TU WIEN Informatics

Explorative Suchschnittstelle für Wissensdatenbanken

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Philip Bugnar
Matrikelnummer 01402617

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Ass. Dr.techn. Manuela Waldner, MSc

Wien, 29. Mai 2021

Philip Bugnar	Manuela Waldner

TU WIEN Informatics

Exploratory Search Interface for Knowledge Databases

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Philip Bugnar
Registration Number 01402617

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Ass. Dr.techn. Manuela Waldner, MSc

Vienna, 29th May, 2021

Philip Bugnar	Manuela Waldner

Erklärung zur Verfassung der Arbeit

Philip Bugnar

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. Mai 2021

Philip Bugnar

Abstract

Visualization has always been a powerful tool to effectively convey knowledge and information. It has also gained attention in the context of search, in particular for newer visualization techniques like “Multifaceted Search” and “Exploratory Search”. There are currently many tools and websites that still rely on an explicit search function or an alphabetically ordered glossary of terms to allow users to filter and browse resources. This results in many useful resources not being discovered by users because of a lack of proper search tools. Exploratory Search is more open-ended, allowing users to search even if they do not exactly know what they are looking for.

This thesis proposes an adaptable, modular, web-based prototype of an exploratory search interface. The goal of the prototype is to serve as a basis for the evaluation of exploratory search interfaces for a wide variety of use cases. In contrast to many existing Exploratory Search tools, this prototype does not require rich meta-data to be present in a dataset. By utilizing an optional preprocessing step to extract named entities via Natural Language Processing, the prototype is compatible with most text-based datasets. The search interface consists of a word cloud created by a force-directed layout algorithm that places related entities close to each other. This interface also serves as the main filtering option, which keeps the users’ focus on the word cloud. After selecting interesting entities, matching documents can be browsed in a list view.

Kurzfassung

Visualisierungen sind seit jeher ein leistungsfähiges Instrument zur effektiven Vermittlung von Wissen und Informationen. Sie haben auch im Zusammenhang mit Suchanfragen an Aufmerksamkeit gewonnen, insbesondere für neuere Visualisierungstechniken wie "Multifaceted Search" und "Exploratory Search". Derzeit gibt es viele Tools und Websites, die immer noch auf eine explizite Suchfunktion oder ein alphabetisch geordnetes Glossar von Begriffen angewiesen sind, um den Benutzern das Filtern und Durchsuchen von Ressourcen zu ermöglichen. Dies führt dazu, dass viele nützliche Ressourcen von Benutzern nicht entdeckt werden, weil es an geeigneten Suchwerkzeugen fehlt. Die explorative Suche ist offener und ermöglicht es den Benutzern auch dann zu suchen, wenn sie die benötigten Suchbegriffe nicht kennen.

In dieser Arbeit wird ein flexibler, modularer, webbasierter Prototyp für eine explorative Suchschnittstelle entwickelt. Das Ziel des Prototyps ist es, als Grundlage für die Evaluierung von explorativen Suchschnittstellen für eine Vielzahl von Anwendungsfällen zu dienen. Im Gegensatz zu vielen bestehenden Exploratory Search Tools setzt dieser Prototyp keine umfangreichen Metadaten in einem Datensatz voraus. Durch die Verwendung eines optionalen Vorverarbeitungsschritts zur Extraktion von Entitäten mittels natürlicher Sprachverarbeitung ist der Prototyp mit den meisten textbasierten Datensätzen kompatibel. Die Suchoberfläche besteht aus einer Wordcloud, die durch einen force-directed Layout-Algorithmus erstellt wird, der verwandte Begriffe nahe beieinander platziert. Diese Schnittstelle dient auch als wichtigste Filteroption, die den Fokus des Nutzers auf die Wordcloud lenkt. Nach der Auswahl interessanter Entitäten können übereinstimmende Dokumente in einer Listenansicht durchsucht werden.

Contents

CHAPTER 1

Introduction

Over the last years, the information that is publicly available on the internet has increased drastically. However, most of it still consists of unstructured textual data, which is difficult for humans to process effectively. The most widespread and commonly known tool for querying such data is the search bar. The classical search bar has been used to browse the web since 1990 and, while becoming increasingly more intelligent over the years, it has not changed much from a user interface perspective. This is partly because it did not have to change all that much when only considering the primary reason for its invention: Querying a dataset by a term the user knows. For this particular task, a search bar is probably the most concise, useful and simple interface possible, although studies have shown that the majority of users still have problems understanding how search bars work [Hea09].

The issues with search bars start to appear when users do not know what they are looking for - either because they lack knowledge of the proper search terms or because they simply want to browse and explore a dataset without a specific goal in mind. In both cases, the users' goals cannot be accomplished with traditional search bars. For this reason, the concepts of exploratory search and faceted search have gained increasing attention over the last years. Different approaches like VisGets [DCCW08] combine spatial, temporal and topical information to allow users to explore a dataset. More examples can be found in the work by White et al. [WMM08], who point out that, while there are multiple approaches that work well, evaluating exploratory search interfaces can be difficult because most of them are context-dependent. This hinders finding an appropriate sample size for a user study. In addition to this, most exploratory search interfaces require some kind of metadata, like geographical, topical or spatial information, which limits the number of datasets these tools can be evaluated against.

This thesis presents a fully functional prototype of a word cloud based exploratory search interface, which works on any text-based dataset and is ready for real-world use cases and

evaluation. To develop this prototype, the research challenges described in the following sections have been identified and tackled.

1.1 Entity Extraction

As mentioned above, the prerequisite of having some sort of structure or metadata available seems to be the biggest bottleneck for exploratory search interfaces in terms of them being usable, and therefore evaluable, against a real-world dataset. The prototype we propose also requires specific metadata about the dataset. However, we provide the means to extract this information in a preprocessing step using Natural Language Processing (NLP). This is the key feature that enables this prototype to be used with any dataset that consists of unstructured textual data.

1.2 Interactive Interface for Exploratory Search

The main part of the search interface consists of a semantic word cloud. This is strongly inspired and based on the work by Xu et al. [XTL16] and further described in Section 3. The basic idea is to construct a graph from the data extracted in the preprocessing step, where the vertices represent the extracted entities and the edges and their respective weights are calculated from the co-occurrence of two entities across all documents. The graph is then rendered as a word cloud using a force-directed layout further described in Section 3.3.3. The word cloud can then be iteratively filtered by selecting entities and different categories. At any time during the filtering process, a list of documents matching the selected entities can be viewed.

Most of the time, datasets consist of many entries, which need to be browsable. This means that the UI needs to be able to handle large amounts of data while still staying fluid and responsive in a web-based environment. The solution for creating an interface that meets those requirements proved to be a major obstacle to overcome and is described further in Section 3.2.

1.3 Adaptable Implementation

To make the prototype as useful as possible, it has to be **adaptable** (e.g. adding a timeline as shown by VisGets if temporal metadata is available) and **modular** (meaning that certain pieces like preprocessing or graph construction should be replaceable by the users' own implementations). More details can be found in Section 4.

CHAPTER 2

Related Work

While this thesis focuses on producing a usable and evaluable prototype, large parts of the underlying theory regarding exploratory search, semantic word clouds, force-directed layouts, entity extraction and NLP have already been compiled and presented. This section provides a quick overview of some particularly interesting pieces of research in those fields.

First and foremost, White et al. [WMM08] criticize that current research focuses too much on creating new exploratory search interfaces and too little on evaluating them. They also speculate that this is because exploratory search interfaces are hard to evaluate. After all, their usefulness is very context and user-dependent. This further reinforced the design goals that our prototype has to be applicable to a wide range of contexts to make it as easily evaluable as possible.

Since the prototype focuses on exploring text-based information, using word clouds for the visualization was a natural choice. Word clouds have proven their usefulness to summarize textual data many times, as shown, for example, by Heimerl et al. [HLLE14] and Seifert et al. [SKK+08]. They also have been utilized for exploratory search interfaces as demonstrated in Imagesieve by Lin et al. [LAB+10].

Studies as conducted by Hearst et al. [HPP+20] indicate that grouped word clouds with sufficient white space are a viable method to summarize textual data. They also explore different layout options. While they explicitly state that they only evaluated semantic groupings that are distinct, we theorize that some of their findings might also apply to overlapping categories, i.e. words that appear in multiple documents, if a sufficient amount of white space is left to separate the categories.

In their work, Dörk et al. [DCCW08] combine multiple information vizualization widgets (= VisGets) to allow for data exploration across multiple data dimensions. This approach inspired us to strive for a high amount of adaptability and modularity in our prototype to make it composable with other exploratory search interfaces in the future.

Finally, the research conducted and the Semantic Word Cloud approach developed by Xu et al. [XTL16] serves as an important implementation reference and guideline for this prototype. While our prototype is a lot less sophisticated from an algorithmic point of view, its adaptability and modularity compensate for that. Furthermore, there are no mentions of whether their approach is web-based and, therefore, as widely applicable as our prototype.

The presented technique relies heavily on Natural Language Processing (NLP), a well established means for text information retrieval for many years, which Liddy defines as follows:

"Natural Language Processing is a theoretically motivated range of computational techniques for analyzing and representing naturally occurring texts at one or more levels of linguistic analysis for the purpose of achieving human-like language processing for a range of tasks or applications."[Lid01, p. 1]

The main subcategory of NLP used for this prototype is called Named Entity Extraction. This technique scans a given text and extracts all important entities from a text, e.g. locations, named events, famous people or geographical regions.

To classify and rank documents based on the users' query, a slight variation of the Term Frequency * Inverse Document Frequency (tf*idf) score is utilized. The tf*idf score describes how important a term is to a document in relation to a collection of documents. The frequency of a term within a document increases the score, and thereby the importance, of the term for the document, whereas the frequency of the same term across multiple documents decreases the score. Since there are multiple variations of the tf*idf score, the one used in this thesis is further described in Algorithm 3.3.

CHAPTER 3

Concept & User Interface Design

This section outlines the design rationales and ideas behind the exploratory search interface. It aims to give an overview of the applied algorithms as well as the general structure of the prototype. Implementation details are covered in Chapter 4.

Figure 3.1 provides a general overview of the structure of the prototype. Every part of this figure will be thoroughly covered in Section 3.3.

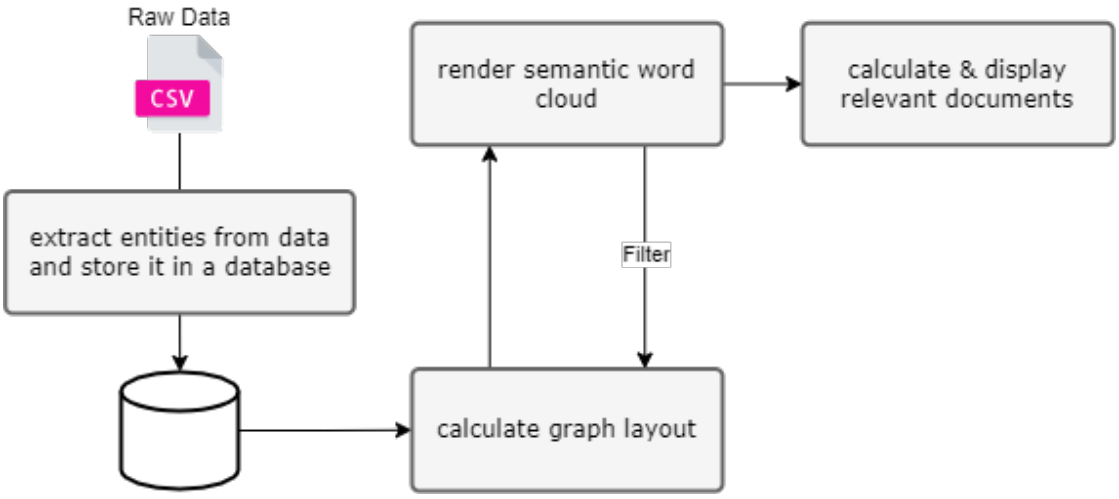

Figure 3.1: Overview of the general structure of the prototype.

3.1 Dataset

As can be seen in Figure 3.1, the .csv file containing the dataset is the main entry point into the prototype. Following the design goals of the prototype, the requirements

regarding the dataset should be as simple as possible. The minimum requirements for each row of the dataset are as follows:

- **ID** - A unique identifier. This identifier will be returned when relevant documents for tag queries are being calculated.
- **Content** - The text content that will be used to extract named entities in the preprocessing step. If a document consists of multiple parts (e.g. a title, an abstract and sections) that should be considered, they can be concatenated into a single column in the .csv file.

This design allows the database created by the prototype to be completely separated from any other persistency layers already present. This allows metadata of any complexity to be fetched purely based on the unique identifier given. At the same time, the preprocessing step can be adapted to include metadata on a per-document basis if one wishes to not use any other persistency layer.

The dataset used to develop, test, and evaluate the interface consists of over 100,000 news articles by major U.S. publishers [Kag18]. The entries in the dataset consist of title, publication, author, date, year, month, URL, and content. However, only the title and the actual content of each article were used. Furthermore, for most of the development process, only a small subset of 1,000 articles was used. All screenshots in this thesis were made using this small dataset unless stated otherwise.

For the remainder of the thesis, the word “tag” will be used to describe a word or a phrase that is characteristic for a document.

3.2 User Interface Design

The goal of the user interface is to provide an overview of the most important content available in the dataset as well as to provide means to further filter the data according to the user’s interest. The following pieces of information are encoded in the size, location, and color of every tag:

1. **Size** indicates the importance of a tag to the entire corpus. Tags that appear more often will be drawn bigger. This is a common and effective visual encoding that is often used in word-clouds like the one created by Cui et al. [CWL+10].
2. **Color** indicates the relation of a tag to a category. Categories help to distinguish the tags into groups and improve the filtering of tags as shown by Hearst et al. [HPP+20].
3. **Location** indicates the relatedness between tags. Tags that appear together more often will be closer together. This helps to direct and expand the search direction and was also used by Xu et al. [XTL16].

The initial state of the search interface can be seen in Figure 3.2. The entire word cloud can be panned and zoomed, and the filter can be hidden to create more space for the word cloud. To avoid overcrowding, the amount of initially displayed tags is limited (see Section 3.3.2).

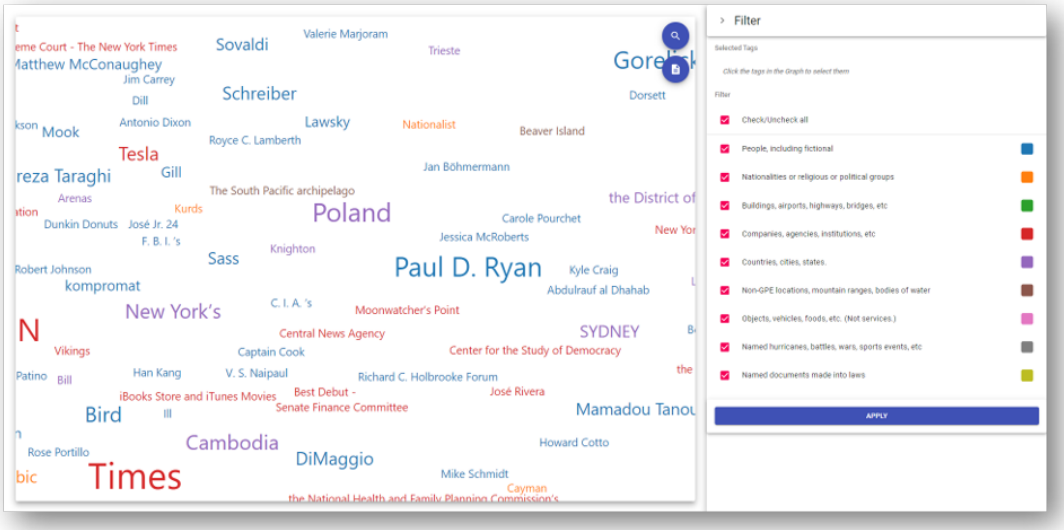

Figure 3.2: The full search interface.

Upon selecting one or more interesting tags and clicking the filter button, the word cloud rearranges itself. Only tags that are related to the selected ones will be displayed and arranged according to the criteria defined above. This allows tags that would usually not be displayed because they are not important to the corpus as a whole to show up. These new tags act as input for the user to further expand their search if desired. The user can then iterate through this process as often as they like to refine the search query.

After the user is done picking tags, they can display a list of documents that match their query, as can be seen in Figure 3.3.

A detailed walkthrough of a possible exploration sequence can be seen in Figure 3.4.

3.3 Concept

The prototype can be broken down into the following distinct parts:

- **Preprocessing**. This part is only required once per dataset and has to be computed offline. In this step, all required named entities and their occurrences across all documents are extracted using NLP.

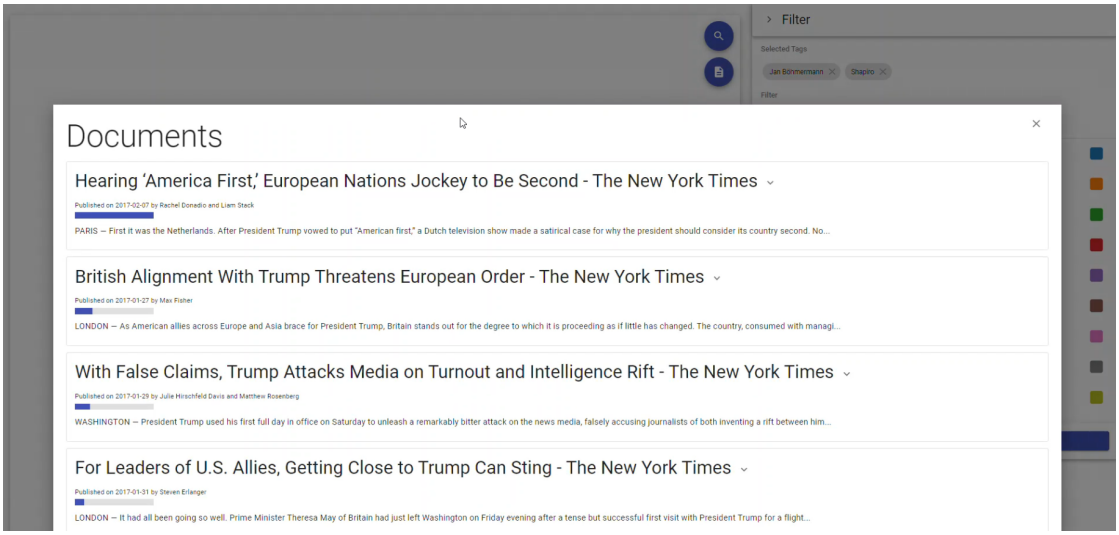

Figure 3.3: List of documents that match the query "Shapiro" and "Jan Böhmermann".

- **Graph data calculation & filtering**. This part calculates the subset of named entities that should be displayed depending on the filter set by the user.
- **Word cloud rendering**. Based on the selected tags, a word cloud is rendered. The layout is based on the importance of tags as well as the relationship between tags.
- **Document score calculation**. This part compiles a set of relevant documents based on the user's selected tags. For every document, a matching score between 0% and 100% is calculated. Afterwards, a list of document surrogates can be viewed. For each document, the calculated matching score is displayed. Furthermore, the selected tags are highlighted in the document text.

These parts will be discussed in more detail in the following sections.

3.3.1 Preprocessing

This first and optional step aims to solve a problem many huge datasets, including the one chosen for this thesis, have in common: The entries in the dataset are not tagged or categorized in any way. The preprocessing step prepares the given dataset for further use and only has to be executed once. The dataset has to be provided in the widespread comma-separated values format (.csv). It will go through an NLP pipeline, during which all named entities, as well as important metadata, are being extracted and stored in a relational database. The following data and metadata is extracted:

- Name of the entity. This string will be displayed as a tag in the word cloud.

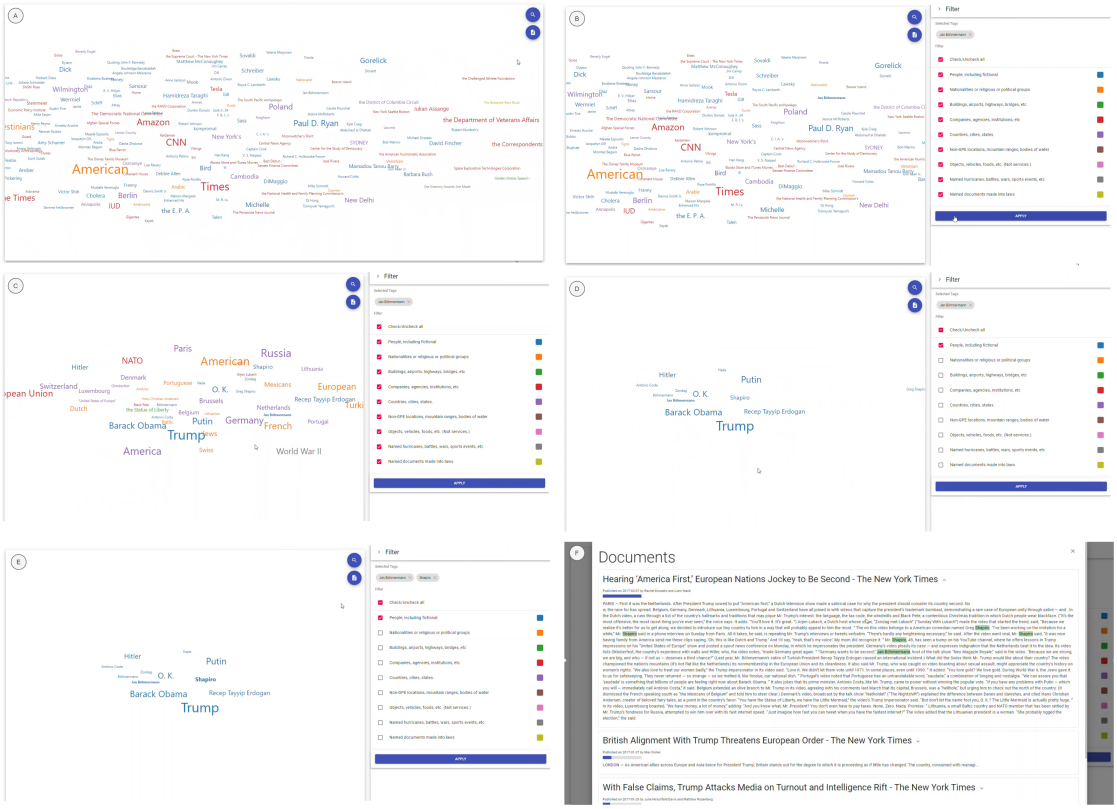

Figure 3.4: Walkthrough of a possible exploration sequence: (A) Initial exploration of the word cloud. (B) Finding and selecting an interesting tag (in this case "Jan Böhmermann"). (C) Applying the filter and exploring the results. (D) Filter results by category (in this case "People, including fictional"). (E) Selecting a second interesting entry (in this case "Shapiro"). (F) Clicking the document icon shows the results ordered by matching rate. Selected tags are highlighted in the text.

- Type of the entity. The following types of entities will be distinguished[1]:
  - People (including fictional)
  - Nationalities or religious or political groups
  - Buildings, airports, highways, bridges, etc.
  - Companies, agencies, institutions, etc.
  - Countries, cities, states
  - Non-GPE locations, mountain ranges, bodies of water
  - Objects, vehicles, foods, etc. (Not services)

[1]Categories are defined by the spacy library used for named entity recognition, further described in Chapter 4.2.

  - Named hurricanes, battles, wars, sports events, etc.
  - Named documents made into laws

- Total amount of times the entity occurs across the entire dataset. This value could be computed at runtime but is stored for performance reasons.
- Amount of times the entity occurs in every document.

All of this information is required for the graph data and document score calculation. The preprocessing step is further illustrated in Algorithm 3.1.

Algorithm 3.1: Preprocessing

Input: A *dataset* where every entry is a document. A function *nlp* which extracts all named entities, including entity type.
Output: A relational database containing all information required for graph data and document score calculation.

```
for document in dataset do
    entities = nlp(document)
    for entity in entities do
        if entity does not exist in database then
            add database entry for entity with occurrence count = 1
        end
        increment total occurrence counter for entity in database
        add relation between document and entity (including amount of
         occurrences)
    end
end
```

The result of the preprocessing step is a multivariate graph where the vertices represent named entities along with their metadata and the edges represent the co-occurrence of two entities in a document with the weight indicating how often the two entities co-occur.

3.3.2 Graph Data Calculation & Filtering

Since this section strongly relies on graph theory, named entities will be referred to as nodes, and links between nodes will be referred to as edges. Two nodes are linked if they co-occur in at least one document.

This step is responsible for calculating the graph data that will later be displayed as a word cloud. If no filter is set, the n heaviest vertices (i.e. the n tags that occur the most across all documents) and their corresponding edges will be fetched, where n is an arbitrary number that limits the amount of rendered vertices to reduce visual clutter. The default value for n is 200 because it yields good rendering times while still providing enough tags to explore without causing too much clutter.

If a filter is set, only the selected vertices plus their directly related edges are fetched. A vertex is considered related to a selected vertex if the two are adjacent, meaning they co-occur in at least one document. This filtering sometimes causes vertices to end up with a degree of 0, meaning they are not co-occurring with any other selected vertex. This causes the vertex to float away in the final layout. To counteract this and to help the user discover more connections between the selected terms, vertices without neighbors are treated differently. For every vertex without neighbors, all "indirect neighbors", meaning vertices with a geodesic distance of 2, are also considered as related. However, in addition to averaging the weight of the two edges between the two vertices, the average weight is also halved to indicate that the connection is only indirect. An example of this is showcased in Figure 3.5. The exact process of calculating these substituted edges is described in Algorithm 3.2.

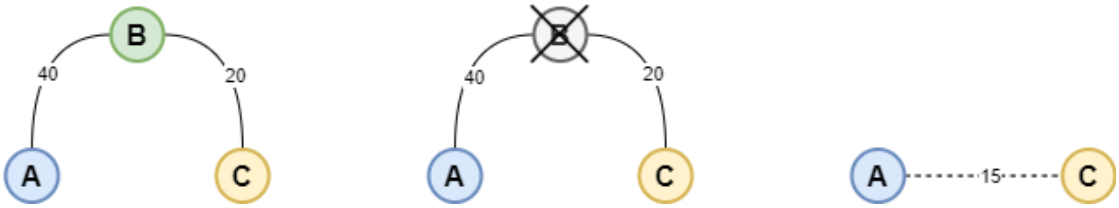

Figure 3.5: Example for indirect link calculation: Node B is removed through filtering, leaving nodes A and C without neighbors. Therefore an indirect link with half the average weight of the two nodes is created.

Algorithm 3.2: Edge Substitution

Input: Graph $G = (V, E)$. Selected vertices $S \subset V$. Selected Categories C.
Output: A subgraph that is ready for rendering.

$relatedVertices = v \in V \mid \exists s \in S \wedge (s, v) \in E$
$verticesToRender = r \in relatedVertices \mid category(r) \in C$
$verticesWithoutEdges = v \in verticesToRender \mid deg(v) = 0$
$indirectLinks = []$
for *v in verticesWithoutEdges* **do**
    $indirectVertices = r \in relatedVertices \mid d(r, v) = 2$
    for *i in indirectVertices* **do**
        $indirectLink = E(v, i)$
        // divide by two to get average and then by two again to cut weight in half
        $weight(indirectLink) = \frac{w(v,i)}{4}$
        $indirectLinks+ = indirectLink$
    end
end
$edgesToRender = indirectLinks \cup E | s \in S, v \in verticesToRender \wedge d(s, v) = 1$
return *verticesToRender, edgesToRender*

The result of this calculation process is a subgraph that contains only entities that are directly or indirectly related to the ones selected by the user. This graph can then be rendered, as described in the next step.

3.3.3 Word cloud rendering

This step is responsible for rendering the graph structure described in Section 3.3.1. The graph is a multivariate undirected graph where every vertex has name, weight, and category attributes, and every edge has a weight attribute.

The main goal of the word cloud visualization is twofold. Firstly, related entities, meaning entities that co-occur in one or more documents, should be placed closer to each other. This should enable users to find relations they did not know about by scanning the proximity of an entity they are interested in. Secondly, important entities, meaning entities that occur more often in the dataset than others, should be increased in size to demonstrate them being more important to the dataset as a whole. Finally, overlaps between tags should be avoided while keeping the layout compact, which are two common challenges when creating word clouds.

To achieve this, a force-directed layout with multiple forces was applied to the graph. The following forces were applied:

- **Charge/Repulsion**. A weak repulsion is applied statically to all nodes to naturally separate them from each other. This is done to prevent nodes from being close by without actually being related. While this causes the graph to be less dense, which is often considered detrimental, it helps to visually distinguish clusters of entities.
- **Link**. A linking force is applied to all entities that are linked, meaning they co-occur in one or more document. The higher the edge weight, the stronger the linking force is, resulting in the two entities being closer together.
- **Collision**. A collision force is applied to all nodes in order to prevent overlaps caused by any other force. The bounding box of the text is used as a base for collision detection.

A basic example of how a finished layout could look like can be seen in Figure 3.6 for Tables 3.1 and 3.2.

The resulting layout can then be further filtered as described in Section 3.2. After selecting one or more entities from the word cloud, a list of matching documents can be viewed.

3.3.4 Document Score Calculation

While the previous sections were focusing mainly on processing and presenting data to enable the user to browse a dataset, this section deals with deriving actual results based

Table 3.1: Nodes and their respective weights used for the force directed layout example in Figure 3.6.

Vertex	Weight
A	20
B	15
C	15
D	10
E	5
F	10

Table 3.2: Links present between nodes and their respective weights used for the force directed layout example in Figure 3.6. Nodes with a high weight are being drawn bigger, and links with heavier weights pull their connected nodes closer together.

Source	Target	Weight
A	B	5
A	C	2
B	C	1
D	E	2
E	F	1

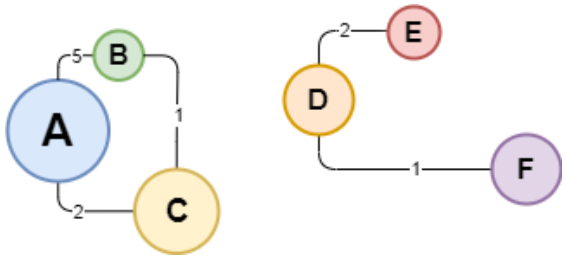

Figure 3.6: Example of a force directed layout created based on the values in Tables 3.1 and 3.2.

on the user's search query. The result consists of a set of documents that match the user's search query, where every document also gets assigned a percentual matching score depending on how well it fits the search query. In the prototype, this score is displayed as a blue bar below the document title.

This is described in more detail in Algorithms 3.3 and 3.4.

Algorithm 3.3: Term Frequency * Inverse Document Frequency Score Calculation

Input: Term t and document d for which the tf*idf score should be calculated
Output: tf*idf score for a given term t and document d

$countTInD$ = count of term t in document d
$countAllTInD$ = count of all terms in document d
// Term Frequency
$tf = \frac{countTInD}{countAllTInD}$
// Inverse Document Frequency
$countDocs$ = total document count (= size of corpus)
$docsWithT$ = count of documents where t occurs at least once
idf = $\log(\frac{countDocs}{docsWithT})$
return *tf * idf*

Algorithm 3.4: Document Score Calculation

Input: Selected entities S. Corpus of documents D.
Output: A set of matching documents $d \subset D$ with scores between 0.0 and 1.0 determining how well they match S.

$matchingDocuments = d \in D \mid S \cap entities(d)$
for *m in matchingDocuments* **do**
  $score(m) = \sum_{s \in S} tfidf(s, m)$
end
$maxScore$ = highest score in $matchingDocuments$
for *m in matchingDocuments* **do**
  $score(m) = \frac{score(m)}{maxScore}$
end
return *matchingDocuments*

CHAPTER 4

Implementation

The main goal of this thesis is to provide a prototypical implementation of an exploratory search interface that works with as many text-based datasets as possible while still being easy to extend and adapt. Furthermore, it should be modular enough to let developers choose which parts of the implementation they want to use and which ones they want to replace with their own implementation.

To keep the implementation as widely applicable as possible, it is entirely web-based. It also follows the common client/server architecture where the back end is responsible for the more complex calculations. The front end is responsible for user interaction and graph rendering. The preprocessing step is separated from the client/server architecture since it only needs to be run once. The three distinct parts (back end, front end, and preprocessing) can be described as follows:

- **Preprocessing**. This step uses Python and a PostgreSQL database to parse the initial data given as a .csv file into a database for later use.
- **Back end**. The back end consists of a NodeJS server based on express.js and is written in Typescript. It has the following responsibilities:
  - Providing the initial graph structure (Fetching vertices and edges from the database).
  - Filtering the initial graph structure depending on the users' query.
  - Calculating matching documents for a given search query.
- **Front end**. The front end consists of a React app with multiple reusable components which are written in Typescript. The d3.js library is used for graph rendering. The front end has the following responsibilities:

- Rendering the initial graph as well as subsequent subgraphs depending on the set filter.
- Providing a UI to filter the graph by selected entities as well as categories.
- Providing a UI to view the resulting documents based on selected entities.

This structure can also be seen in Figure 4.1. The following sections will elaborate on each layer and responsibility, pointing out configuration/customization options as well as implementation details if necessary.

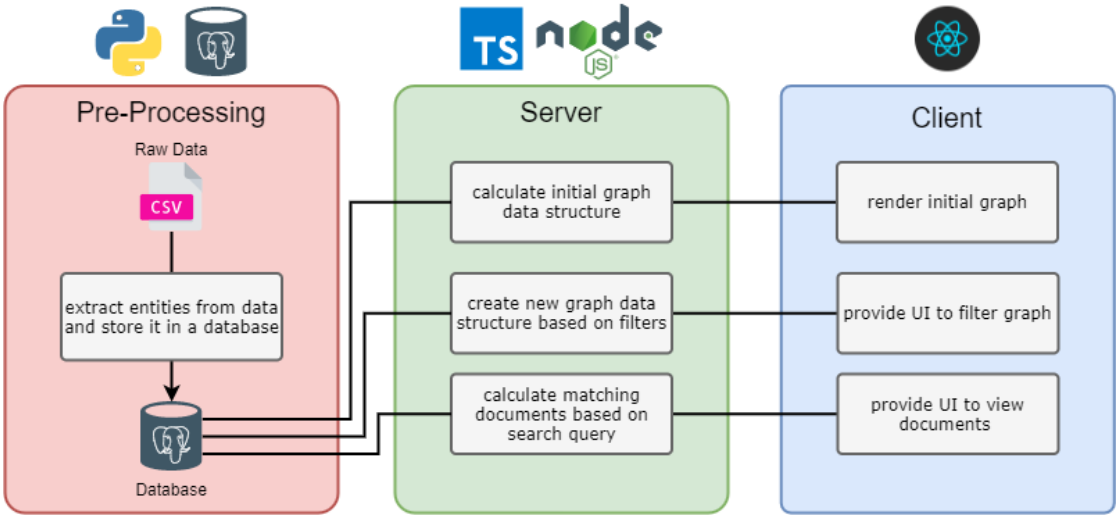

Figure 4.1: Overview of the software architecture of the prototype, including responsibilities.

Moreover, Figure 4.2 showcases the most important modules. These will be further described in the following sections.

4.1 Technology Stack

- Preprocessing
  - **Python** as a scripting language.
    - **Pandas** to read and parse .csv files.
    - **Spacy** to perform NLP on the documents.
  - **PostgreSQL** as a database solution.
- Back end
  - **Typescript** as a Javascript extension for type-safety and ease of use.

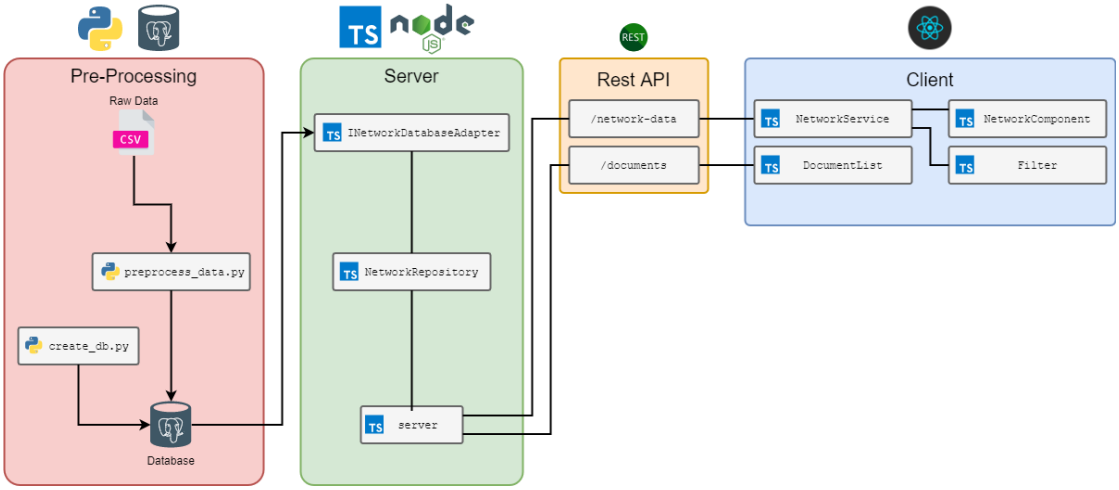

Figure 4.2: Overview of the most important modules and their interfaces.

  - **Node.js** as a Javascript runtime.
    - **Express** as a basic webserver.

- Front end
  - **Typescript** as a Javascript extension for type-safety and ease of use.
  - **React** as a web-framework for reusable components.
  - **d3.js** for force-directed layouts and SVG manipulation.

4.2 Preprocessing

The preprocessing step is what enables this prototype to work without any meta-information being present on the initial dataset. The only input required is a .csv file containing two columns: a unique ID for each document and the text of the document itself. The .csv file is then read using the popular Pandas library available in Python. For each document, the NLP library Spacy is then used to extract named entities (see Section 3.3.1) which are then stored in a PostgeSQL database. If required, the extraction process can be modified in `preprocess_data.py`.

Running the `create_db.py` script creates the required database structure, which can be seen in Figure 4.3. Again, this structure can be extended if necessary.

4.2.1 Omitting the preprocessing step

If the dataset that should be visualized already contains all required meta-information, the preprocessing step can be omitted. See Section 4.3.2 for details.

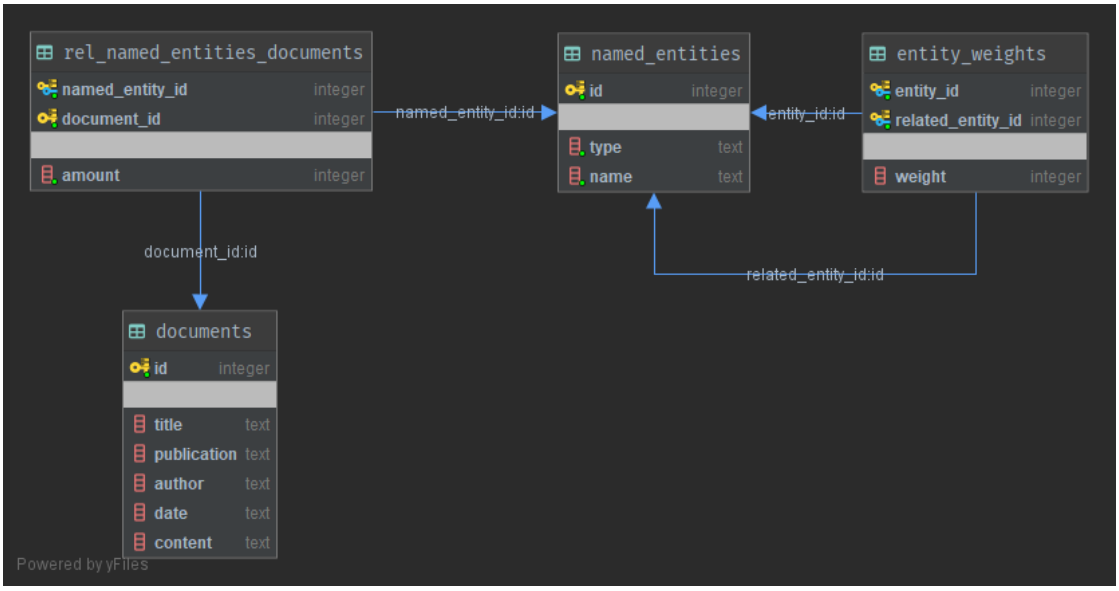

Figure 4.3: ER-diagram of the relational database containing all required meta-information about the dataset.

4.3 Back end

The back end is responsible for the more complex and CPU-heavy calculations of the prototype, like graph calculation and document score calculation. The following sections discuss these calculations, which were also outlined in Figure 4.2.

4.3.1 Network Repository

The `NetworkRepository` has two main responsibilities:

- **Graph data structure calculation**. Calculates the initial and filtered graph data structure following the algorithms outlined in Section 3.3.2. The graph data structure can be retrieved through the `calculateNetworkData` function, which accepts an optional `filter` argument of type `NetworkFilter`, which allows to filter by related entities as well as entity types. To improve performance and reduce visual clutter, entities and links can be limited by the constants `ENTITY_LIMIT` and `LINK_LIMIT`. Using those limits only returns the n heaviest entities or links respectively.
- **Document score calculation**. Calculates matching documents based on a set of entities following the algorithms outlined in Section 3.3.4. The matching document identifiers and matching rates (in percent) can be retrieved through the `calculateDocumentsForEntities` function, which accepts an array of entities for which the matching documents should be calculated. The amount of returned

documents can be limited using the `DOCUMENT_LIMIT` constant. If a limit is set, only the n best-matching documents will be returned. Note that this does not increase the calculation performance since all scores have to be calculated anyway.

4.3.2 Database Adapter

The `NetworkRepository` utilizes the Adapter Pattern to increase compatibility with various database implementations. In order to be compatible with the `NetworkRepository`, the `INetworkDatabaseAdapter` interface needs to be implemented (see Listing 4.1).

```
interface INetworkDatabaseAdapter {
    getEntityById(id: number): Promise<Entity | null>;
    getAllEntities(): Promise<Entity[]>;
    getRelatedEntities(id: number): Promise<Entity[]>;
    getLinksForEntity(entityId: number): Promise<EntityLink[]>
    getEntityCountsInDocument(documentId: number): Promise<EntityOccurrence
        []>
    getDocumentCount(): Promise<number>
    getAllEntityCounts(): Promise<EntityOccurrence[]>
    getDocumentsForEntityId(entityId: number): Promise<number[]>
}

type Entity = {
    id: number,
    name: string,
    weight: number
    type: EntityType
}

type EntityType = "PERSON" | "NORP" | "FAC" | "ORG" | "GPE" | "LOC" |
            "PRODUCT" | "EVENT" | "LAW"

type EntityLink = {
    source_entity: number,
    target_entity: number,
    weight: number
}

type EntityOccurrence = {
    entityId: number,
    count: number
}
```

Listing 4.1: INetworkDatabaseAdapter interface definition, including types.

4.3.3 Rest API interface

The Rest API interface is a basic express server located in the `server.ts` file. It exposes two endpoints:

- **POST /network-data**. Forwards the data returned from `NetworkRepository.calculateNetworkData`. The optional filter can be passed as a POST parameter.
- **POST /documents**. Forwards the data returned from `NetworkRepository.calculateDocumentsForEntities`. The entities required for the calculation have to be passed as a POST parameter.

The API interface does not hold any important logic and can be easily replaced by any other Javascript-based server solution.

4.4 Front end

The main responsibility of the front end is rendering and calculating the word cloud layout. Furthermore, it provides a graphical user interface for filtering the word cloud and displaying document results. It is split into multiple components with clearly defined interfaces, which allows them to easily be extended and adapted. The following sections discuss these topics, which were also outlined in Figure 4.2, in more detail.

4.4.1 NetworkComponent

The `NetworkComponent` holds all the logic required to render the word cloud layout. The implementation of the force-directed layout makes use of the `d3-force` module. However, to make the layout work, the solution deviates slightly from the concept presented in Section 3.3.3. We found that staggering the simulation into multiple sub-simulations that run sequentially yields significant performance improvements compared to applying all forces simultaneously. The best layout quality was achieved by using the following simulation sequence:

1. **Spreading**. The only purpose of the first part of the simulation is to spread the tags out to prevent as many initial overlaps as possible. The main goal is to prevent tags from being stuck inside each other since that causes issues when applying other forces, like collision detection or linking. The results of this step can be seen in Figure 4.4.
2. **Linking & Collision Detection**. After spreading the tags, a linking force is applied to bring related tags closer together. Simultaneously, collision detection is applied to prevent tags from overlapping due to being pulled too close together. For collision detection, the bounding box of the SVG is used. The results of this step can be seen in Figure 4.5.
3. **Collision Detection**. Since there are still many overlaps remaining after the previous step, the last part of the simulation only applies collision detection for a short amount of steps to smoothen the final layout. The results of this step can be seen in Figure 4.6.

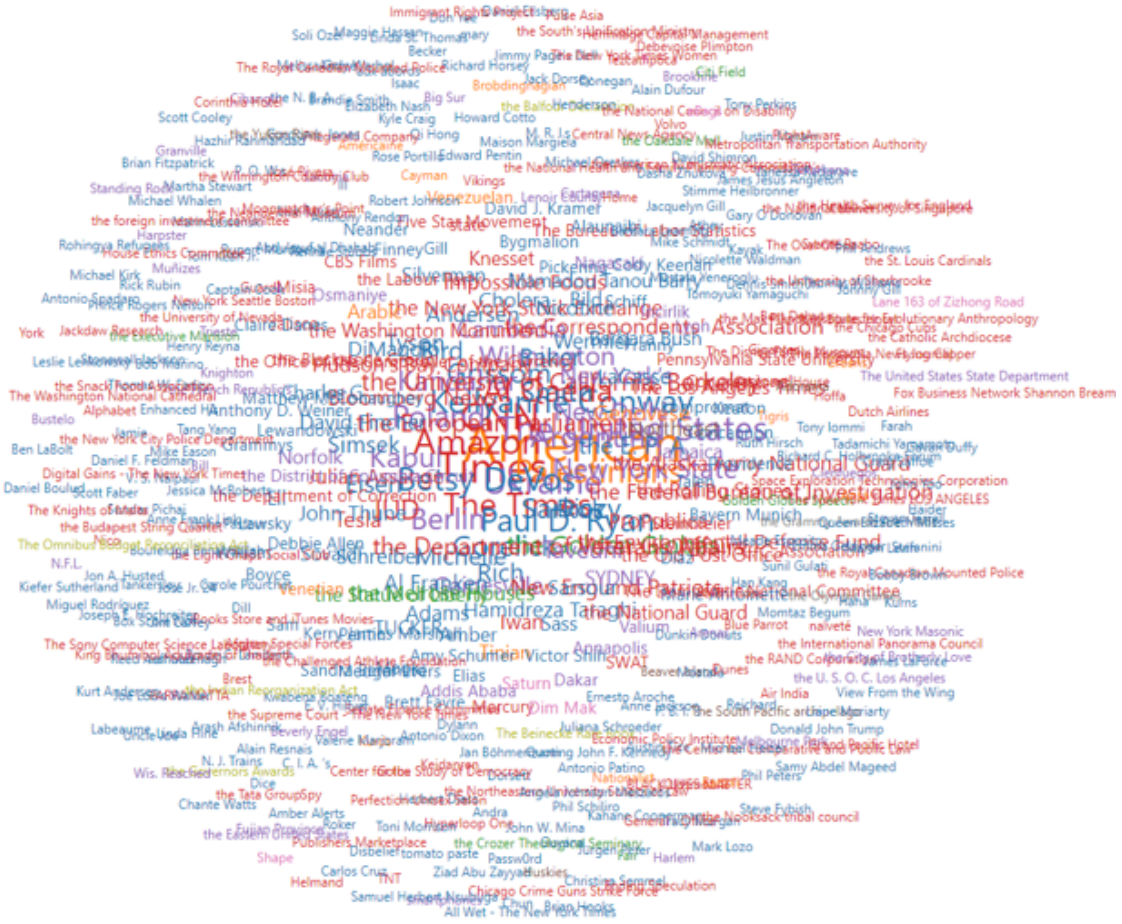

Figure 4.4: In the first simulation step, all tags are spread out to prevent initial overlaps, which can cause calculation problems in later steps.

The `NetworkComponent` then makes use of d3 transitions to create a smooth transition from one state of the word cloud to another (e.g. when applying a filter). This aims to reduce visual strain and make the layout changes more perceivable and coherent for the end-user.

4.4.2 Filter

The filter consists of the `Filter` class and the `FilterComponent`. The `Filter` class utilizes the Singleton Pattern to manage one filter state across the entire application. This class can be used to either apply the filter set by the `NetworkComponent` to other components or vice-versa.

The `FilterComponent` displays a basic filter interface and showcases proper use of the `Filter` class. It can easily be replaced by a more sophisticated filter interface if desired. For categories the `d3.schemeCategory10` color scheme has been used.

Figure 4.5: In the second simulation step, a linking force is applied to bring related tags closer together. Furthermore, a collision detection force is applied, to prevent tags from overlapping. As can be seen in (A), there are still some overlaps left after running the second simulation step.

4.4.3 Document List

The document list is an example implementation of a component that displays the resulting documents and their matching rate. It is tailored to the dataset described in Section 3.1 and includes metadata specific to the dataset like publishing date and author. To increase scannability, selected tags are highlighted.

Figure 4.6: The final simulation step applies only collision detection for a short amount of simulation steps to get rid of most of the remaining overlaps, resulting in the final layout being created.

CHAPTER 5

Results

This chapter aims to give an overview of the performance of the prototype in a real-world scenario as well as showcase how the prototype works with a second dataset.

5.1 Benchmarking Environment

All benchmarks were performed on a Dell XPS 15 Laptop with the following specs:

- **CPU** - Intel Core i7-8750H @ 2,20 GHz
- **RAM** - 16GB

For all benchmarks the front end, as well as the back end, were served directly from the laptop, meaning all calculations were also performed on the laptop. Performance in a real-world scenario can be assumed a bit better than what the benchmarks show.

To perform the benchmarks, a subset of 10,000 articles from the dataset described in Section 3.1 was used. After entity extraction, the database consisted of 116,908 unique entities and 13,317,120 links.

5.2 Performance

The following sections detail the performance of various parts of the prototype. Since performance is highly dependent on the characteristics of the dataset used, the values presented in the following sections are only intended to serve as rough guidelines.

5.2.1 Entity Extraction

The entity extraction process is the slowest and least optimized part of the prototype. This is mostly due to the fact that it only has to be run once per dataset and requires no user interaction, meaning that performance is negligible. For the **10,000 articles** used, extracting all **116,908 unique entities** and **13,317,120 links** took about **100 minutes**, meaning that entity extraction runs at $\approx 1.\dot{6}$ **documents per second**.

5.2.2 Graph Calculation

This benchmark showcases the performance of the calculation of the initial graph layout with any combination of 0 to 5000 links and 0 to 500 entities. As explained in Section 4.3.1, these parameters only take the n heaviest links or entities into account. Calculation times range from **133ms** to **19,067ms** depending on how many entities and links are considered. As can be seen in Figure 5.1, entities have by far the biggest impact on calculation time, with anything above 300 entities making the user interface feel very sluggish. However, rendering that many entities in the initial layout also causes a lot of visual clutter, which is why the default limit for entities is set to 200, where the performance of the graph calculation is also still below 1 second.

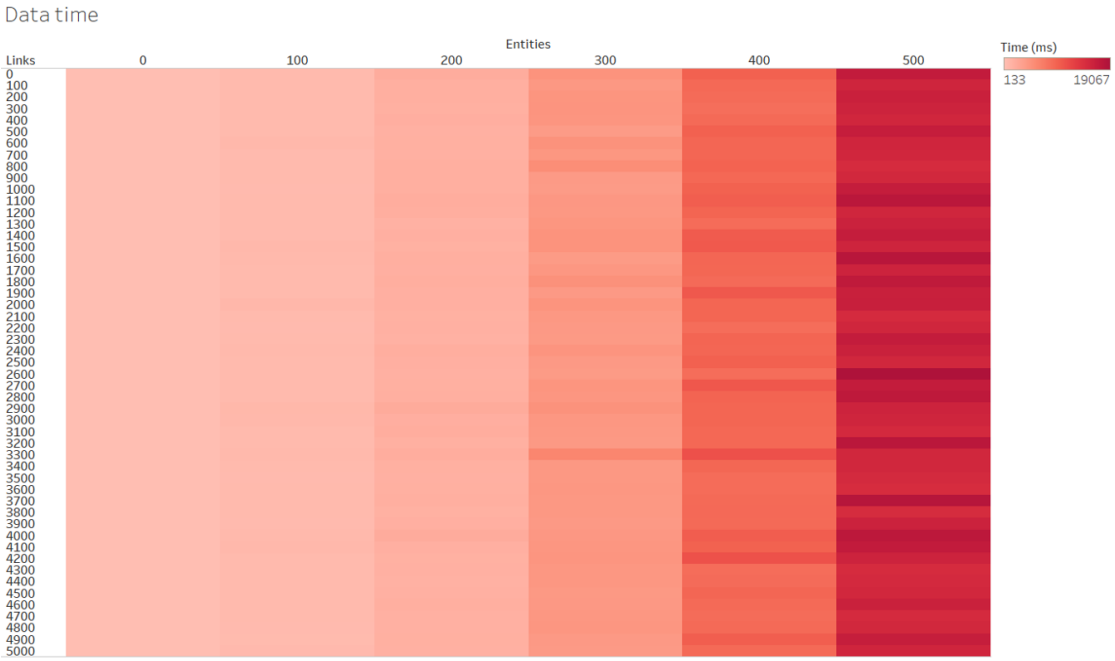

Figure 5.1: Benchmarks for graph layout calculation with 0 to 5000 links and 0 to 500 entities.

Calculations of subsequent layouts based on filters only render small subgraphs compared to the initial layout and consistently yielded results around 200ms to 500ms. These

calculations always consider the entire dataset and therefore no parametrization was necessary.

5.2.3 Document Retrieval

The performance of the document retrieval process is mainly dependent on the number of entities selected. Calculation times ranged from 144ms to 324ms depending on how many entities were selected. However, even under extreme conditions, like 500 selected entities, which is highly unlikely in a regular scenario, document retrieval still only took 324ms.

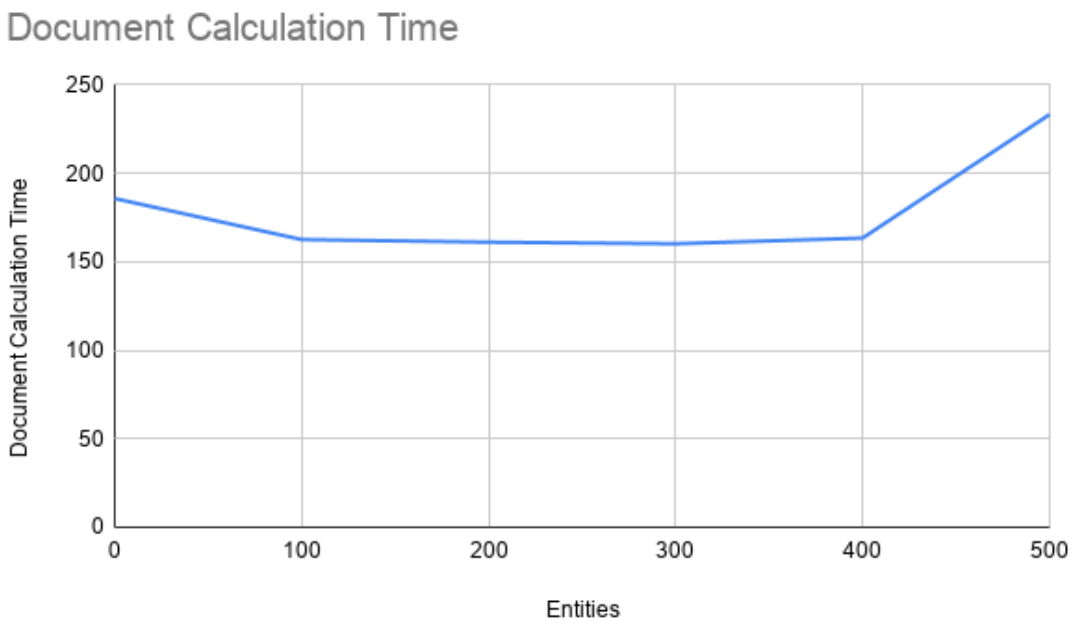

Figure 5.2: Benchmarks for document retrieval with 0 to 500 entities and 50 samples per parameter.

5.2.4 Initial Rendering

The last and probably most important benchmark is the time it takes from when a user accesses the page to when the word cloud is actually displayed in the browser. As mentioned in Section 5.2.2, subsequent rendering of subgraphs after filtering are really fast and render in under 500ms total. Therefore, only the initial render is considered in this section. As can be seen in Table 5.1 and Figure 5.3, the total rendering time ranges from 1,886ms to 20,691ms. The performance scales badly with an increasing amount of entities, which is mainly due to the issues mentioned in Section 5.2.2. While this can

Entities	Client (ms)	Server (ms)	Total (ms)
100	1,300	600	1,900
200	1,300	1,700	3,000
300	1,400	4,000	5,500
400	1,300	8,600	9,900
500	1,600	19,000	20,600

Table 5.1: Benchmarks for requesting, calculating, sending, and rendering the initial word cloud with 100 to 500 entities. The "Total" column represents the time it takes from when the user accesses the page until the word cloud is displayed.

possibly be optimized, the recommended amount of 200 entities renders in roughly three seconds.

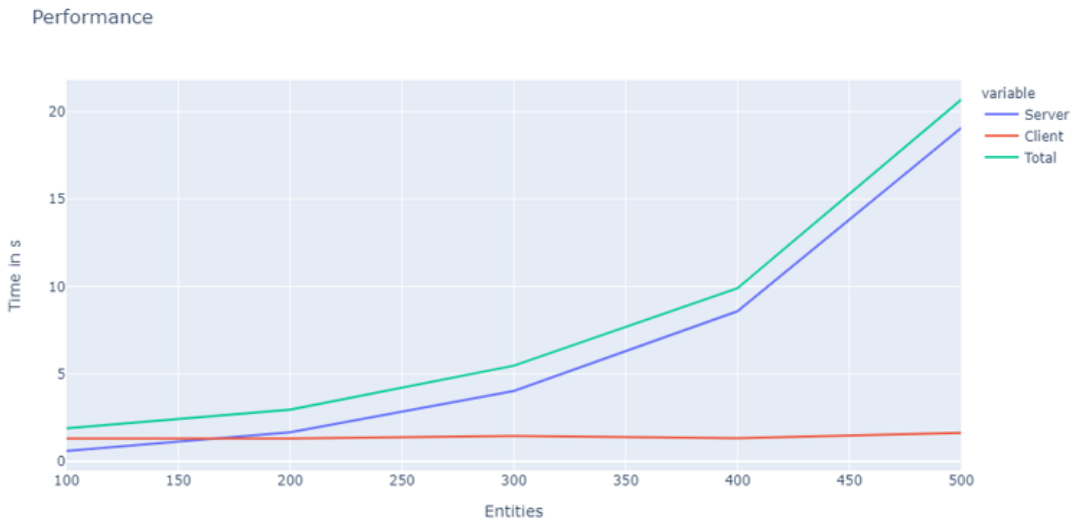

Figure 5.3: Line chart visualizing the data shown in Table 5.1.

5.3 Evaluation of another Dataset

To evaluate the versatility of the prototype, a second dataset was processed. The chosen dataset was the Visualization Publication Dataset [IHK+17] from which the title and the abstract were used for entity recognition. The results of the initial word cloud can be seen in Figure 5.4. This figure also showcases the prototype's debug mode, which displays links as lines in the word cloud.

Utilizing this debug mode one can quickly see, that the dataset does not seem to be very connected, making it not very suitable for exploratory search. This is proven further

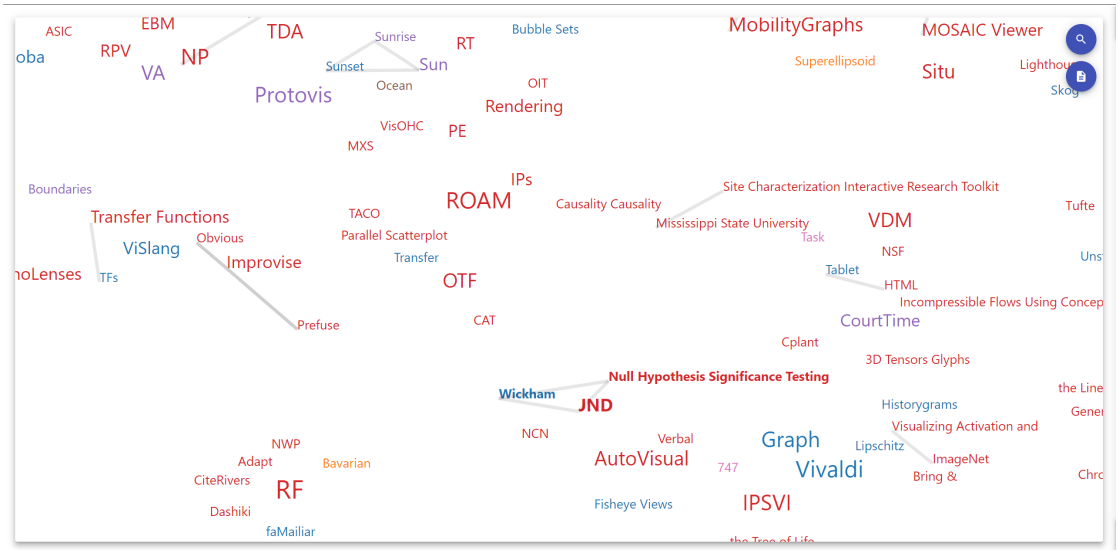

Figure 5.4: Zoomed in view of the Visualization Publication Dataset [IHK+17]. This view also showcases the use of the prototype's debug mode, which displays links as lines with varying opacity.

after analyzing the results of the preprocessing step, which reveals that almost a third of the 3394 processed abstracts do not contain a single named entity, and the rest of the abstracts contain only two named entities on average. This could either mean that the used NLP library is not suited to extract named entities from scientific texts or due to the abstracts being rather short.

This example showcases that the prototype is very useful to determine if a dataset is suitable for exploratory search. The combination of having a visual representation of the data on one hand, and the means to query the dataset for statistical information using SQL on the other hand, allowed for a quick evaluation, which is exactly what the prototype was designed for.

CHAPTER 6

Discussion, Conclusion & Future Work

This chapter discusses the results of the thesis as well as possible improvements to the prototype and future work.

6.1 Discussion & Conclusion

This section discusses the prototype resulting from this thesis, focusing on the following points:

- **Implementation** - Is the prototype working as intended?
- **Modularity** - Can the prototype be adapted to suit different user needs?

6.1.1 Implementation

To ensure the prototype is working as intended, the following steps were taken:

- **Preprocessing**. Manual tests with small datasets have been conducted to ensure that the extracted entities and their relations to the documents are being stored properly.
- **Back end**. Automated tests using Jest have been written to ensure that all algorithms are working as intended.
- **Front end**. Manual tests with small datasets have been conducted to ensure that the word cloud is being rendered properly. In addition, a debug flag has been added to the `NetworkComponent`. If set, this flag shows links as lines varying in

thickness and opacity depending on their strength. This can be used to verify that the word cloud has been drawn correctly. However, this proved to be the hardest part of the application to test, making it the most likely part to contain bugs and errors.

6.1.2 Modularity

Making the prototype adaptable and modular was one of the main goals of this thesis. This goal was achieved in the following areas of the prototype:

- **Preprocessing**. This step is entirely optional if the required data has already been extracted by other means. If required, this step works with a basic .csv file, arguably making it very easy to use.
- **Database Access**. Database access is completely separated from the entire application, making the entire back end compatible with almost any database through the Adapter Pattern. This resulted in a potential performance cost since query optimization is not possible. Furthermore, query languages like GraphQL that might be better suited for graph calculations cannot be fully leveraged because of this abstraction layer. Considering that the performance of the prototype is still decent and that its main purpose is to serve as an evaluation tool, this tradeoff is worth it in our opinion.
- **Rest API**. The existing web framework (express.js) can be easily replaced by any other javascript-based web framework.
- **Front end**. The component-based architecture of React makes it easy to use the components in any React app. Outside of React, this becomes a bit more complicated but can still be done by utilizing WebComponents or adding React to your project.

6.2 Future Work

While the prototype is definitely suitable for a general evaluation of exploratory search interfaces and can be adapted to individual needs, there are still many improvements that could be made which would benefit almost every use case of the prototype:

- **Entity Grouping**. Currently, extracted entities that are of similar origin are still treated as distinct entities (e.g. “Trump”, “Donald Trump” and “Donald J. Trump”, all refer to a president of the USA, so they should be combined into one entity). However, this is not a trivial task, since other entities like “New York”, “New York Times” and “New York Yankees” should not be grouped, despite their names being similar.

- **Database Setup**. The current setup of the prototype requires the user to either set up their own database and connect it via the `INetworkDatabaseAdapter` interface (see Section 4.3.2) or to use a blank PostgreSQL database. Setting up PostgreSQL can be a cumbersome process for some users and could be streamlined in some way, maybe through using a Docker image or another container solution.
- **Graph Layout Calculation**. Currently, the force-directed layout is calculated on the client. While this works quite well, doing the calculations on the back end should yield better performance. Moving the layout calculation to the back end would also allow for further optimizations, like caching the initial layout for a faster initial loading time and better user experience.
- **Evaluation**. The most important future work that should be conducted is a user study to evaluate the usability of this exploratory search interface. One of the main goals of this thesis was to provide a prototype that can be adapted to a wide variety of datasets to make an evaluation as easy as possible.
- **Front End Components**. For better encapsulation, the React components could be provided as an npm-package.

Bibliography

[CWL+10] Weiwei Cui, Yingcai Wu, Shixia Liu, Furu Wei, Michelle X Zhou, and Huamin Qu. Context preserving dynamic word cloud visualization. In *2010 IEEE Pacific Visualization Symposium (PacificVis)*, pages 121–128. IEEE, 2010.

[DCCW08] M. Dörk, S. Carpendale, C. Collins, and C. Williamson. Visgets: Coordinated visualizations for web-based information exploration and discovery. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1205–1212, 2008.

[Hea09] M.A. Hearst. *Search User Interfaces*. Cambridge University Press, 2009.

[HLLE14] F. Heimerl, S. Lohmann, S. Lange, and T. Ertl. Word cloud explorer: Text analytics based on word clouds. In *2014 47th Hawaii International Conference on System Sciences*, pages 1833–1842, 2014.

[HPP+20] M. A. Hearst, E. Pedersen, L. Patil, E. Lee, P. Laskowski, and S. Franconeri. An evaluation of semantically grouped word cloud designs. *IEEE Transactions on Visualization and Computer Graphics*, 26(9):2748–2761, 2020.

[IHK+17] Petra Isenberg, Florian Heimerl, Steffen Koch, Tobias Isenberg, Panpan Xu, Chad Stolper, Michael Sedlmair, Jian Chen, Torsten Möller, and John Stasko. vispubdata.org: A metadata collection about IEEE visualization (VIS) publications. *IEEE Transactions on Visualization and Computer Graphics*, 23(9):2199–2206, September 2017.

[Kag18] Kaggle. All the news, https://www.kaggle.com/snapcrack/all-the-news, 2018.

[LAB+10] Yiling Lin, Jae-Wook Ahn, Peter Brusilovsky, Daqing He, and William Real. Imagesieve: Exploratory search of museum archives with named entity-based faceted browsing. *Proceedings of the American Society for Information Science and Technology*, 47(1):1–10, 2010.

[Lid01] Elizabeth D Liddy. Natural language processing. 2001.

[SKK+08] C. Seifert, B. Kump, W. Kienreich, G. Granitzer, and M. Granitzer. On the beauty and usability of tag clouds. In *2008 12th International Conference Information Visualisation*, pages 17–25, 2008.

[WMM08] Ryen W White, Gary Marchionini, and Gheorghe Muresan. Evaluating exploratory search systems. *Information Processing and Management*, 44(2):433, 2008.

[XTL16] J. Xu, Y. Tao, and H. Lin. Semantic word cloud generation based on word embeddings. In *2016 IEEE Pacific Visualization Symposium (PacificVis)*, pages 239–243, 2016.