detail-window.

5.6.1 Data Held by a Recommendation

Since the retrieved documents are not downloaded or processed by any of the *EEXCESS* modules, only some metadata is available to perform visualizations, comparisons or interactions.

The following relevant data can be found within the results retrieved from *EEXCESS*:

- **title**: The document's title.
- **description**: Short description of the document.
- **icon**: Optional link to a small image that can be used as thumbnail.
- **URI**: Link to the document for reading or viewing it.

Additionally, the so called ***facets*** are retrieved. These metadata consist of the following values:

- **language**: Two-character-code of the language the document is written in.
- **license**: Any kind of string describing the license of the document. Since this is often just a URL to the underlying license-description, visualizing the license inside the *Ring-Representation* (Section 4.2.4) is difficult due to its length. A perfect representation of the language string is still an open task in future work (see Section 7.3.2).

- **provider**: A provider delivers the results to *EEXCESS*, depending on the query. The provider can be identified through this string, which could be "`Deutsche Digitale Bibliothek`" or "`ZBW`" for example (see Section 3.2).
- **type**: Describes the type of document which could be "`text`", "`image`" or "`sound`", for instance.
- **year**: Should hold a date related to the document. Since there is no clear specification regarding how this date should look like, the values may vary from "`2015`" to "`2014-10-02`". Many documents do not even have a date set ("`unknown`") and some dates may have wrong values ("`17881788`").

If a facet is empty its value is set to "`unknown`".

5.7 Text-Elements

Since *THREE.js* does not provide native methods to render text inside the scene, it was necessary to implement a particular solution. The `ECHO.Text`-class provides a simple method to create text-elements with variations on their appearance like colors, background, size etc. When creating an `ECHO.Text`-object, a DIV-element with the specific visual values, set as CSS-properties, is created with the text as content. It is rendered on a HTML-canvas element, and the result is used as a texture of a `THREE.PlaneBufferGeometry` element, which is then added to the scene.

Since the text is not available as a vector-graphic, but rather as a rendered image, different distances of the camera to the element could lead to annoying pixel-artifacts on the scene. Therefore, each text-element can be created in a variable resolution. A factor value makes it possible to render the element with a bigger size, which is afterwards reduced by transforming the rendered element.

It is also possible to define the visual appearance of the text-element while hovering (a different background-color, for example), or to define methods for interactions like mouse-over or a click (see Section 5.9).

5.8 Updating and Rendering

In *ECHO* most of the objects, containing *THREE.js*-elements, hold a `preRender`-method. They are used to bring internal values like positions, flags, etc. on the screen through a process described in this section.

5.8.1 The `preRender`-Method

If properties of scene-elements are set, changes on its visual appearance may also have to be made. For example, if the rotation of a collection is set through its setter, a single float-value is modified at this moment. Further, the collection's dirty-flag is set to `true`. This does not result in a visual change yet: every time the `preRender`-method of the object is called, the value of the dirty-flag is checked. If it is not set, the method is left immediately.

However, in the case of the example above, the `preRender`-method continues. It has to apply the single rotation-degree-value on the *THREE.js*-object, that holds all primitives. The `preRender`-methods are also responsible for triggering the render-process to potential sub-elements. After finishing, the dirty-flag is reset to `false`.

5.8.2 The Animation-Loop

The rendering first starts through calling the `ECHO.Scene.animate`-method. It runs in a loop until the scene is deleted.

To prevent unnecessary calculations, while the browser window or tab is not active, the method not only calls itself, but also uses a `requestAnimationFrame`-call. This ensures that the *JavaScript*-engine only recalls the method again if the window is active.

The `ECHO.Scene.render`-method is called inside this method.

5.8.3 Rendering the Scene

After calling the `ECHO.Scene.render`-method, and checking the dirty-flag (see Section 5.8.1), the `preRender`-method of each collection is called. Each collection performs its calculations and further necessary manipulations on the *THREE.js*-objects it manages. The collections call `preRender`-methods on each of their sub-elements (recommendations, labels). Those objects may perform the same process on their sub-elements, etc.

After all *THREE.js*-elements are set in the way they should be, the `ECHO.WebGlHandler` calls the `preRender`-method of the `THREE.WebGLRenderer` object it holds. This finalizes one step of the animation-loop by performing the entire *WebGL*-rendering.

5.9 Interactions

The `ECHO.InteractionHandler` is responsible not only for forwarding mouse-events to the corresponding elements, but also for processing key-events and for transferring values to the `NavigationHandler`. If the mouse is moved over the scene, the mouse-wheel is used, or a click is performed, the `InteractionHandler` uses an instance of `THREE.Raycaster` to retrieve all objects that are in an area of influence of the mouse-cursor.

Those *THREE.js* objects may hold an object that was created by their *ECHO*-parent-object. It refers to different methods (mostly held by the parent-object), such as `handleClick` or `handleMouseover`. Depending on the type of interaction, and if such a method was set, it is called to perform an interaction on that object.

5.10 Configuration

The visualizations, animations, and the performance of *ECHO* depends on hundreds of variables. One principle while developing was to prevent leaving any value that is used for configurating *ECHO* inside the code.

Variables like sizes, positions, as well as colors etc. can be found within the `ECHO.config`-object.

6 Case Study

6.1 Research for Historical Images

This chapter will demonstrate the benefits of *ECHO* through the following scenario: the user Alice, who is working as a librarian, is doing research on a historical topic (*WWI*) over a long period of time. After navigating over many documents and retrieving a lot of recommendations through several searches, she decides to find and collect images of *WWI* locations in Europe. Instead of attempting to generate similar searches again, she wants to make use of the already existing queries and results, which are conveniently tracked and saved by *ECHO*.

6.1.1 Initial Position

Alice has been investigating the *WWI* events for a long time. Most of the topics relate to *WWI*. However, other searches from other fields also occur. Since her research has taken place over a long period of time, she no longer remembers every query or result. Alice just previously performed a search with the keyword *"infanterie"* (infantry) (Figure 6.1) . Instead of performing new searches, she wants to investigate the already stored results. Therefore, she starts *ECHO* to access past queries and their results (Figure 6.2).

6.1.2 Finding Related Query-Collections

Alice wants to know if results from the current query (*"infanterie"*) also occur in past queries. Therefore, she hovers over the center of that query-collection.

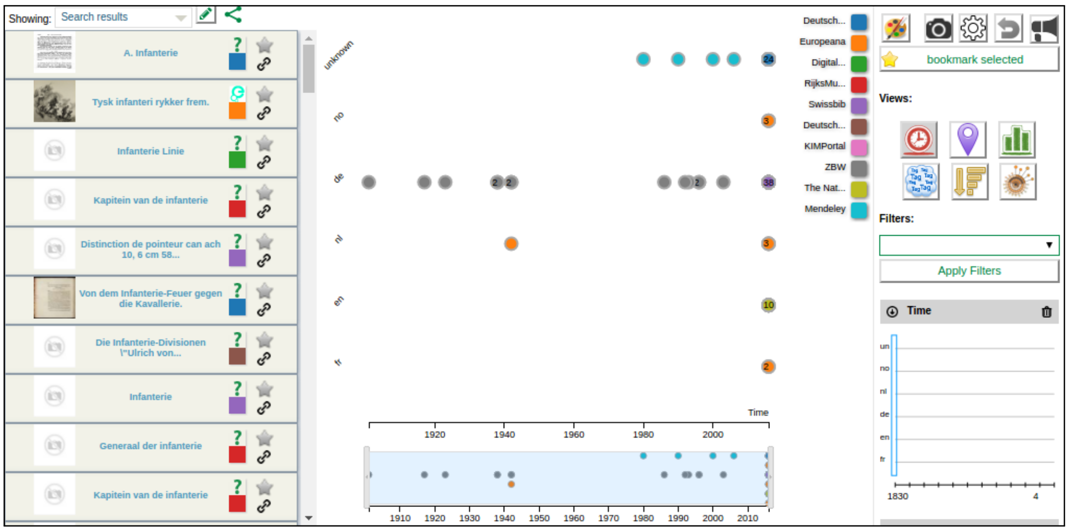


Figure 6.1: Current search results on the keyword *"infanterie"* (infantry) in the *RD*.

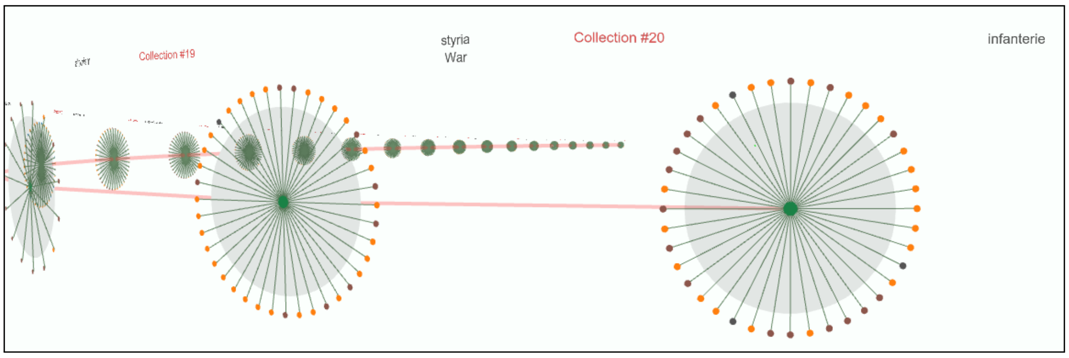


Figure 6.2: Several searches over different topics with a focus on *WWI*. The last search was *"infanterie"* (infantry).

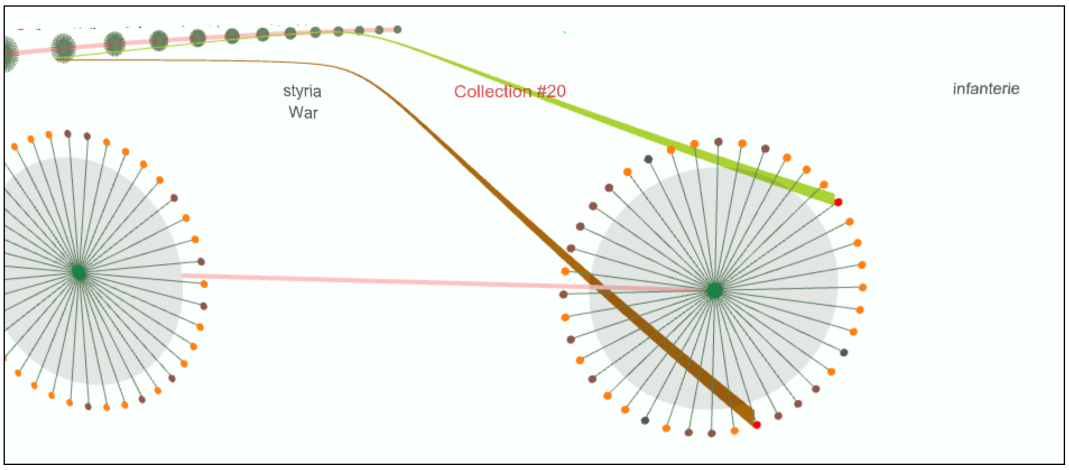


Figure 6.3: Hovering over the query-collection's center activates the comparison: splines to same results in previous query-collections are shown.

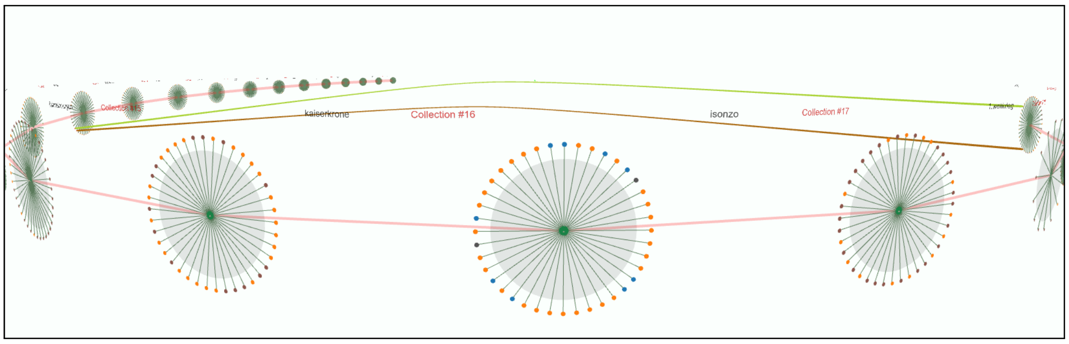


Figure 6.4: After hovering, the keyboard-cursor-keys allow movement around the scene from the last query-collection (right) to the collection containing two same results (left).

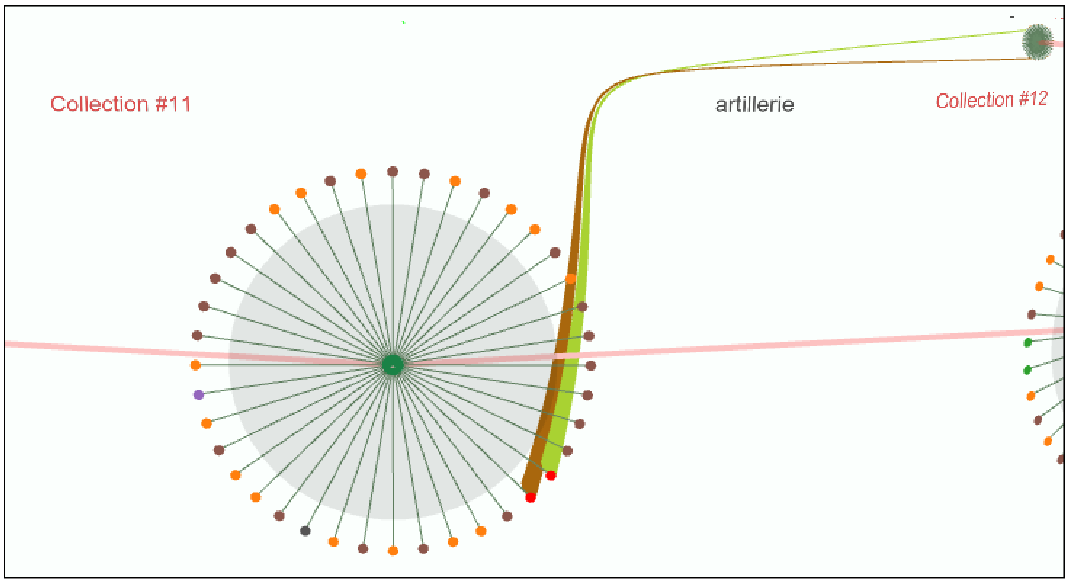


Figure 6.5: After moving to the collection with two same results

In fact, two results also occur in another query-collection (Figure 6.3). Since both results end up in the same query-collection, it may be interesting to investigate the previous result set. Alice could now click directly on the query-collection where the splines end, or she could move to it while leaving the splines intact. She decides to move through the keyboard's cursor-keys to see which results of the other query-collection the splines connect to (Figure 6.4). That brings her to an interesting query-collection (Figure 6.5), generated by the query-term *"artillerie"* (artillery).

6.1.3 Investigating an Interesting Query-Collection

Due to the interesting query-term (*"artilliere"*), Alice decides to investigate that query-collection further. She clicks the query-collection node to activate the *Ring-Representation* (see Figure 6.6). After investigating its results, she wants to know if there are further related queries in the search history. By hovering over the collection's center, she can see the two matching recommendations from the base-collection before (Figure 6.7). On the top-

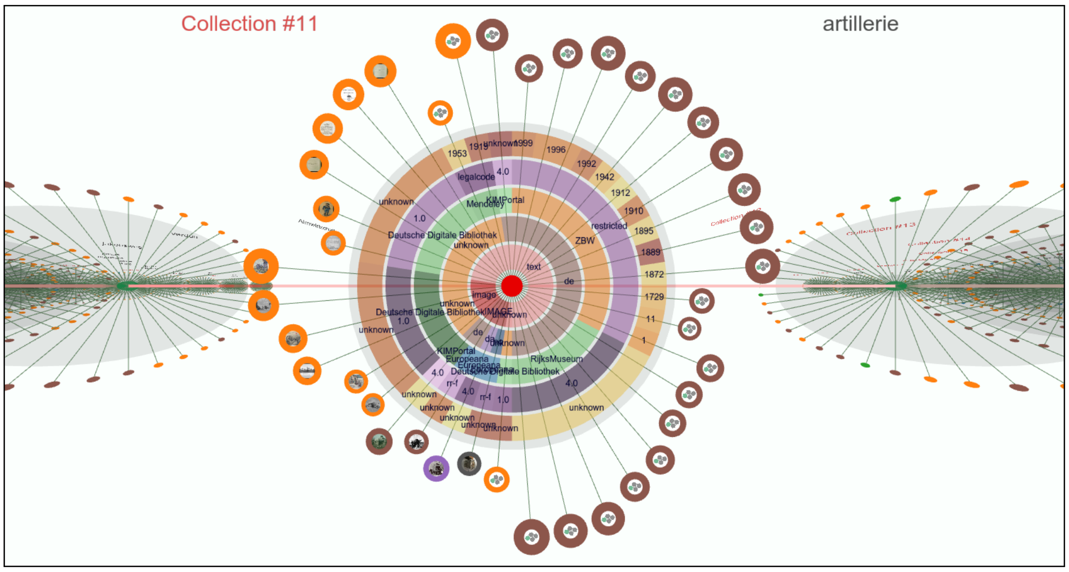


Figure 6.6: Focus on a query-collection to investigate

right of the query-collection, two results are linked to another result set. They are those that also occurred in the later *"infanterie"* query-collection.

Alice discovers a third recommendation on the bottom-left that links to another query-collection. She decides to take a closer look. Therefore, she clicks on the recommendation node, which is immediately focused on (see Figure 6.8). The preview-image, which shows a theater of war, seems to be very interesting. Since she hopes to find similar documents in the other query-collection, Alice decides to follow the link. Since she wants to get an idea of where in the history the other query was performed, she wants to return to the overview first by scrolling down with her mouse-wheel two times, she first reaches the level of the collection-focus, and then the outer level that allows free navigation. To find the desired query-collection, she hovers over the center of the *"artillerie"* collection again. The connections appear and she can clearly see the one that leads to the still unknown one (Figure 6.9). Alice wants to focus that query-collection and clicks on it.

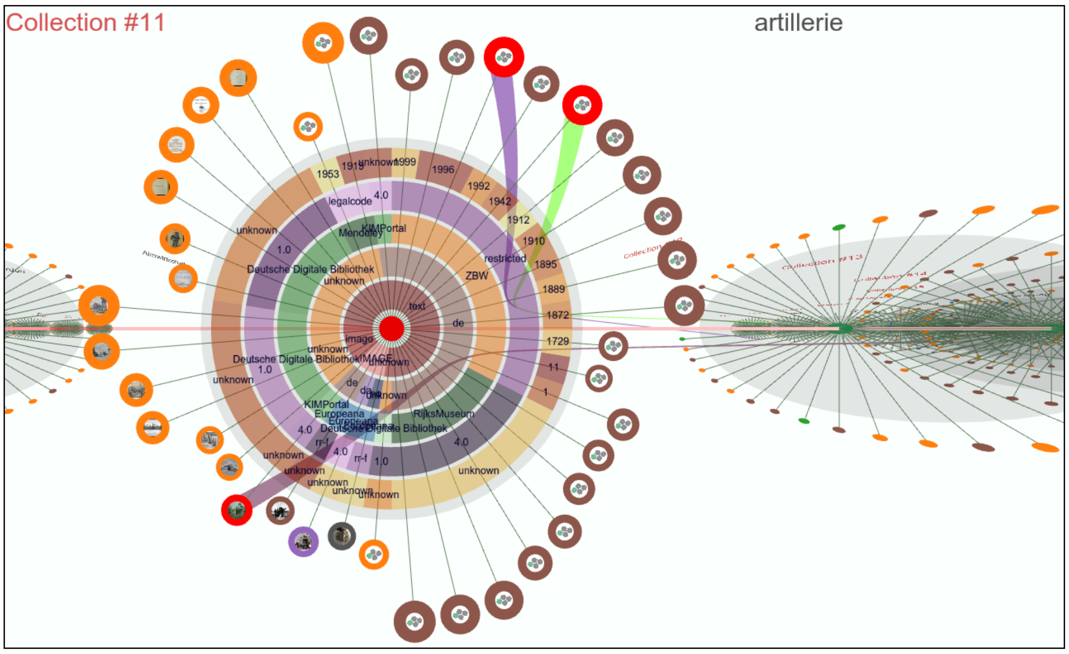


Figure 6.7: After hovering the query-collection's center node, three documents are linked to other queries. Two (upper right corner) link back to the previous viewed query-collection, one (bottom left) to a still unknown query-collection.

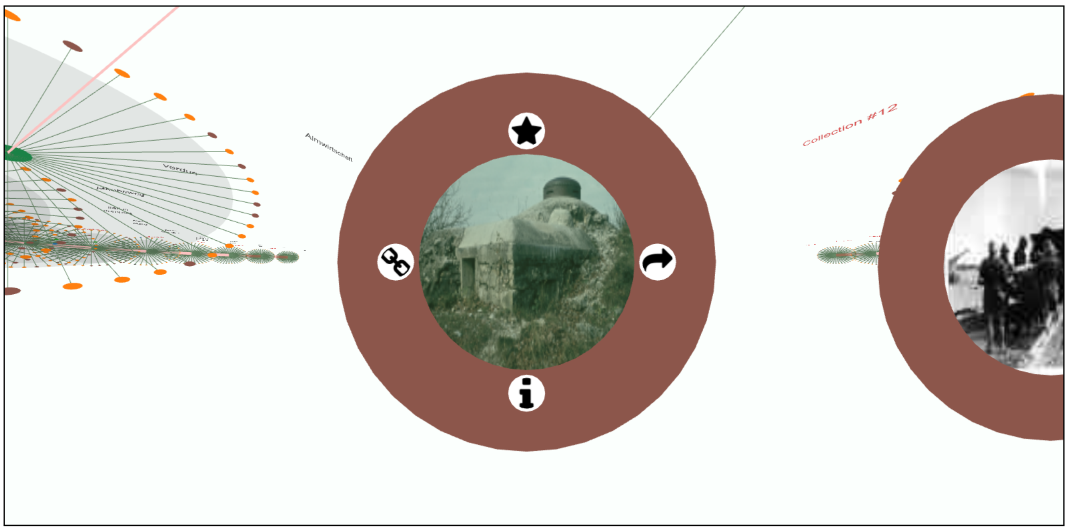


Figure 6.8: Focusing on a recommendation by clicking on it zooms in and shows a preview of the underlying document and several buttons: *Reference* (left) for external usage.
textitInfo & Filter (bottom) for opening the detail window, *Open* (right) for opening the document in the browser and *Bookmark* for storing the document in a bookmark-collection.

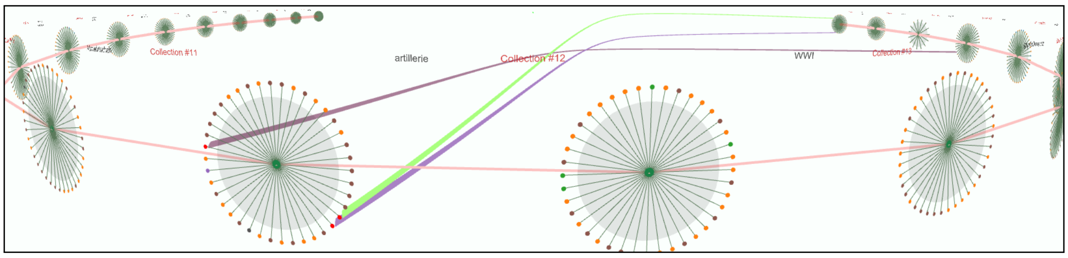


Figure 6.9: Hovering over the query-collection's center shows a connection to another collection on the right side in the overview mode.

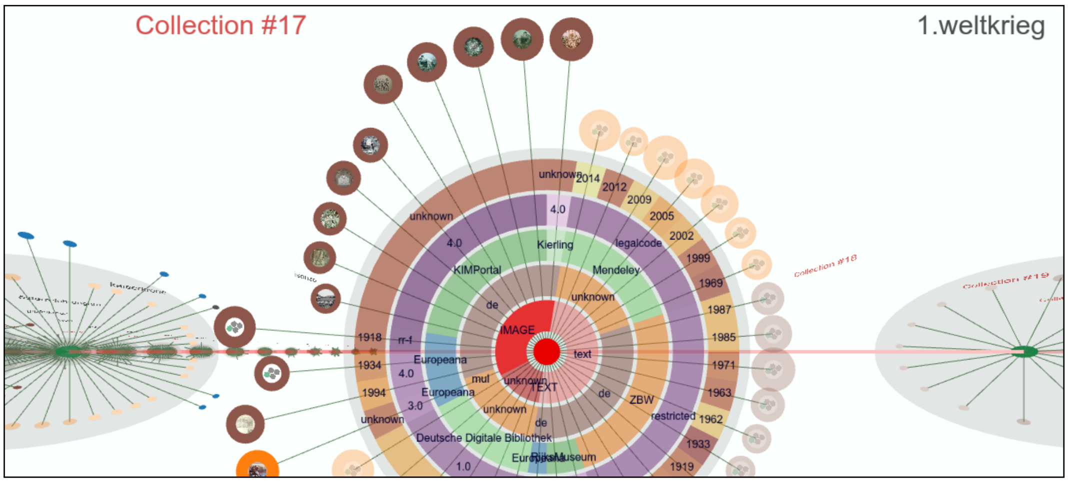


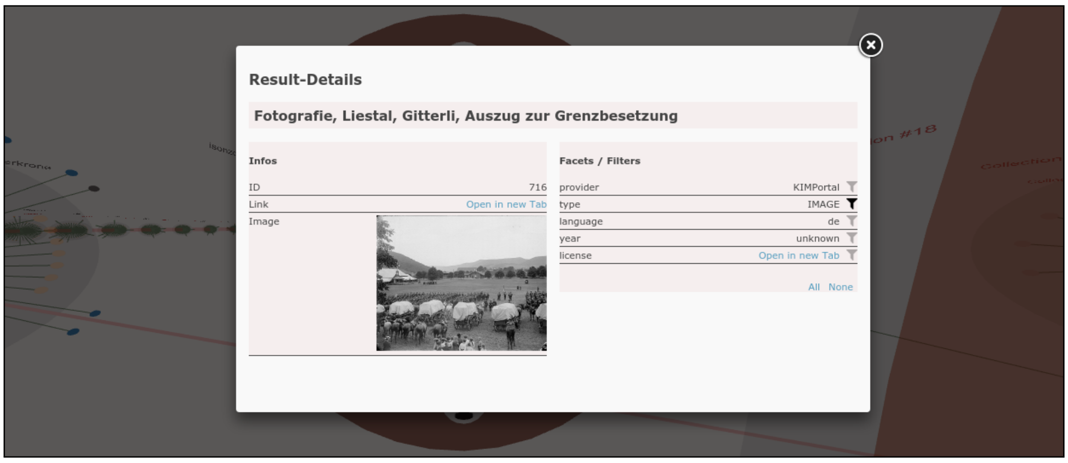
Figure 6.10: Filtering for documents with the type "`IMAGE`" on another collection

6.1.4 Discovering & Filtering a Third Query-Collection

After clicking on the query-collection, the camera focuses on it. Alice reads that the search query was "*1.weltkrieg*" (*WWI*). At first sight, she realizes that it contains some pictures. To find all documents that are classified as images, she clicks on the *IMAGE*-segment of the inner circle (*type*) of the *Ring-Representation*. Only documents that are declared as images, independently of whether they have preview-images or not, are shown now (Figure 6.10). Alice finally begins to investigate the interesting results. To check the title and further details of a result, she clicks on the node's ***i***-button (see Figure 6.8, bottom) which opens the detail-window (Figure 6.11). She decides to create a user-defined bookmark-collection on *WWI*, to save and organize relevant results. To do this, she clicks the node's *star*-button which allows her to bookmark different results and organize them into bookmark-collections (Figure 6.12).

6.1.5 Usage of the Collected Results

Finally Alice got a bookmark-collection containing a lot of relevant documents. For further investigations of the documents' metadata she decides



Result-Details

Fotografie, Liestal, Gitterli, Auszug zur Grenzbesetzung

Infos

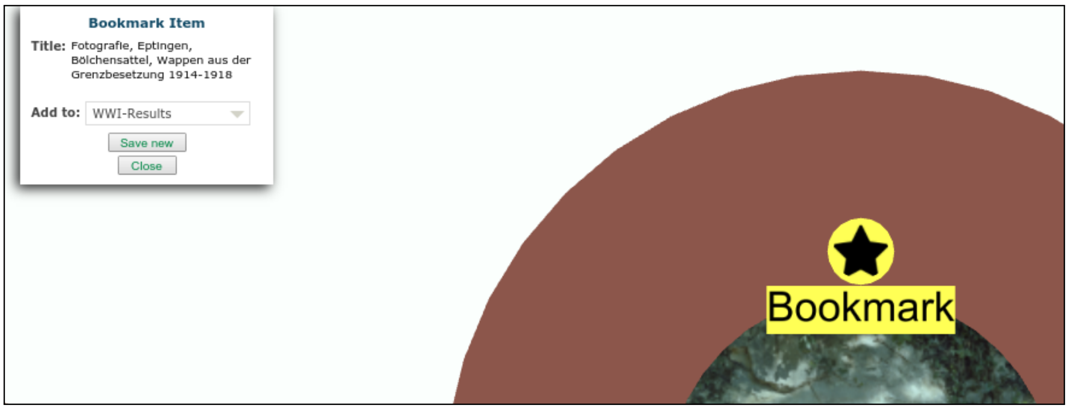
ID	716
Link	Open in new Tab
Image	

Facets / Filters

provider	KIMPortal
type	IMAGE
language	de
year	unknown
license	Open in new Tab

All None

Figure 6.11: Detail window showing a recommendation (info-panel)



Bookmark Item

Title: Fotografie, Eptingen, Bölchensattel, Wappen aus der Grenzbesetzung 1914-1918

Add to: WWI-Results

Figure 6.12: Bookmarking a result

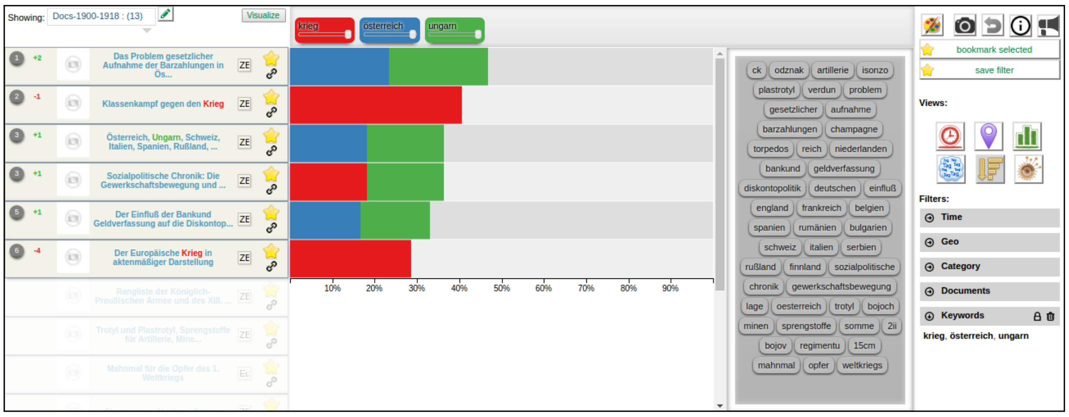

Showing: Docs-1900-1918 : (13)
Visualize
krieg
österreich
ungarn
1 +2 Das Problem gesetzlicher Aufnahme der Barzahlungen in Ös...
2 -1 Klassenkampf gegen den Krieg
3 +1 Österreich, Ungarn, Schweiz, Italien, Spanien, Rußland, ...
3 +1 Sozialpolitische Chronik: Die Gewerkschaftsbewegung und ...
5 +1 Der Einfluß der Bankund Geldverfassung auf die Diskontop...
6 -4 Der Europäische Krieg in aktenmäßiger Darstellung
Rangliste der Königlich-Preußischen Armee und des XIII. ...
Trotyl und Plastrotyl, Sprengstoffe für Artillerie, Mine...
Mahnmal für die Opfer des 1. Weltkriegs
10% 20% 30% 40% 50% 60% 70% 80% 90%
ck odznak artillerie isonzo plastrotyl verdun problem gesetzlicher aufnahme barzahlungen champagne torpedos reich niederlanden bankund geldverfassung diskontopolitik deutschen einfluß england frankreich belgien spanien rumänien bulgarien schweiz italien serbien rußland finnland sozialpolitische chronik gewerkschaftsbewegung lage oesterreich trotyl bojoch minen sprengstoffe somme 2ii bojov regimentu 15cm mahnmal opfer weltkriegs
bookmark selected
save filter
Views:
Filters:
Time
Geo
Category
Documents
Keywords
krieg, österreich, ungarn

Figure 6.13: Opening a bookmark-collection created inside *ECHO* and using *uRank* on it

to make use of the *RD*'s visualizations. She is interested about the documents' keywords in particular and thus opens the *uRank*-visualization (see Section 3.3.2). She opens the bookmark-collection by clicking on the *RD*'s bookmark-dropdown and selects the entry with the name she has chosen for the bookmark-collection before.

Now, after bundling the relevant documents from all her previous queries in a single bookmark-collection making use of *ECHO*, she can further refine and investigate them in *uRank* (see Figure 6.13).

7 Conclusion

This work concludes with the following chapter. Beginning with a summary, followed by a collection of lessons learned during this project, it finishes with suggestions for possible future work on *ECHO* and a final conclusion.

7.1 Summary

After an overview of related work in Chapter 2 on the topics of recommender systems, graph-comparing, and dynamic-graphs, ***EEXCESS*** and its ***RD*** were introduced in Chapter 3.

Chapter 4 introduced ***ECHO***, which is the tool generally described in this work. Before discussing its features in Section 4.2, it was necessary to list functional and non functional requirements in Section 4.1, which were derived from use cases before designing *ECHO*.

At first, considerations about the visual layout were made in Section 4.2.1. The general design of the collection and recommendation nodes was elaborated and a layout for the query history was designed. Section 4.2.2 discussed the navigation and the concepts of LODs used in *ECHO*, followed by the description of filtering in Section 4.2.3. As a key feature of *ECHO*, the ***RingRepresentation*** was introduced next in Section 4.2.4. This section included a short introduction to its creation-algorithm, its appearance, and its functionality and its benefits. The next sections discussed *ECHO*'s comparison mechanisms (Section 4.2.5) and the usage of bookmark-collections (Section 4.2.6). Finally fulfilling the non functional requirements were discussed, followed by a short summary of the chapter.

Chapter 5 went deeper into the **implementation** of *ECHO*. This was necessary to provide a good understanding of *ECHO*'s code and functionality.

At first, the general **implementation-environment**, such as the used programming language and the *RD*'s plugin-system was discussed. Further, Section 5.2 gave a more detailed explanation of ***WebGL*** and ***THREE.js*** and how they are used within *ECHO*.

The next Sections (5.3, 5.4) introduced **how *ECHO* is initialized** as a **plugin** and as the described scene. Its **internal components** which work in the background and how the code is structured were described in detail. The next aections showed two essential element-classes of the graph: Section 5.5 and Section 5.6 described the structure, data, and functionality of the `ECHO.Collection` and `ECHO.Recommendation`-classes.

Technical details about the internal **rendering-process** were described in Section 5.8, followed by an introduction to the usage of *ECHO*'s **interaction** mechanism (5.9) and the `ECHO.Text`-class (5.7). The chapter concluded with a short section about the **storage of variables** inside a configuration-object.

The work was concluded by providing a **case study** (Chapter 6). It demonstrated *ECHO*'s benefits by describing a task a user might want to complete, supported by screenshots for a better understanding. It summarized all of the major features described in Chapter 4.

7.2 Lessons Learned

During this work, some challenges came up, which were not expected at the beginning. The lessons learned through mastering those problems, regarding the **visual design** and **implementation**, are summarized in this section.

7.2.1 Visual Design and Graph-Layout

This section discusses challenges in designing the graph's layout and the general visualization of its elements, as well as what was learned through facing those problems.

7.2.1.1 Camera & Graph layout

The graph-layout was originally designed with the idea of visualizing the collections in a *Cover-Flow* representation, known from many music players (see Section 4.2.1.4). The basic idea supported the requirement of visualizing most of the scene at the same time continuously, even if a single collection was selected.

7.2.1.2 Problems Discovered

While implementing the visualization, the following problems regarding the camera and its position occurred:

- Positioning the collections on a **hyperbolic curve**, that lies in the x/z-plane, with the apex pointing towards the camera would help show all data easily in the field of view, but would lead to an **overlapping** of the collections.
- The solution was to set the collections on a **circular arch**. This helped provide enough space between the collections, but a trade-off between the **camera's distance** and the **camera-angle** had to be made: a higher distance of the scene to the camera allowed the use of a narrower angle, which does not distort the scene, but also leads to fewer details that can be seen. Otherwise, if the angle of view is high, the collections can be set nearer to the camera. Thus, more details in the middle of the camera's field of view can be recognized, but collections on the left and right side of the canvas are distorted.
  Nevertheless, the latter concept was used for visualization because of a preference of showing details over preventing a fisheye-effect. This concept, with a higher fish-eye-effect, also led to the side-effect that comparing splines (see Section 4.2.5) needed to be made thicker to be visible in the far distance.

7.2.1.3 Lessons Learned

- These problems lead to the question of whether positioning the collections on a **(flat) ring** was the optimal solution.
- The benefits of using a **three-dimensional** representation were **not optimally used** by leaving out a whole dimension (Y).
- The camera's navigation was implemented to allow a movement around a sphere surrounding the collections, but the collections were positioned only in the x/z-plane. Properly **using the whole sphere** could have helped prevent the problems mentioned above and could have helped visualize more data at the same time.

7.2.2 Visual Design

While implementing the layout of the collections, several limitations came up, which are outlined below.

7.2.2.1 Collection Layout

- The layout was planned in a state of the *EEXCESS* project, where the **received results were limited to 20 recommendations**. Thus, the place around the two-dimensional plane seemed to be enough.
- After implementing the collection-visualization that **limit was dropped**. Thus, it was hard to procure **additional space** for more results.
- It resulted in an internal limit of 40 recommendations around the collection node.

The following lessons were learned from that problems:

- An **a priori assumption of limits of data is dangerous** during an ongoing project consisting of different working-groups and sub projects.
- Similar to the ideas regarding the graph-layout, a different approach would be to not only use a vertical plane, but to **allow a position of the nodes all around the collection** in a three-dimensional space.

7.2.2.2 Ring-Representation

- During planning of the *Ring-Representation* (Section 4.2.4), **not enough considerations** about **placing the segments' labels** were made.
- This resulted in long, overlapping labels.
- To solve this problem, particularly long strings, like **URLs, were cut**.
- This lead to the **loss of meaning** of some labels.

These problems resulted in the following ideas that may be considered in future projects and in potential future work on this project:

- **A dynamic way of displaying labels** as text may be necessary. If there is enough space inside a segment, the size of the label may be different than a long text in a smaller segment.
- Segments on the left and right of the collection, in particular those which are vertically aligned, do not provide enough space for horizontal labels.
- The **usage of icons**, especially for a known subset of values, may help save space, and also help the user to recognize values more easily.
- An interactive possibility (hovering) may be found to provide more detailed information on a value if the user is explicitly interested.

7.2.3 Implementation

The implementation led to problems, especially regarding *ECHO*'s performance.

7.2.3.1 Performance

As discussed in the Sections 1.1 and 4.1 possible **performance problems** due to a lot of data should be prevented by using ***WebGL***, which makes use of the client's graphic card's hardware-acceleration.

During implementation, it became clear that the graphical rendering was **not the only bottle neck** to expect. Even the usage of *WebGL* lead to a **high usage of the CPU** and an especially high amount of memory-usage.

The problem may be identified in ***ECHO*'s structure** of how elements are held and calculated (see Section 5.4). Decisions were made to separate the logical elements that internally represent the scene's objects from the *THREE.js*-elements. This helped not to mix up the methods and values of a library's object with own code. However, it may have lead to **redundant data**. Optimized methods to calculate values may have been missed through that strict separation, which may result in performance issues.

Finally, *ECHO*'s usage is limited to machines containing a proper graphic card and at least an average CPU. Scenes containing a lot of collections may lead to high memory and CPU usage, which may cause a juddering scene.

For future work, considerations may have to be made to make **more use of *WebGL*'s features** and structures to gain **better performance through its internal optimizations**.

7.3 Future work

Concluding this work, some challenges and ideas remain open to be resolved and implemented in the future. This section gives an outlook of possible future work on *ECHO*.

7.3.1 Performance Tweaks

As mention in Section 7.2.3, one of the remaining problems is the performance of *ECHO*. Although *WebGL* is being used, the tool consumes a more than average amount of CPU and memory.

In the best case scenario, it should be possible to also use *ECHO* on computers with moderate power. This could be possible if a detailed investigation is carved out to optimize algorithms and find calculations that may can be performed by the GPU instead of the CPU.

7.3.2 Improving the Ring-Representation

The *Ring-Representation* (Section 4.2.4) is one of the most essential parts of this work. Thus, it was necessary to provide an implementation that works well and is easy to use.

Although there was a high focus on working out the *Ring-Representation*, some problems need to be solved in the future:

- As discussed above (Section 7.2.2), the values of the Ring-Segments are not visualized in an optimal way. A lot of the values' labels use more space than the segment provides. This leads to overlapping.
- It is necessary to find a way to let the user recognize the value on the one hand, but also to prevent overlapping values.
- Some of the facets that appear as Ring-Segments do not contain proper values. Often they are empty and currently shown as "`unknown`". Others, especially the *year*-facet, do not provide consistent values every time (see Section 4.2.3.1). Future developments may concentrate on better preparation of those values before loading them into *ECHO*.

7.3.3 Further Comparison Methods

The current comparison mechanism compares several collections and connects similar results. Since this method is a basic way of showing similarities between collections, further methods may be found in the future to extract and visualize relationships between different collections.

One possibility could be to use the received keywords of the recommendations. *ECHO* is designed to also allow other visualizations of relations, and is not only limited to connecting results through splines.

7.3.4 Filtering

ECHO's applied filters are currently summarized in the *RD*'s micro-visualization in the right column. However, they are not connected to the filtering mechanism of the *RD*, due to the following problems:

- Applying filters from *ECHO* on the *RD*'s data may lead to problems because of the different underlying data: the *RD* only handles one collection, while *ECHO* holds multiple collections. Thus, if filters are applied, it is possible, that those filtered values will not occur in the *RD*. Currently, it cannot handle this case and would throw an error.

7.4 Final Conclusion

ECHO realized a way to **visualize and analyze recommender result histories**. It was possible to fulfill the requirements set out in Section (4.1) and to provide a graph that allows the user to navigate, manage, analyze and compare collections and results.

The navigation (Section 4.2.2) allows easy movement between collections, as well as access to them and their results. Comparison (Section 4.2.5) and *Ring-Representation* (Section 4.2.4) allow an easy and productive way of finding recommendations within multiple collections. The usage of the *RD*'s bookmarking system gives possibilities to organize recommendations. Furthermore, the user has the option of visualizing bookmark-collections through *ECHO*. Several interaction mechanisms finally allow the user to open and use recommendations in external tools (see Section 4.2.1.3). An example of the practical use of *ECHO* was finally proved in Chapter 6.

Appendix

Bibliography

Ahn et al. (2011). "Temporal visualization of social network dynamics: Prototypes for nation of neighbors." In: *Social computing, behavioral-cultural modeling and prediction*. Springer, pp. 309–316 (cit. on p. 27).

Anderson (2013). *The long tail*. Nieuw Amsterdam (cit. on p. 29).

Andrews, Wohlfahrt, and Wurzinger (2009). "Visual Graph Comparison." In: *Proceedings of the 2009 13th International Conference Information Visualisation*. IV '09. Washington, DC, USA: IEEE Computer Society, pp. 62–67. ISBN: 978-0-7695-3733-7. DOI: 10.1109/IV.2009.108. URL: http://dx.doi.org/10.1109/IV.2009.108 (cit. on pp. 12, 15).

Archambault (2009). "Structural Differences Between Two Graphs Through Hierarchies." In: *Proceedings of Graphics Interface 2009*. GI '09. Kelowna, British Columbia, Canada: Canadian Information Processing Society, pp. 87–94. ISBN: 978-1-56881-470-4. URL: http://dl.acm.org/citation.cfm?id=1555880.1555905 (cit. on pp. 14, 16, 17).

Bach, Pietriga, and Fekete (2014). "GraphDiaries: animated transitions andtemporal navigation for dynamic networks." In: *Visualization and Computer Graphics, IEEE Transactions on* 20.5, pp. 740–754 (cit. on p. 26).

Becker and Cleveland (1987). "Brushing scatterplots." In: *Technometrics* 29.2, pp. 127–142 (cit. on p. 32).

Brandes and Corman (2003). "Visual Unrolling of Network Evolution and the Analysis of Dynamic Discourse." In: *Information Visualization* 2.1, pp. 40–50. ISSN: 1473-8716. DOI: 10.1057/palgrave.ivs.9500037. URL: http://dx.doi.org/10.1057/palgrave.ivs.9500037 (cit. on pp. 22, 24).

Brandes, Eiglsperger, et al. (2010). *Graph markup language (GraphML)*. Citeseer (cit. on p. 14).

Brill (1992). “A Simple Rule-based Part of Speech Tagger.” In: *Proceedings of the Workshop on Speech and Natural Language*. HLT ’91. Harriman, New York: Association for Computational Linguistics, pp. 112–116. ISBN: 1-55860-272-0. DOI: 10.3115/1075527.1075553. URL: http://dx.doi.org/10.3115/1075527.1075553 (cit. on p. 41).

Chau (2011). “Visualizing Web Search Results Using Glyphs: Design and Evaluation of a Flower Metaphor.” In: *ACM Trans. Manage. Inf. Syst.* 2.1, 2:1–2:27. ISSN: 2158-656X. DOI: 10.1145/1929916.1929918. URL: http://doi.acm.org/10.1145/1929916.1929918 (cit. on p. 1).

Chaudhri, Imran (2010). *Animated graphical user interface for a display screen or portion thereof*. US Patent D624,932 (cit. on p. 69).

Collberg et al. (2003). “A System for Graph-based Visualization of the Evolution of Software.” In: *Proceedings of the 2003 ACM Symposium on Software Visualization*. SoftVis ’03. San Diego, California: ACM, 77–ff. ISBN: 1-58113-642-0. DOI: 10.1145/774833.774844. URL: http://doi.acm.org/10.1145/774833.774844 (cit. on pp. 25, 26).

Congote et al. (2011). “Interactive Visualization of Volumetric Data with WebGL in Real-time.” In: *Proceedings of the 16th International Conference on 3D Web Technology*. Web3D ’11. Paris, France: ACM, pp. 137–146. ISBN: 978-1-4503-0774-1. DOI: 10.1145/2010425.2010449. URL: http://doi.acm.org/10.1145/2010425.2010449 (cit. on p. 102).

Danchilla (2012). “Beginning WebGL for HTML5.” In: Berkeley, CA: Apress. Chap. Three.js Framework, pp. 173–203. ISBN: 978-1-4302-3997-0. DOI: 10.1007/978-1-4302-3997-0_7. URL: http://dx.doi.org/10.1007/978-1-4302-3997-0_7 (cit. on p. 102).

Deshpande and Karypis (2004). “Item-based top-N Recommendation Algorithms.” In: *ACM Trans. Inf. Syst.* 22.1, pp. 143–177. ISSN: 1046-8188. DOI: 10.1145/963770.963776. URL: http://doi.acm.org/10.1145/963770.963776 (cit. on p. 7).

Erten et al. (2003). “GraphAEL: Graph Animations with Evolving Layouts.” In: *Graph Drawing*. Ed. by Giuseppe Liotta. Vol. 2912. Lecture Notes in Computer Science. Springer, pp. 98–110. ISBN: 3-540-20831-3. URL: http://dblp.uni-trier.de/db/conf/gd/gd2003.html#ErtenHKWY03 (cit. on pp. 21, 23).

Fruchterman and Reingold (1991). “Graph drawing by force-directed placement.” In: *Software: Practice and experience* 21.11, pp. 1129–1164 (cit. on p. 39).

Fu et al. (2007). "Visualization and analysis of email networks." In: *Visualization, 2007. APVIS'07. 2007 6th International Asia-Pacific Symposium on*. IEEE, pp. 1–8 (cit. on pp. 26, 27).

Gajer and Kobourov (2001). "GRIP: Graph dRawing with Intelligent Placement." In: *Proceedings of the 8th International Symposium on Graph Drawing*. GD '00. London, UK, UK: Springer-Verlag, pp. 222–228. ISBN: 3-540-41554-8. URL: http://dl.acm.org/citation.cfm?id=647552.729406 (cit. on p. 21).

Granitzer et al. (2013). "Unfolding Cultural, Educational and Scientific Long-Tail Content in the Web." In: *Late-Breaking Results, Project Papers and Workshop Proceedings of the 21st Conference on User Modeling, Adaptation, and Personalization., Rome, Italy, June 10-14, 2013*. URL: http://ceur-ws.org/Vol-997/umap2013_project_1.pdf (cit. on pp. 29, 30).

Harris, Robert L (2000). *Information graphics: A comprehensive illustrated reference*. Oxford University Press (cit. on p. 68).

Hascoët and Dragicevic (2011). *Visual Comparison of Document Collections Using Multi-Layered Graphs*. Tech. rep. RR-11020, p. 10. URL: http://hal-lirmm.ccsd.cnrs.fr/lirmm-00601851 (cit. on pp. 19, 20).

Herlocker et al. (2004). "Evaluating Collaborative Filtering Recommender Systems." In: *ACM Trans. Inf. Syst.* 22.1, pp. 5–53. ISSN: 1046-8188. DOI: 10.1145/963770.963772. URL: http://doi.acm.org/10.1145/963770.963772 (cit. on p. 6).

Hofmann (2004). "Latent Semantic Models for Collaborative Filtering." In: *ACM Trans. Inf. Syst.* 22.1, pp. 89–115. ISSN: 1046-8188. DOI: 10.1145/963770.963774. URL: http://doi.acm.org/10.1145/963770.963774 (cit. on p. 6).

Huang, Chen, and Zeng (2004). "Applying Associative Retrieval Techniques to Alleviate the Sparsity Problem in Collaborative Filtering." In: *ACM Trans. Inf. Syst.* 22.1, pp. 116–142. ISSN: 1046-8188. DOI: 10.1145/963770.963775. URL: http://doi.acm.org/10.1145/963770.963775 (cit. on p. 6).

"Introduction to Recommender Systems: Algorithms and Evaluation" (2004). In: *ACM Trans. Inf. Syst.* 22.1. Ed. by Konstan, pp. 1–4. ISSN: 1046-8188. DOI: 10.1145/963770.963771. URL: http://doi.acm.org/10.1145/963770.963771 (cit. on p. 6).

Koop, Freire, and Silva (2013). "Visual summaries for graph collections." In: *IEEE Pacific Visualization Symposium, PacificVis 2013, February 27*

2013-March 1, 2013, Sydney, NSW, Australia, pp. 57–64. DOI: 10.1109/PacificVis.2013.6596128. URL: http://dx.doi.org/10.1109/PacificVis.2013.6596128 (cit. on pp. 16, 18).

Kuhn (1955). "The Hungarian method for the assignment problem." In: *Naval research logistics quarterly* 2.1-2, pp. 83–97 (cit. on p. 14).

Melnik, Garcia-Molina, and Rahm (2002). "Similarity flooding: A versatile graph matching algorithm and its application to schema matching." In: *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, pp. 117–128 (cit. on p. 18).

Middleton, Shadbolt, and De Roure (2004). "Ontological User Profiling in Recommender Systems." In: *ACM Trans. Inf. Syst.* 22.1, pp. 54–88. ISSN: 1046-8188. DOI: 10.1145/963770.963773. URL: http://doi.acm.org/10.1145/963770.963773 (cit. on p. 6).

Miles (2016). *jQuery Essentials*. Packt Publishing Ltd (cit. on p. 50).

Munzner et al. (2003). "TreeJuxtaposer: Scalable Tree Comparison Using Focus+Context with Guaranteed Visibility." In: *ACM Trans. Graph.* 22.3, pp. 453–462. ISSN: 0730-0301. DOI: 10.1145/882262.882291. URL: http://doi.acm.org/10.1145/882262.882291 (cit. on pp. 11–13).

Mutlu, Hoefler, et al. (2014). "Suggesting visualisations for published data." In: *Information Visualization Theory and Applications (IVAPP), 2014 International Conference on*, pp. 267–275 (cit. on p. 30).

Mutlu and Sabol (2015). "Visual Analysis of Scientific Content." In: *The Special Technical Community on Social Networking (STCSN) E-Letter on Science 2.0, May 2015.* (Cit. on p. 30).

Mutlu et al. (2015a). "User Modeling, Adaptation and Personalization: 23rd International Conference, UMAP 2015, Dublin, Ireland, June 29 – July 3, 2015. Proceedings." In: ed. by Francesco Ricci et al. Cham: Springer International Publishing. Chap. Towards a Recommender Engine for Personalized Visualizations, pp. 169–182. ISBN: 978-3-319-20267-9. DOI: 10.1007/978-3-319-20267-9_14. URL: http://dx.doi.org/10.1007/978-3-319-20267-9_14 (cit. on p. 42).

Mutlu et al. (2015b). "VizRec: A Two-Stage Recommender System for Personalized Visualizations." In: *Proceedings of the 20th International Conference on Intelligent User Interfaces Companion*. IUI Companion '15. Atlanta, Georgia, USA: ACM, pp. 49–52. ISBN: 978-1-4503-3308-5. DOI: 10.1145/2732158.2732190. URL: http://doi.acm.org/10.1145/2732158.2732190 (cit. on p. 44).

Nowell, Hetzler, and Tanasse (2001). “Change blindness in information visualization: A case study.” In: *infovis*. IEEE, p. 15 (cit. on p. 22).

O’Donovan et al. (2008). “PeerChooser: Visual Interactive Recommendation.” In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’08. Florence, Italy: ACM, pp. 1085–1088. ISBN: 978-1-60558-011-1. DOI: `10.1145/1357054.1357222`. URL: `http://doi.acm.org/10.1145/1357054.1357222` (cit. on pp. 8, 10).

Parra et al. (2012). “Conference Navigator 3: An Online Social Conference Support System.” In: (cit. on p. 7).

Porter (2006). “An algorithm for suffix stripping.” In: *Program* 40.3, pp. 211–218. DOI: `10.1108/00330330610681286`. eprint: `http://www.emeraldinsight.com/doi/pdf/10.1108/00330330610681286`. URL: `http://www.emeraldinsight.com/doi/abs/10.1108/00330330610681286` (cit. on p. 41).

Rauch et al. (2015). “Knowminer Search - a Multi-Visualisation Collaborative Approach to Search Result Analysis.” In: *Information Visualisation (iV), 2015 19th International Conference on*. IEEE, pp. 379–385 (cit. on pp. 27, 28).

Ricci et al. (2010). *Recommender Systems Handbook*. 1st. New York, NY, USA: Springer-Verlag New York, Inc. ISBN: 0387858199, 9780387858197 (cit. on pp. 5, 7).

Salton and McGill (1986). “Introduction to modern information retrieval.” In: (cit. on p. 41).

Sciascio, di, Sabol, and Veas (2015). “*uRank*: Exploring Document Recommendations through an Interactive User-Driven Approach.” In: *Proceedings of the Joint Workshop on Interfaces and Human Decision Making for Recommender Systems, IntRS 2015, co-located with ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 19, 2015.* Pp. 29–36. URL: `http://ceur-ws.org/Vol-1438/paper5.pdf` (cit. on pp. 40, 41).

Sciascio, di, Sabol, and Veas (2016). “Rank As You Go: User-Driven Exploration of Search Results.” In: *Proceedings of the 21st International Conference on Intelligent User Interfaces*. IUI ’16. Sonoma, California, USA: ACM, pp. 118–129. ISBN: 978-1-4503-4137-0. DOI: `10.1145/2856767.2856797`. URL: `http://doi.acm.org/10.1145/2856767.2856797` (cit. on p. 41).

Tschinkel, Hafner, et al. (2016). “Using Micro-Visualisations to Support Faceted Filtering of Recommender Results” (cit. on pp. 32, 34, 39, 42).

Tschinkel, di Sciascio, et al. (2015). "The Recommendation Dashboard: A System to Visualise and Organise Recommendations." In: *19th International Conference on Information Visualisation, IV 2015, Barcelona, Spain, July 22-24, 2015*, pp. 241–244. DOI: 10.1109/iV.2015.51. URL: http://dx.doi.org/10.1109/iV.2015.51 (cit. on pp. 31–33, 36, 37).

Tschinkel, Veas, et al. (2014). "Using Semantics for Interactive Visual Analysis of Linked Open Data." In: *Proceedings of the 2014 International Conference on Posters; Demonstrations Track - Volume 1272*. ISWC-PD'14. Riva del Garda, Italy: CEUR-WS.org, pp. 133–136. URL: http://dl.acm.org/citation.cfm?id=2878453.2878487 (cit. on p. 30).

Ulbrich et al. (2015). "Reading Through Graphics: Interactive Landscapes to Explore Dynamic Topic Spaces." In: *Human Interface and the Management of Information. Information and Knowledge Design*. Springer, pp. 127–137 (cit. on pp. 39, 40).

Veas et al. (2015). "Visual Recommendations for Scientific and Cultural Content." In: *Proceedings of the 6th International Conference on Information Visualization Theory and Applications (VISIGRAPP 2015)*, pp. 256–261. ISBN: 978-989-758-088-8. DOI: 10.5220/0005352802560261 (cit. on p. 31).

Verbert et al. (2013). "Visualizing Recommendations to Support Exploration, Transparency and Controllability." In: *Proceedings of the 2013 International Conference on Intelligent User Interfaces*. IUI '13. Santa Monica, California, USA: ACM, pp. 351–362. ISBN: 978-1-4503-1965-2. DOI: 10.1145/2449396.2449442. URL: http://doi.acm.org/10.1145/2449396.2449442 (cit. on pp. 7–9).

Ware (2012). *Information visualization: perception for design*. Elsevier (cit. on p. 1).

Watson (2012). *Learning Modernizr*. Packt Publishing Ltd (cit. on p. 105).

Woo et al. (1999). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201604582 (cit. on p. 102).

Wozelka, Kröll, and Sabol (2015). "Exploring Time Relations in Semantic Graphs." In: *SIGRAD 20155 (the Swedish Chapter of Eurographics)*. Stockholm, Sweden (cit. on p. 27).

Wright, Lipchak, and Haemel (2007). *Opengl®Superbible: Comprehensive Tutorial and Reference, Fourth Edition*. Fourth. Addison-Wesley Professional. ISBN: 9780321498823 (cit. on p. 103).

Zheng et al. (2013). "A survey of faceted search." In: *Journal of Web engineering* 12.1&2, pp. 041–064 (cit. on p. 27).