TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Chair of Databases

DEVELOPING A WEB FRONT END FOR A WORD-EMBEDDING-ENABLED RELATIONAL DATABASE SYSTEM

Zdravko Yanakiev
Born on: 5th April 1995 in Sofia
Matriculation number: 4037330
Matriculation year: 2014

BACHELOR THESIS

to achieve the academic degree

BACHELOR OF SCIENCE (B.SC.)

Supervisor
Dr.-Ing. Maik Thiele
Supervising professor
Prof. Dr.-Ing. Wolfgang Lehner

Submitted on: 10th July 2018

ABSTRACT

Word embedding is a method in the field of natural language processing which maps words or phrases to vectors of real numbers. Vectors generated using this method, which are also called word embeddings, encode many of the input data's semantic and syntactic features. They can also be manipulated using vector operations to reveal further regularities between words or phrases. Word embedding has found numerous applications, including such in recommendation algorithms, machine translation and sentiment analysis. In previous work, a database system based on PostgreSQL called FREDDY was developed to integrate the functionalities of word embeddings into a relational database management system. However, the database system lacked a dedicated graphical front end exhibiting its special features. This thesis documents the development of a web front end for FREDDY. The web application was implemented in JavaScript using the web frameworks AngularJS and Express.js. The developed front end allows the user to explore several data sets by running word-embedding-enabled SQL queries on them using word embeddings generated from various sources. Furthermore, it provides options for the user to choose between different approximation methods used for a query's execution, adjust their parameters and inspect the effects of their actions on a query's results and the system's performance and precision.

STATEMENT OF AUTHORSHIP

I hereby certify that I have authored this Bachelor Thesis entitled *Developing a Web Front End for a Word-Embedding-Enabled Relational Database System* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 10th July 2018

Zdravko Yanakiev

ACKNOWLEDGMENTS

First and foremost, I would like to thank my supervisors, Dr.-Ing. Maik Thiele and Dipl.-Inf. Michael Günther, for the extraordinary support and advice they gave me. I am grateful to my parents for giving me life, for always encouraging me in all of my pursuits and for supporting me during my entire life. I would also like to thank to my sister Iskra who has always been both a huge influence for me and a great friend. Last but not least, I would like to thank Betani for all her love and for always being there for me.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 MOTIVATION

We live in the Age of Big Data. A blog post by Bernard Marr published in May 2018 cites intriguing statistics about the quantity of data created by mankind which could only confirm this statement. Over 90 percent of the data in the world was generated over the last two years alone. Currently, 2.5 quintillion bytes of data are created each day, and this rate is expected to accelerate. Furthermore, every minute 16 million text messages and 156 million emails are sent [1].

Extracting relevant information from such a huge volume of data proved itself a complex, time- and resource-consuming task. A large part of the business-relevant information in data is contained in so-called *unstructured data* [2], that is, information that is not organized in any predefined manner, primarily in the form of text, such as the aforementioned text messages and emails. Developing special techniques to find patterns in unstructured data and interpret it has been a major focus of computer science research in the last few decades, as efficient processing of texts using computers turned out to be particularly challenging. Examples for areas in which such techniques are being developed are *data mining*, *natural language processing* and *text analytics*.

Among these, natural language processing is especially relevant to this thesis. Natural language processing is defined as "*a branch of artificial intelligence that deals with analyzing, understanding and generating the languages that humans use naturally in order to interface with computers in both written and spoken contexts using natural human languages instead of computer languages*" [3]. One of the approaches in the field of natural language processing is *word embedding*, a method which maps words or phrases to vectors of real numbers, allowing the efficient processing of enormous amounts of textual data by computers. Its applications include accurate machine-aided translation [4], sentiment analysis [5] and phrase auto-completion [6].

Despite the rise of other database models, relational database management systems (RDBMS) still dominate the database market. As of May 2018, four out of the top five most popular database management systems are relational ones [7]. Using the possibilities for information retrieval provided by word embedding could provide new insights into the semantic content of existing and new relational databases. In order to achieve this, systems extending queries on relational databases with new functions that use operations on word embeddings have to be developed. The development of one such system based on *PostgreSQL*[1], called *FREDDY*[2] (**F**ast Wo**r**d **E**mbed**d**ings in **D**atabase S**y**stems), was started

[1] https://www.postgresql.org/

[2] https://wwwdb.inf.tu-dresden.de/research-projects/freddy/

in 2017 at the Technische Universität Dresden by Michael Günther. It *"is able to use word embeddings [to] exhibit the rich information encoded in textual values"* [8] by extending SQL queries with operations on word embeddings.

1.2 GOAL

The developed system proved itself to be effective and resource-efficient. However, it suffered from the lack of an intuitive, user-friendly interface. Interacting with FREDDY required the use of traditional PostgreSQL tools, such as the terminal-based front end *psql*. Such tools are unfitting for use cases such as investigating the system's capabilities and analyzing its performance. A dedicated user interface for FREDDY has to be both well-suited for use cases such as these and allow the intuitive adjustment of specific parameters used in the system's algorithms. The goal of this bachelor thesis project is to develop a graphical user interface for FREDDY demonstrating its features in the form of a web application. The finished *demo application* has to fulfill several objectives:

1. it has to allow the user of the application to test the system's features by *exploring several different example datasets* and *running word-embedding-enabled SQL queries* on them;

2. the web application also has to offer the functionality of *visualizing FREDDY's word embeddings operations' performance* in real time;

3. it should provide a transparent way to *adjust options and parameters* of the database management system powering it;

4. the demo application is supposed to use *modern web technologies and frameworks* and to conform to *contemporary user interface design principles* in order to ensure a smooth user experience.

This bachelor thesis describes in detail the development of the aforementioned web application.

1.3 STRUCTURE

The thesis follows a bottom-up structure, starting with the research and technologies based on which the demo application was built, proceeding to analyze its related core requirements, then detailing the architecture of its implementation, and finally providing a walkthrough of the finished software product's interface and an evaluation of the fulfillment of its requirements.

In the next chapter of this thesis (Chapter 2), I introduce the fundamental natural language processing and database concepts and technologies that are key to the demo application's purpose. Chapter 3 proceeds to describe the use cases for the developed application and an example scenario of a user's interaction with it and derives a list of functional and non-functional requirements from them. In Chapter 4, I detail the architecture of the software solution and its implementation specifics, including used technologies, programming languages, frameworks and external libraries. This chapter also contains a description of the data sets used in the application's deployment. Chapter 5 provides a walkthrough of the finished application's graphical user interface. Furthermore, I assess the quality of its user interface according to contemporary user experience (UX) design principles and evaluate the other non-functional requirements for it. Chapter 6 examines the possibilities for the further development of the demo application and concludes this bachelor thesis.

2 FUNDAMENTALS

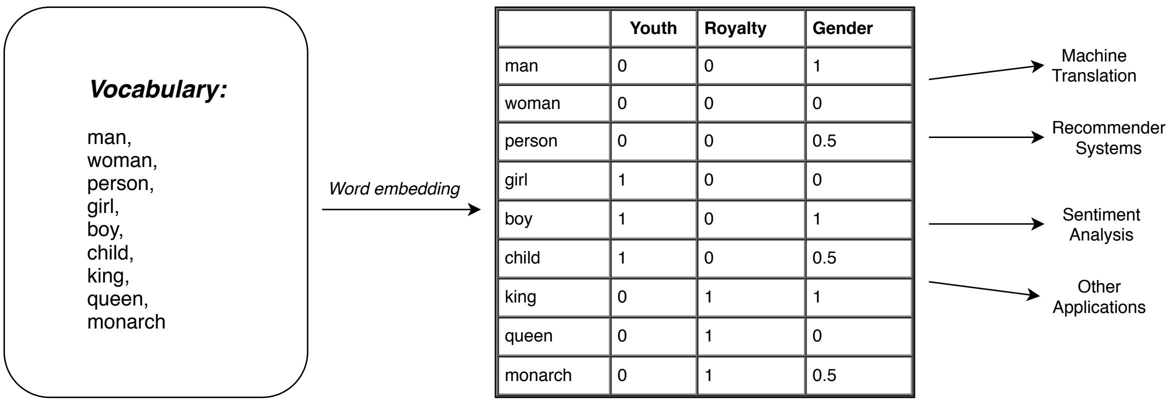
In this chapter, I present the fundamental natural language processing concepts relevant to this thesis (*word embedding*) and previous work focused on integrating their capabilities into relational database management systems (*FREDDY - Fast Word Embeddings in Database Systems*).

2.1 WORD EMBEDDING

Word embedding is considered one of the key breakthroughs in using deep learning methods to challenge natural language processing problems [9]. Here, I provide a short overview of its concept and list several of its practical applications.

2.1.1 DEFINITIONS AND CONCEPT

The term *word embedding* denotes a number of language modeling and feature learning techniques used in natural language processing. Their common feature is that they map words or phrases from a vocabulary to vectors of real numbers in a vector space. Generating the word-vector mappings may be done in different ways, e.g., by using neural networks, probabilistic models or explicit representation in terms of the context in which words appear. Most commonly, large text corpora are processed by neural networks to generate the word vectors, which is a time-consuming task. Because of this, word embeddings data sets readily available on the Internet (listed in Section 4.2.1) were used for this project's purposes.



	Youth	Royalty	Gender
man	0	0	1
woman	0	0	0
person	0	0	0.5
girl	1	0	0
boy	1	0	1
child	1	0	0.5
king	0	1	1
queen	0	1	0
monarch	0	1	0.5

Figure 2.1: A figure illustrating word embedding and its applications.

The term *word embeddings* is also used for the vectors generated using such methods. Each word embedding corresponds to a single word or phrase of the input corpus. They capture the semantical, morphological, contextual or hierarchical information contained in the corpus' vocabulary. Depending on the technique used, they may reflect the structure of a word in terms of morphology, serve as a word-context representation or show the relationship between a set of documents and the terms contained in them [10]. Each element of a vector represents a different aspect, or feature, of a word. Word embedding vectors are usually dense and low-dimensional (up to several hundred dimensions). The idea behind using dense vectors is generalization, which is hard to obtain when using a sparse representation. For example, in *one-hot encoding*, a word-representing vector has a length equal to the vocabulary's size and consists only of zeros, except for a one in the position standing for the word. This representation cannot encode the complex semantical relations between words, as the number of features of a word is much smaller than the size of the vocabulary. Thus, a dense representation such as word embedding is well-suited to capture the likeness between words having similar features [11] [12]. This results in words used in a similar way also having a similar word embedding representation (see Figure 2.1, where words are represented as three-dimensional vectors based on deduced features).

2.1.2 APPLICATIONS

Word embeddings have found many applications in various fields. As word embeddings are numerical representations of similarities between words, vector operations (henceforth called *word embedding operations*) can be used to manipulate them and grant insights into previously unavailable semantic information. As a basic example, adding the vector for *king* to the vector for *woman*, then subtracting the vector for *man* from the sum may result in a vector corresponding to *queen*.

The real-valued vector space into which words were embedded comes with the notions of distance and angle. These notions also extend to embedded words and provide information, e.g., about relationships between them and their similarity [13]. For example, a team at Google Inc. developed an algorithm called *word2vec* which learned to distinguish between gendered forms of nouns and found out the difference between singular and plural forms of nouns in English. The same algorithm also discovered the existence of a relationship between a country and its capital. Interestingly, the team developing *word2vec* also found that the learned structure for one language often correlates to that of another language. This opens new possibilities of using word embeddings for *machine translation*, a traditionally challenging problem [14].

Word embeddings can be used in recommendation algorithms by various businesses. Among others, the music streaming application *Spotify* uses them to provide new music recommendations to its users [15]. Furthermore, Google's search engine makes use of *word2vec* as part of its recently developed RankBrain search algorithm [16].

Sentiment analysis, the process of determining the emotional tone behind textual information, is another field of application of word embeddings. One and the same word may carry a different connotation depending on the context of its use. For example, *soft* may be used in a negative way when talking about a sport player's performance, but have a positive meaning when discussing stuffed animals [5].

The numerous possible applications of word embeddings have sparked an ongoing research interest in them, with major companies such as Facebook developing their own word embedding algorithms [17]. However, work on integrating operations on word embeddings into relational database management systems, thus enhancing robust, well-established systems by word embedding's unique functionalities, has been both much needed and practically non-existent until recently.

2.2 FREDDY - FAST WORD EMBEDDINGS IN DATABASE SYSTEMS

The web application developed in the course of this thesis uses *FREDDY* [18], one of the few systems integrating word embeddings into relational database management systems, as its data source. In this section, I give a short overview of its architecture and the functionalities it provides. I also list the different word embedding operations implemented as a part of FREDDY and used in the demo application. A detailed description of the developed system including an evaluation of its performance can be found in [19].

2.2.1 ARCHITECTURE AND FEATURE OVERVIEW

FREDDY is based on a *PostgreSQL* database management system. It provides facilities for importing, storing and managing word embeddings and executing operations on them. FREDDY uses relational tables to store multiple sets of word embeddings (for example, for different languages within the same database), as well as the index structures needed for the execution of word embedding operations. The system also provides a convenient way to create the tables required for its operation by including a set of import scripts written in Python.

Another important part of FREDDY is a PostgreSQL extension containing the word embedding operations implemented in PL/pgSQL as PostgreSQL *user-defined functions* (UDFs). For a list of the word embedding operations provided in the extension, see the next subsection. On one hand, the UDFs wrap a shared library written in C containing their native implementations. On the other hand, some of them use the native library's functions to provide new functionalities to the system's user. After installing the extension into a regular PostgreSQL system and importing the data needed for its functioning, one can use the word embedding UDFs as a part of normal SQL queries on arbitrary textual data.

FREDDY offers the user control over the balance of performance and precision of word embedding operations. It provides two indexes for the improvement of word embedding operations' performance: *Product Quantization* (PQ) and *Inverted File System with Asymmetric Distance Computation* (IVFADC), both of which allow the fast calculation of approximated distances between vectors at the cost of reduced precision.

To better the precision of indexed word embedding operations, FREDDY also offers an option to use additional *post-verification* (PV). An indexed search using post-verification finds a number of results equal to the product of the *post-verification factor* and the requested number of results. The results are then additionally filtered by calculating their exact similarities to the search term. In the end, only the most similar ones are left and the requested number of results is provided to the user.

To enable the use of one of the indexes or additional post-verification and adjust their parameters (in case IVFADC or PV is enabled), one must only invoke the respective UDF in order to apply the settings database-wide.

2.2.2 WORD EMBEDDING OPERATIONS

The following word embedding operations provided by FREDDY are used in the demo web application:

- ***Cosine similarity***: measures the similarity between two words by computing the cosine of the angle between their respective embeddings. A similarity query on a word embeddings dataset may reveal that the most similar words to *comedy* are *comedic*, *sitcom* and *satire*.

- ***k-Nearest neighbor search (k-NN)***: finds the top *k* nearest vectors to a certain vector. In other words, a k-NN query is able to identify the top *k* most similar words to a specific word in a word embeddings data set. For example, a k-NN query with the keyword *Beatles* may return a list containing classic rock bands such as *Led Zeppelin*, *Pink Floyd* and *The Rolling Stones*. FREDDY also provides functions for the batch execution of multiple k-NN queries at once and for narrowing the k-NN search to a specific set of words.

- *Analogy queries*: provided three tokens `a`, `b` and `c`, an analogy query finds a token `d` whose relationship to `c` is most similar to the one between `a` and `b`. For example, one could execute an analogy query representing the director-film relationship. In this case, `a` could be the film *Godfather* and `b` the filmmaker *Francis Ford Coppola*. Executed on a database table containing film data, the query would iterate over the film titles as the parameter `c` and find the respective analogy token `d`. A list of films and their respective directors would be returned. For instance, the algorithm could return the director *David Lynch* for the film *Blue Velvet*. As with k-NN search, analogy queries also allow restricting the possibilities for the analogy token to a user-specified set of words. FREDDY provides several different implementations of analogy queries, too.

- *Grouping queries*: a grouping query matches a list of tokens to a list of group tokens. For instance, one could use a grouping query to match a list of films to the respective continents where they were made or a list of music bands to their respective music genres.

All of the listed operations can use an index to improve their performance. Additionally, k-NN search results can be narrowed down by post-verification for better precision. Post-verification and the index used for a query are selected automatically by FREDDY based on the user-specified global settings.

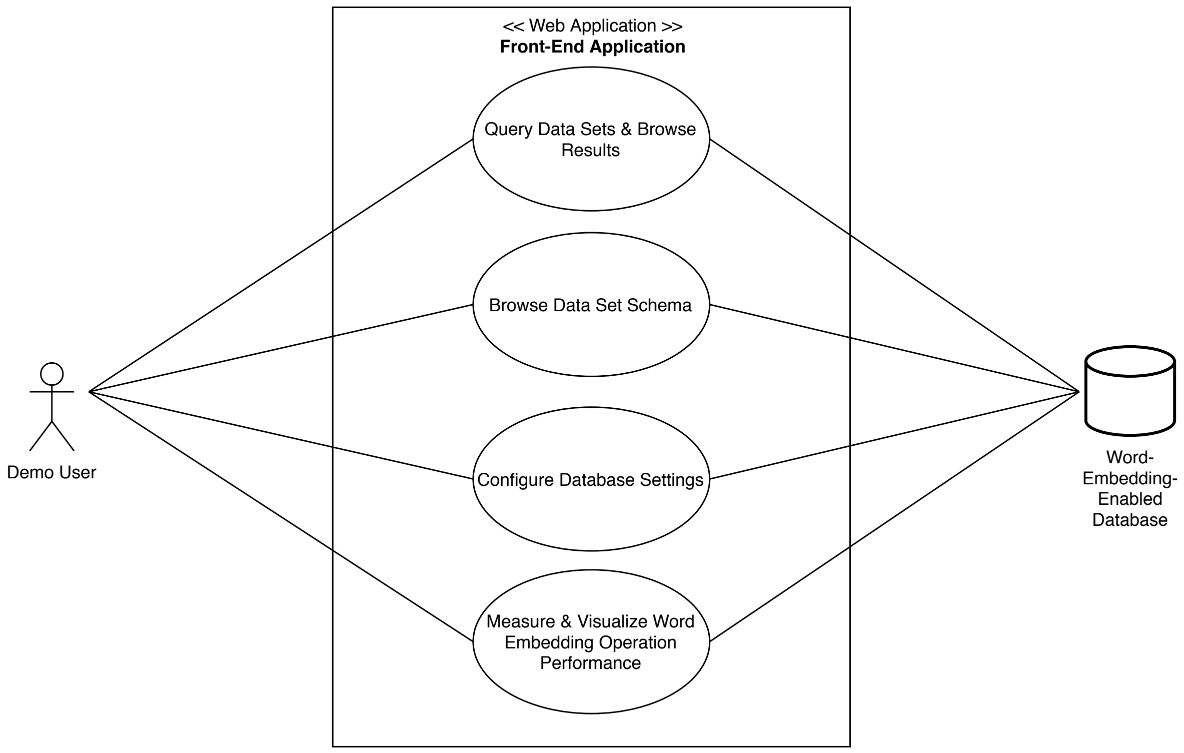
3 USE CASES AND REQUIREMENTS

Traditional tools for interacting with relational databases were built only with the relational model in mind. Using them to accomplish use cases specific to a word-embedding-enabled database, such as switching between different word embeddings data sets or measuring the performance of a word embedding operation, can be done only in an awkward, user-hostile manner. Specifically, every task which the user wishes to complete has to be broken down into a series of relational queries, which are then to be executed sequentially. Using the PostgreSQL command-line front end *psql* to switch to a different set of word embeddings, set search factors, execute a k-NN query and sort the results alphabetically requires the user to type in and execute a series of SQL queries such as the following:

```
-- Switch to the word2vec word embeddings data set:
SELECT init('google_vecs', 'google_vecs_norm', 'pq_quantization', '
    pq_codebook', 'fine_quantization', 'coarse_quantization', '
    residual_codebook');
-- Use the IVFADC index with post-verification for future k-NN queries:
SELECT set_knn_function(k_nearest_neighbour_ivfadc_pv);
-- Set global post-verification factor to 20:
SELECT set_pvf(20);
-- Set the number of nearest coarse quantization clusters for IVFADC search
    to 3:
SELECT set_w(3);
-- Finally, execute a k-NN query and order the results alphabetically:
SELECT DISTINCT title, ANN.word
        FROM imdb.title,
                        knn(regexp_replace(title, '␣', '_', 'g'), 5) AS ANN
        ORDER BY title;
```

Such a use case requires knowledge of both the SQL programming language and word embedding operations available in the database management system and prior information about the database's schema. Furthermore, the user is forced to scroll through a list of possibly thousands of query results in a terminal with no possibility to change their order or filter them. The command-line interaction paradigm is inappropriate for exploratory use cases where the user is supposed to be able to freely navigate through an interface and interact with it at a whim. Obviously, command-line tools are not well-suited for the specifics of a word-embedding-enabled database. A graphical web front end using interaction paradigms familiar to the user and requiring minimal technical knowledge would be much better suited to overcome the challenges presented by a word-embedding-enabled database and support exploratory use cases.

In the rest of this chapter, I analyze the use cases for a front-end web application for a word-embedding enabled database system using a series of diagrams. I present an example of a complete user interaction with the finished web application and determine the func-



<< Web Application >>
Front-End Application
Query Data Sets & Browse Results
Browse Data Set Schema
Configure Database Settings
Measure & Visualize Word Embedding Operation Performance
Demo User
Word-Embedding-Enabled Database

Figure 3.1: A diagram showing the use cases for the web application.

tional and non-functional requirements for the application, on which the project's evaluation in Chapter 5 is based.

3.1 USE CASES

In Figure 3.1 above, the different use cases for the web application are shown as a UML use case diagram. There are two chief actors: the *demo user* and the *word-embedding enabled database*. The application developed in this thesis, here called the *front-end application*[1], serves as an intermediary between them. The demo user interacts with the web application via a graphical web interface, while the web application is responsible for communicating with the database, sending queries to it and processing and presenting data received from it. The *word-embedding enabled database* is where the different data sets (both the ones containing word embeddings and domain-specific ones) are stored and word embedding operations are executed.

Four main use cases for the web application have been identified, which I describe one by one:

U1 querying domain-specific data sets using word embedding operations and browsing query results;

U2 browsing the schema of the selected domain-specific data set;

U3 configuring the database's word-embedding-specific settings;

U4 measuring and visualizing a word embedding operation's performance.

[1] *"Front-end"* here is used relative to the database system. The web application's actual architecture comprises both a front end and a back end in the web development sense.

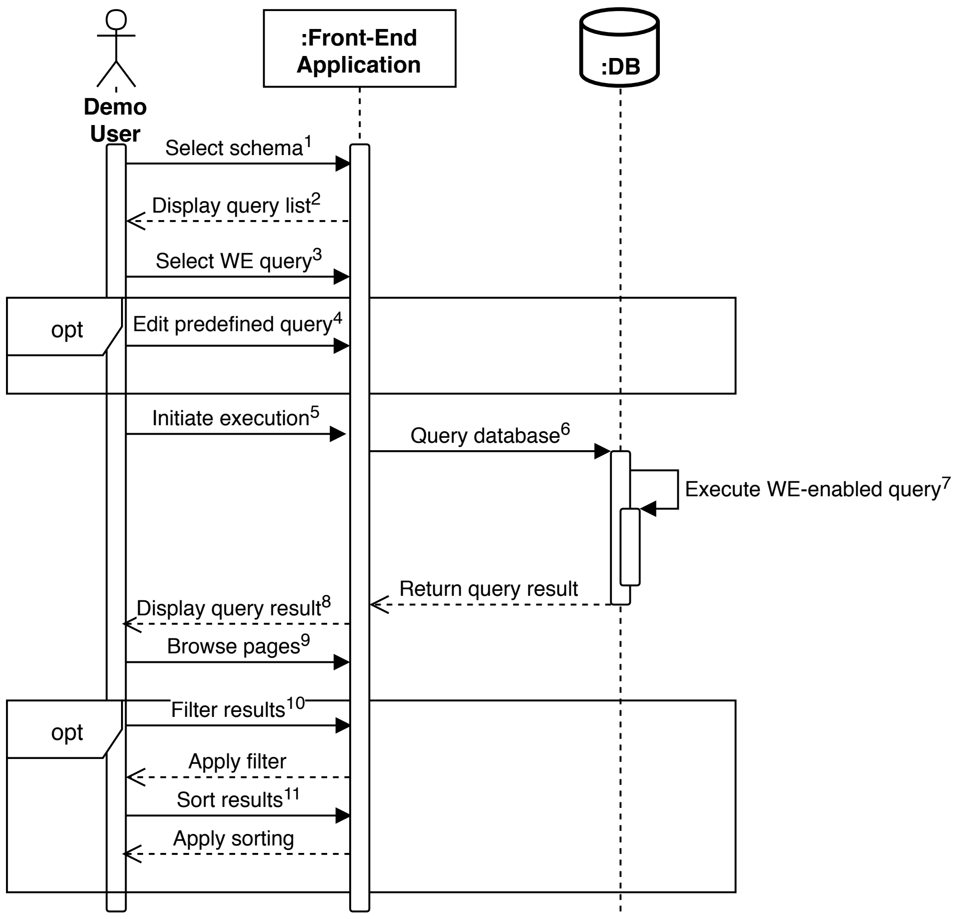


Figure 3.2: A sequence diagram for the use case of querying a word-embedding-enabled database.

In the following descriptions of the use cases, it is assumed that both the web application and the database system are up and running and the user has loaded the web application using their browser.

[U1] QUERY DATA SETS AND BROWSE RESULTS

A FREDDY system, just like any PostgreSQL system, is able to store arbitrary relational data sets, regardless of their specific domain. The web application should allow the user to execute word-embedding operations on textual data contained within them and present a query's results clearly. Figure 3.2 shows a scenario for this use case.

1. The user selects the data set to be queried from a list of schemas.
2. The web application automatically offers the user a list of predefined word-embedding-enabled queries relevant to the selected data set.
3. The user selects a predefined query.
4. *Optionally, the user edits the predefined query according to their intent using a built-in text editor.*
5. The user initiates the execution of the word-embedding enabled query.
6. The web application sends the user-selected query to the word-embedding-enabled database system for execution.

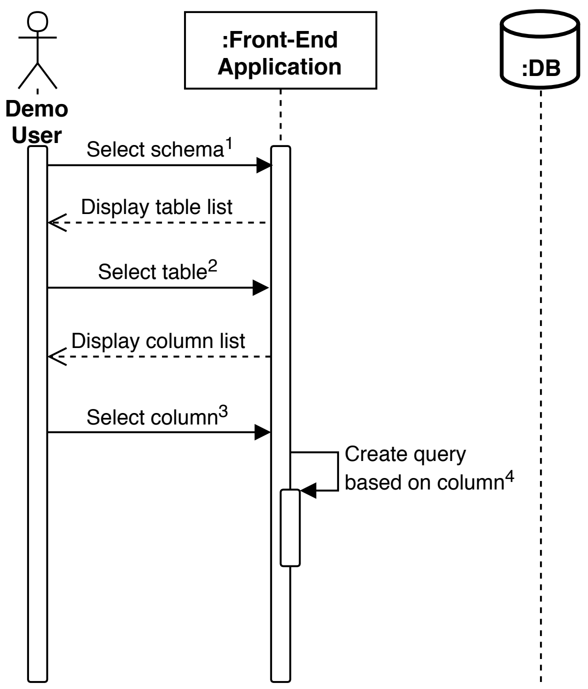


Figure 3.3: A sequence diagram for the use case of browsing a domain-specific data set's schema.

7. The database system executes the SQL query and returns its result to the web application.
8. The web application displays the results graphically.
9. The user browses the results, switching between different pages.
10. *Optionally, the user can filter the results by a keyword specified by them.*
11. *Again optionally, the user can sort the results by any column in ascending or descending alphanumerical order.*

[U2] BROWSE DATA SET SCHEMA

While the web application should provide a list of predefined queries for each schema in the database, the user should also have the possibility to compose new queries on any chosen data set. To this end, the user should be able to obtain information about a data set's structure. Figure 3.3 shows a scenario for this use case.

1. The user selects a data set from a list of schemas. The front-end application responds by displaying a list of tables in the selected data set.
2. The user requires information about the properties of a specific table. The web application displays a list of columns in the table, including their data types.
3. The user selects a column from the list.
4. The web application automatically proposes to the user to send a query to list values of the specified column of the chosen table in order to give them a general idea about the column's contents.

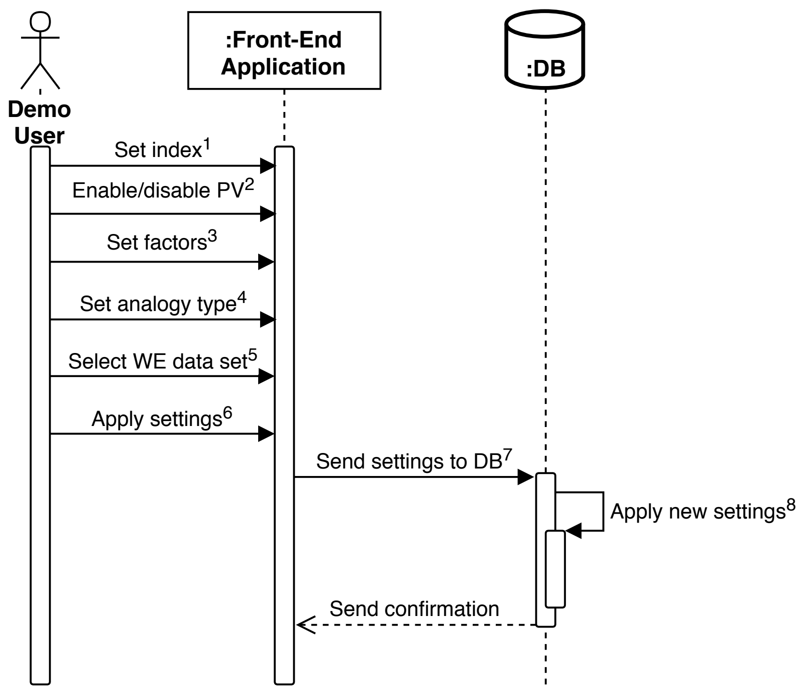


Figure 3.4: A sequence diagram for the use case of configuring the settings of a word-embedding-enabled database.

[U3] CONFIGURE DATABASE SETTINGS

The user should be able to alter the database's word-embedding-specific settings graphically using the front-end application at any time. The settings are then to be applied to all following word embedding operations executed from the front end. Figure 3.4 shows a scenario for this use case.

1. The user sets the index to be used for word embedding operations: no index (*RAW*), PQ or IVFADC.
2. The user enables or disables additional post-verification (if an index was selected in the previous step).
3. The user sets index-dependent factors and the factor for post-verification.
4. The user selects the analogy implementation to be used.
5. The user selects a word embeddings data set to be used.
6. The user requests the application of the selected settings.
7. The front-end application sends the user-defined settings to the database system.
8. The database system applies the selected settings and sends a confirmation to the web application.

[U4] MEASURE & VISUALIZE WORD EMBEDDING OPERATION PERFORMANCE

The web application should provide a way for the user to test FREDDY's word embedding operations' performance using various parameters and visualize the metrics of a performance test in its graphical user interface. Figure 3.5 shows a scenario for this use case.

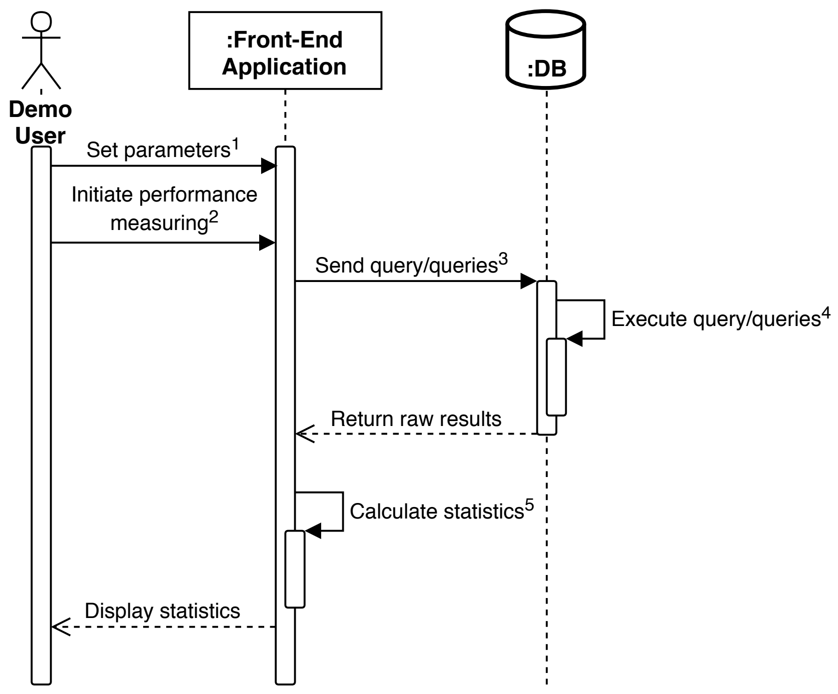


Figure 3.5: A sequence diagram for the use case of measuring a word embedding operation's performance.

1. The user sets the parameters for the performance test.
2. The user requests the initiation of the performance test.
3. The front end requests the execution of the specific queries constituting the performance test.
4. The database executes the requested word embedding operations and returns their respective results and durations to the front end.
5. The web application calculates the statistics of the operation's *precision* and *average execution time*.
6. The front end displays the results of the performance test as a point on a chart with *precision* and *average execution time* as its axes.

With each performance test executed, new points should be added to the chart. They should also be visually distinguishable according to the respective test's parameters and results.

3.2 EXAMPLE SCENARIO

Figure 3.6 presents an example scenario of a complete user interaction with the web application, combining the previously described use cases. First, the user loads the web application using their browser. They select one of the provided domain-specific data sets and an example query for it. Using the built-in text editor, they edit the query according to their preferences. Afterwards, the query is executed and its results are shown to the user as a table. They switch between multiple pages of results, sort them according to a specific attribute and search for a keyword in the results. Then, the user selects another domain-specific data set. The web application shows a list of tables contained in it. The user uses the displayed list to find a specific property of a table they are looking for and sends a query

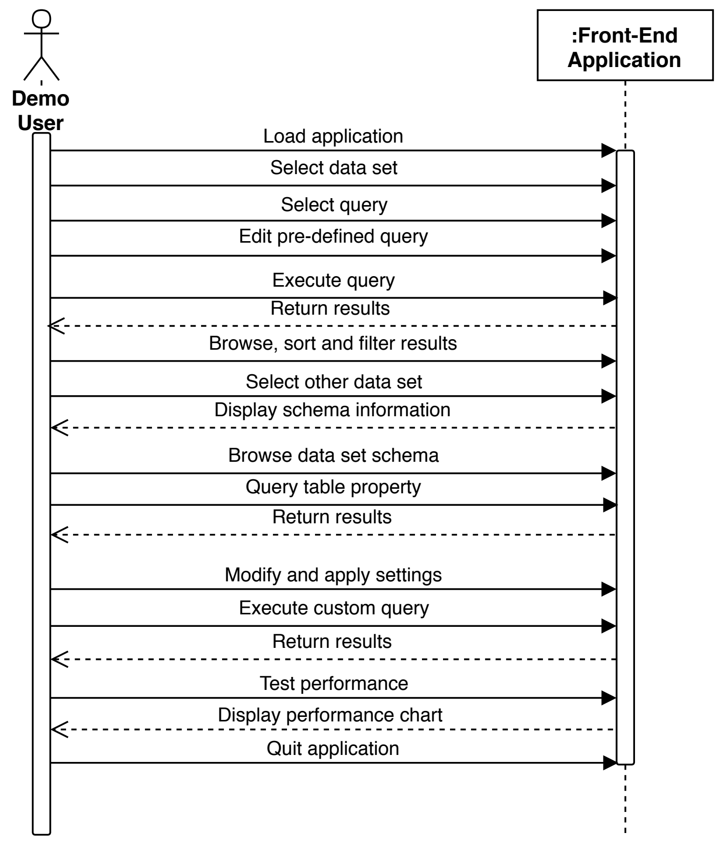


Figure 3.6: A sequence diagram presenting an example scenario for the web application's use.

for it. Then, the user alters the settings for used index, post-verification and used analogy implementation various and switches to another word embeddings data set. They enter a custom query into the text editor, execute it and examine the results. After that, the user decides to test FREDDY's k-NN search implementation's performance. They enter the desired parameters for the k-NN test and execute it. The web application conducts the requested test and displays a chart with its result. The user repeats the performance test with different indexes, post-verification and parameters. After examining the test results, the user decides that their interaction with the application is over and exits the application.

3.3 REQUIREMENTS

The requirements for the web application are divided into *functional* and *non-functional* requirements. The *functional* requirements are derived from the use cases and example scenario described above and define the web application's functionality and behavior, while the *non-functional* requirements elaborate its performance and usability characteristics.

3.3.1 FUNCTIONAL REQUIREMENTS

[R1] REQUIREMENTS FOR USE CASE U1: QUERY DATA SETS AND BROWSE RESULTS

R1.1 The web application should provide an option to switch between different domain-specific data sets stored in the database.

R1.2 A set of example predefined queries containing word embedding operations should be provided by default.

R1.3 The web application should provide a built-in, syntax-highlighted query editor for query customization.

R1.4 Initiating a query execution in the web application should result in the word-embedding enabled database executing the query and returning the query results to the front end.

R1.5 Query results should be shown in the front end as a table, with options for the user to sort and filter results. Furthermore, for the sake of better usability, the results should be split into multiple pages.

[R2] REQUIREMENTS FOR USE CASE U2: BROWSE DATA SET SCHEMA

R2.1 After selecting a domain-specific data set, a list of tables contained within it should be displayed.

R2.2 The columns of each table and their data types should be shown explicitly.

R2.3 An option to display several values from a given column should be provided for a better overview over a table's contents by auto-filling the query editor with the respective projection query.

[R3] REQUIREMENTS FOR USE CASE U3: CONFIGURE DATABASE SETTINGS

The web application should provide options to change the following FREDDY settings:

R3.1 index used;

R3.2 post-verification;

R3.3 index and post-verification factors;

R3.4 analogy function implementation;

R3.5 word embeddings data set used.

R3.6 Furthermore, settings should take effect immediately and for all following operations after their application.

[R4] REQUIREMENTS FOR USE CASE U4: MEASURE & VISUALIZE WORD EMBEDDING OPERATION PERFORMANCE

R4.1 There should be an option for the user to adjust the parameters for a performance test.

R4.2 The web application should calculate the measures of *precision* and *average execution time*.

R4.3 A performance test's results should be presented as a data point on a chart with the aforementioned measures as its axis.

R4.4 There should be a visual distinction between different tests' data points based on their parameters, index and post-verification used.

R4.5 Executing further performance tests should not result in the loss of previous data points.

3.3.2 NON-FUNCTIONAL REQUIREMENTS

N1 The application's graphical user interface should be designed according to modern design principles and ensure good usability and short response times.

N2 The web application's architecture should be split into multiple components in order to separate the presentation layer, application layer and the data access layer.

N3 The web application should not cause a large overhead for query execution: a query's execution time should depend chiefly on the database system's performance.

N4 The web application's front end should be compatible with modern web browsers.

EVALUATING NON-FUNCTIONAL REQUIREMENTS

While the functional requirements can be evaluated by simply assessing the features of the completed web application, evaluating the non-functional requirements warrants additional examination. N1 is evaluated according to user experience design principles derived from psychological research [20]. N2 is taken into consideration when choosing the technologies and architectural patterns used in the project's implementation. N3 is evaluated by conducting performance measurements on the database system and the web application. N4 is evaluated by testing the web application using two state-of-the-art web browsers. For further information about methods used in the evaluation of non-functional requirements and their results, see Section 5.3.

4 IMPLEMENTATION

In this chapter, I describe the design and implementation of the developed web front end. First, I introduce the application's high-level architectural components, explain the interactions between them and give an overview of their deployment specifics. Then, I list the different word embedding and domain-specific data sets used in the web application's database. Most importantly, I describe the implementation of the web application's front and back end, including the reasons for choosing the specific technologies used.

The web application's source code was licensed under the terms of the free software MIT license and can be found in a GitLab repository[1].

4.1 ARCHITECTURAL OVERVIEW

4.1.1 HIGH-LEVEL DESIGN

The software solution's architecture is based on a classic *three-tier* web application architecture. A three-tier architecture is a *client-server* software architecture pattern. It separates the user interface of the application (the *presentation tier*), the functional logic (the *application tier*) and the data storage and access (the *data tier*). Not only does it provide the advantages of modularity, speed of development, scalability and performance, but it also al-

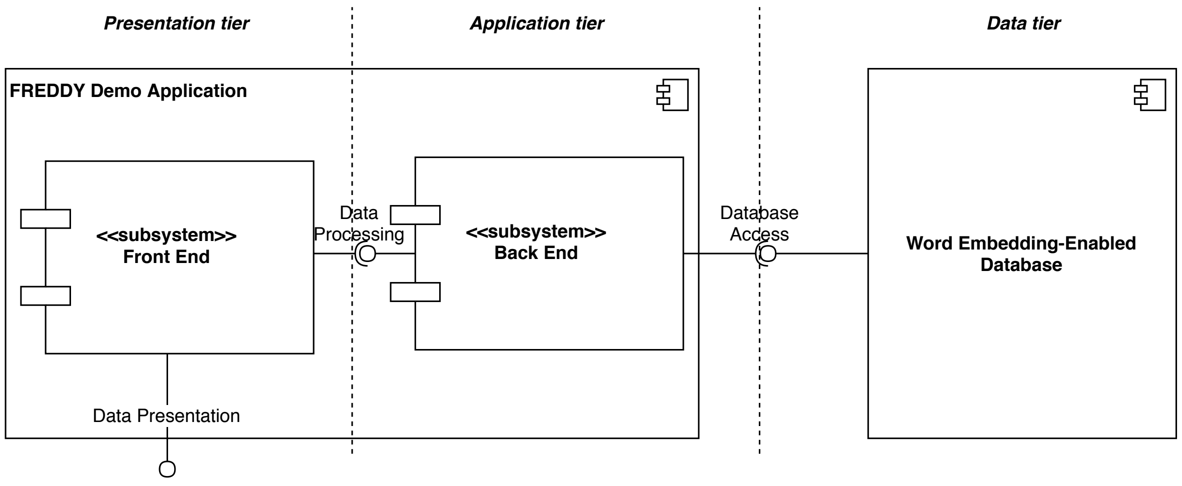


Figure 4.1: A component diagram showing the web application's three-tier architecture.

[1]https://gitlab.com/yanakiev/freddyDemo

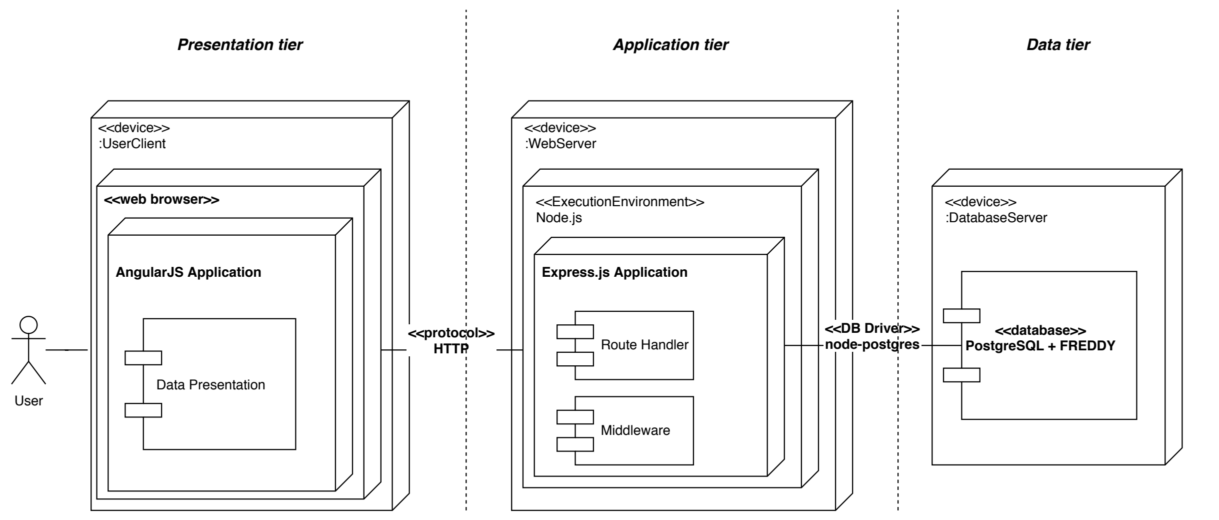


Figure 4.2: A diagram detailing the web application's deployment.

lows the independent replacement of any of its components. For example, one can use any user interface without modifying the data processing component behind it. Furthermore, many well-established modern web development stacks are based on this model. These advantages make the three-tier architecture especially well-suited to the demo application's use cases. Figure 4.1 shows the different components that constitute the application's architecture. A detailed description of technologies and languages used in the application's components follows in the next sections.

The data tier comprises both the data sets used in the web application and the database management software that manages and provides access to the data. It consists of the word-embedding-enabled relational database system and is the data source for the web application. It provides access to word embedding operations and stored data sets to the next tier in the architecture, the application tier.

The application tier, also called *back end*, provides a middleware between the data tier and the presentation tier and drives the application's core capabilities. It extracts information from the database at the front end's request, processes it, dynamically generates data based on it and forwards it to the presentation tier. The back end can also relay other types of commands to the database, such as adjusting the database's settings. It also provides the functionalities of a classic web server, serving the web application's static content.

The web application's user directly accesses the presentation tier, also called *front end*. The user can achieve the different use cases by interacting with its graphical user interface. The front end of the web application requests dynamic and static data from the back end and displays content and information useful to the end user, such as the results of a word embedding SQL query.

4.1.2 DEPLOYMENT MODEL

Figure 4.2 shows the demo application's distributed deployment model[2]. The user accesses the web application's front end using a conventional web browser running on their own device. The front end was realized as an *AngularJS* application and provides a way for the user to retrieve query results, examine them, adjust word embedding settings and measure the database's performance using graphical interaction paradigms. In other words, it is responsible for the presentation of data obtained from the back end in a user-friendly manner. The front end and the back end, which runs on a web server device, communicate over HTTP

[2]The web application's components may also run on the same physical machine.

using asynchronous RESTful API calls. The back end was implemented as an *Express.js* application written in JavaScript and running in the *Node.js* run-time environment. It consists of a route handler which redirects requests from the front end to the relevant API functions and other middleware. The middleware itself has facilities for both handling a conventional web server's tasks and API functions specially implemented for accessing FREDDY's capabilities. The back end communicates with the database using a PostgreSQL driver for Node.js. The database system, consisting of a PostgreSQL system with word embeddings data sets and installed FREDDY extensions, runs on another device called a database server.

4.2 DATABASE

The relational database management system powering the web application is a standard PostgreSQL 10.4 installation running on an Ubuntu 16.04 virtual machine. In addition, the word embedding extensions provided by FREDDY have been installed into the database system, giving access to all word embedding operations implemented by it. Several word embeddings and domain-specific data sets have also been imported into the database in order to use word embedding queries in the demo application.

The web application provides examples of ten predefined queries using word embeddings on the domain-specific data sets. The predefined queries are stored in a JSON file using a custom format on the web application's back end. Adding new predefined queries is as simple as appending a new entry to the JSON file (see Table A.1 in Appendix A.1 for its technical description).

4.2.1 WORD EMBEDDINGS DATA SETS

The database contains three different pre-trained sets of word embeddings, each of them including the index tables required by FREDDY's high-performance word embedding operations.

The first set of word embeddings was trained on part of the data set of *Google News* using the word2vec algorithm and contains 300-dimensional vectors for 3 million words and phrases. Considering the broad range of topics covered in the Google News corpus, this data set is suitable for extracting semantic information in different areas, such as entertainment, current events or science. Its downside is that it was trained on news from only the last ca. 10 years, and thus covers only recent events and topics. The second word embeddings data set was trained on data from *Wikipedia*, and the last one contains vector representations trained using the *GloVe*[3] unsupervised learning algorithm. The index structures for the three word embeddings data sets were created using the tools bundled with FREDDY.

4.2.2 DOMAIN-SPECIFIC DATA SETS

Three domain-specific data sets from various areas of human knowledge were imported into the database in order to give different users an idea about the capabilities of word embeddings and their uses in relational databases.

IMDb[4] is an online database of information about films and television programs, including cast, production crew and personnel biographies and plot summaries. It contains more than 250 million data items including more than 4 million movies or other programs and 8 million cast and crew members [21]. As movies and TV are an integral part of pop culture, this data set was chosen to demonstrate FREDDY's capabilities in a way that is easily understandable

[3]`https://nlp.stanford.edu/projects/glove/`

[4]`https://www.imdb.com/`

for almost everyone. The IMDb data set was imported using the *imdbpy2sql*[5] command-line tool by Davide Alberani.

Discogs[6] is a crowdsourced database of information about audio recordings, including commercial releases, promotional releases, and bootleg or off-label releases. More than 415,000 people have contributed their knowledge to it, to build up a catalog of more than 9,900,000 recordings and 5,700,000 artists [22]. Users of the web application can extract semantic information about their favorite bands or artists and their body of work by querying the Discogs database. The Discogs data set was imported using the *discogs2pg*[7] tool written by Ezequiel A. Alvarez.

The last data set used in the demo application's setup is *dblp*[8]. It is the online reference for bibliographic information on major computer science publications and indexes over 3.3 million publications, published by more than 1.7 million authors. To this end, dblp indexes about than 32,000 journal volumes, more than 31,000 conference or workshop proceedings, and more than 23,000 monographs [23]. By querying the dblp data set, the user can extract information about the relationships between the authors of publications and other semantic data.

4.3 BACK END

The web application's back end was developed using the *Node.js* web framework *Express.js* and was written in ECMAScript 6-compatible JavaScript. Node.js is a server-side runtime environment that lets developers use JavaScript for server-side scripting. It has an event-driven architecture capable of asynchronous I/O. Furthermore, it has the largest ecosystem of open source libraries in the world, which include database drivers and components for front and back end development. This facilitates the task of quick development of web applications by providing ready solutions to frequently encountered problems in web development. Express.js is a web application framework which functions as a thin layer of fundamental web application features on top of Node.js, thereby allowing the easy development of a web application's robust API. Another reason why Node.js and Express.js were chosen for this project is their frequent use together with AngularJS in a web development stack and the resulting wide availability of resources on their combined usage.

4.3.1 DESIGN OVERVIEW

The components of the application's back end, its relation to Node.js and Express.js and its top-level external library dependencies are shown in Figure 4.3. Both the back end itself and the external libraries required by it run inside the Node.js runtime environment.

Two of the back end's components were implemented using the interfaces provided by Express.js: the *Application Programming Interface* (API) and the *Routing Handler*. The API contains the implementations of the functions called by the front end using standard HTTP GET and POST requests. Each of these functions parses parameters contained in a client's request, processes them, creates one or more SQL queries for FREDDY from them, relays the queries to the database using a database driver and produces an HTTP response from the result. The *routing handler* parses the URL of a client's request. If the client requests an API URL, it dispatches the request to the respective API function in order to return dynamic content generated by it. It is also used to determine what static content should be returned as a response to a client's request, such as a web page, a CSS stylesheet or JavaScript code.

[5]https://github.com/alberanid/imdbpy
[6]https://www.discogs.com
[7]https://github.com/alvare/discogs2pg
[8]https://dblp.uni-trier.de/

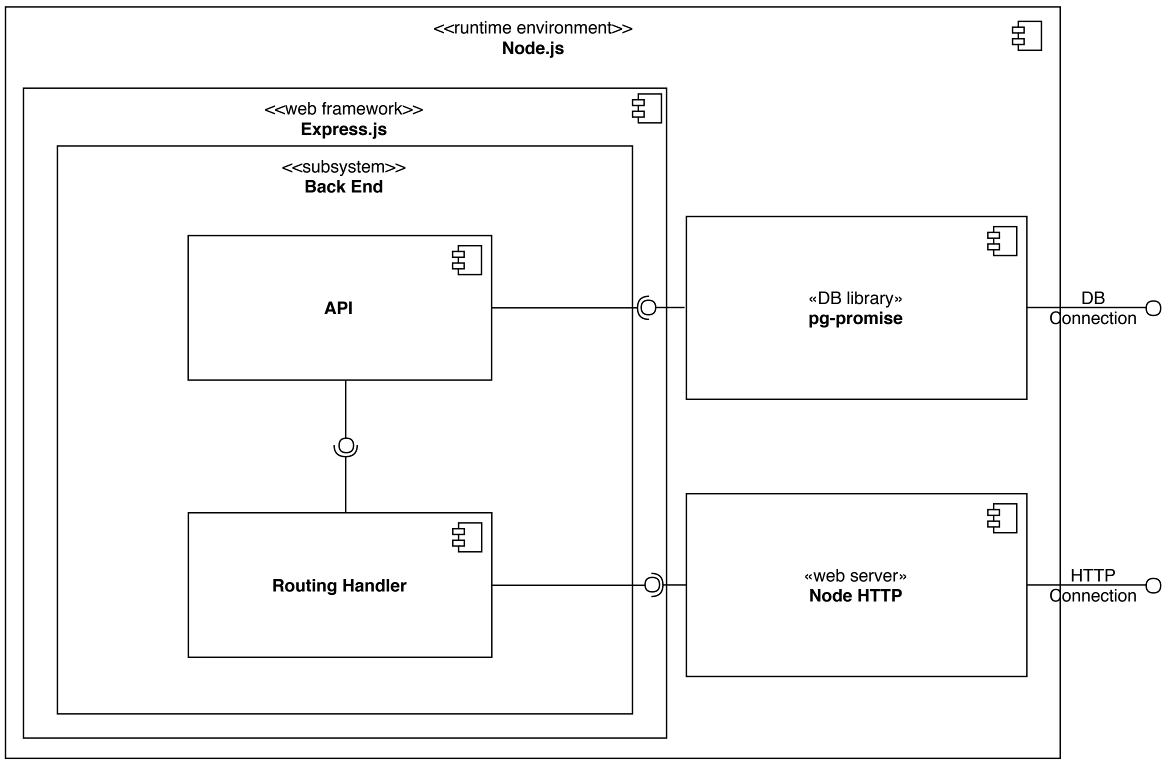


Figure 4.3: A component diagram showing the components of the web application's back end.

The web application's back end delegates communication with the word-embedding enabled database system to *pg-promise*, a database library built on top of *node-postgres*, the Node.js database driver for PostgreSQL. The library was chosen over the lower-level database driver for its features of automatic connection management and pooling, automatic transactions, query formatting and asynchronous operations, of which the API makes use. Communicating over HTTP is also handled by an external Node.js library: *Node HTTP*. It is one of the core modules in Node.js and functions as a web server, listening for client requests and responding to them. The routing handler registers with the web server at the application's initialization to redirect client requests appropriately.

4.3.2 API DESCRIPTION

The web application's API conforms to Representational State Transfer (REST) principles and makes use of HTTP GET and POST requests, as well as JSON-formatted responses, to exchange data between the front end and the back end. Twelve API functions have been implemented. The first seven of them only wrap FREDDY's built-in word embedding operations. The remaining five have been implemented specifically for the web application's use cases. These include a function fetching information about a schema in the database, a function passing a custom query to the database, a function applying user-defined settings to the database, a special k-NN performance test function and a command for the database to pre-load required index tables into the server's RAM. All of the API functions except for the settings function are called by the client via a parameterized HTTP GET request and return a response containing the function's results in a JSON format. See Table A.3 in Appendix A.2 for a technical description of the back end's API.

The performance test function uses a list containing one thousand predefined k-NN query terms and their raw results to send a number of k-NN queries for random terms to the

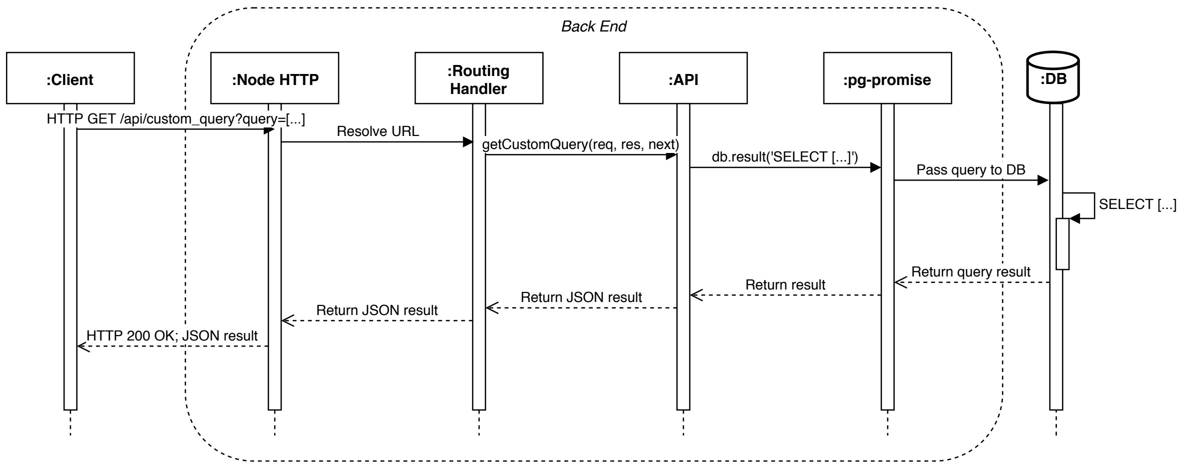


Figure 4.4: A sequence diagram showing an example interaction with the web application's back end.

database and calculate the average precision and execution time of the operations, comparing to the predefined raw results.

The settings function can be invoked via a POST request carrying a settings object in a custom format (see Table A.2 in Appendix A.2 for the format's description).

4.3.3 EXAMPLE API CALL

Figure 4.4 shows the interactions between the different back end components, the client and the database when an API function is invoked. A client requests the execution of a user-defined query by sending an HTTP GET request to the listening web server. The Node HTTP instance uses the routing handler to resolve the requested URL, which invokes the respective API function implementation. It builds an SQL query from the client's request and calls a method of pg-promise's API in order to forward the query to the database. The database processes the query and returns the result to the back end, which is converted into a JSON response by the back end's API. The JSON response is then forwarded via the routing handler to the web server, which responds with a 200 OK HTTP message containing the query's results in its body. The JSON data can then be used to present the query's results in the front end's graphical user interface.

4.4 FRONT END

4.4.1 LANGUAGES USED

The web application's front end was implemented in the standard languages used for web development. HTML5 was used for web page markup, CSS3 for describing the content's presentation and JavaScript for the application's front-end logic and DOM manipulation. Furthermore, the web application makes use of several web development frameworks and their respective libraries, as well as external components.

4.4.2 IMPLEMENTATION

Figure 4.5 shows an overview of the front end's components and their dependencies. The web application's front end was implemented as a single-page application using the open

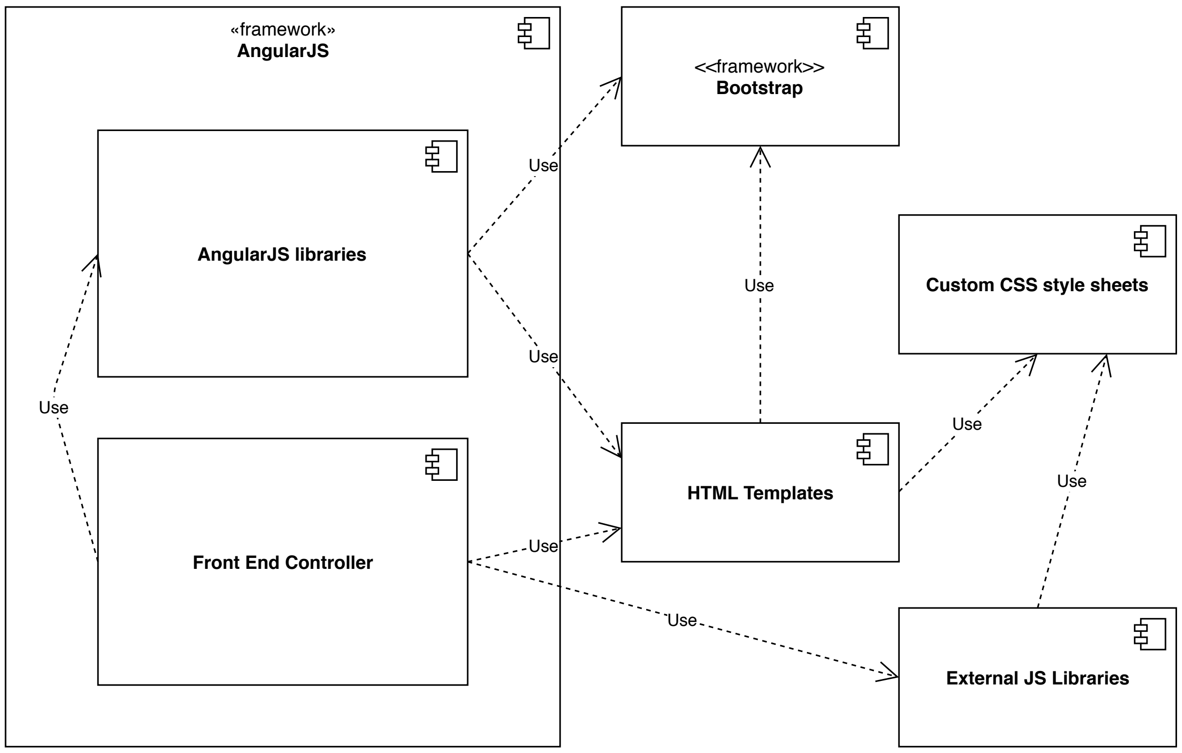


Figure 4.5: A component diagram showing the components of the web application's front end.

source web framework *AngularJS*. AngularJS greatly simplifies web application development by presenting a high level of abstraction to the developer. It supports the Model-View-Controller (MVC) paradigm and provides data binding and dependency injection, which is why it was chosen for this project over using pure JavaScript with HTML. AngularJS is a completely client-side framework and runs in the user's browser, thus separating the server side from the client side. Furthermore, it is completely modular and provides many libraries for common tasks, including libraries used for graphical user interface design or HTTP communication. The framework guides the developer through the whole process of building a web application, including designing the user interface and writing the business logic [24].

AngularJS functions by extending standard HTML by its own directives, attaching specified behavior to HTML elements. Templates written in HTML using AngularJS directives are transformed by the framework into a *view* in the MVC model. The business logic behind views is contained in a *controller*, and a *scope* accessible by both controllers and directives is used to store the data model [24]. In this web application, a controller is used for all client-side logic: sending requests to the back end and processing its responses, presenting data, manipulating the user interface and storing relevant data in the model. Additionally, the controller makes use of several AngularJS libraries to achieve its tasks:

- *angular-ui-bootstrap*, *angular-loading-bar* and *angularjs-slider* for AngularJS directives for commonly used UI elements;
- *ngTable* for presenting a query's results in a sortable and filterable table;
- *ngAnimate* for UI animations;
- *ngRoute* for web page routing;
- furthermore, the *$http* AngularJS service is used for communicating with the application's back end.

The web application's controller can be split logically roughly into five different parts, some of them also using external JavaScript libraries:

1. *query list and query editor* component: fetches a list of predefined queries from the back end, stores them in the model and provides a syntax-highlighted query editor. For the query editor, the JavaScript component *CodeMirror* was used.
2. *schema information* component: fetches information about a database schema's structure from the back end and stores it in the model.
3. *settings* component: controls the presentation of a settings widget and sends the user-specified settings to the back end.
4. *query results* component: controls the presentation of query results and provides the features of sorting and filtering them.
5. *performance view* component: initiates new performance tests by sending a request to the back end, controls the performance chart's presentation and adds new data points to it. For the performance chart, the JavaScript graphing library *plotly.js* was used.

The front end's interface was developed using the open source *Bootstrap* framework. It contains HTML- and CSS-based design templates for typography, forms, buttons, navigation and other interface components. Reasons for choosing Boostrap include its ease of use and its good integration with other web frameworks, including AngularJS. In this project, *angular-ui-bootstrap* was used for native AngularJS directives based on Bootstrap's markup and CSS, providing ready implementations of commonly used user interface elements. Bootstrap also allows the rapid development of responsive, mobile-friendly web applications. It is valued for its high customizability and is compatible with the custom CSS style sheets used to define the web application's appearance.

5 EVALUATION

In this chapter, I evaluate the web application's fulfillment of the use cases, functional and non-functional requirements specified in Chapter 3. First, I present the different components of the application's graphical user interface. Then, I analyze which parts of the application fulfill which functional requirements. Afterwards, I evaluate the application's user interface according to modern UX design principles. To conclude this chapter, I also provide a short analysis of other non-functional requirements.

5.1 INTERFACE WALKTHROUGH

5.1.1 START PAGE

When a user first loads the web application using their web browser, they are greeted by its *start page* (see Figure 5.1). Most of the start page is blank, as results of queries or performance tests are shown in its main part. At the top of it is a navigation bar **(1)** with the application's name and hyperlinks to the project's abstract and the demo application itself. By default, the demo application is loaded. By clicking on the *Settings* button **(2)**, the user can show or hide an advanced settings menu. There are two drop-down menus **(3)** for selecting respectively a database schema to be queried and a word embeddings data set. One can also switch between a database querying view and a word embedding operation performance view by clicking on the respective tab **(4)**. There is another drop-down menu for selecting a word-embedding-enabled SQL query from a list of predefined queries **(5)**. On the left of the page, a list of tables in the currently selected schema is displayed **(6)**.

5.1.2 SETTINGS

ADVANCED SETTINGS MENU

When a user clicks the Settings button, a panel containing widgets for configuring a number of *advanced settings* for FREDDY appears on the right of the page (see Figure 5.2a). There is an information tooltip providing the user with a short description of the respective setting for each of the options. The user can choose between raw word embedding operation results or enable the usage of an index **(1)**. If the user has selected the *IVFADC* index, they can also set the number of nearest coarse quantization clusters considered for a word embedding operation **(2)**. A drop-down menu **(3)** allows the selection of one of three analogy function implementations. The user can also enable or disable additional result post-verification **(4)** and set the number of results to be considered for it by setting the PV factor **(5)**. The number of results considered for post-verification is equal to the PV factor multiplied by *k* in a k-NN

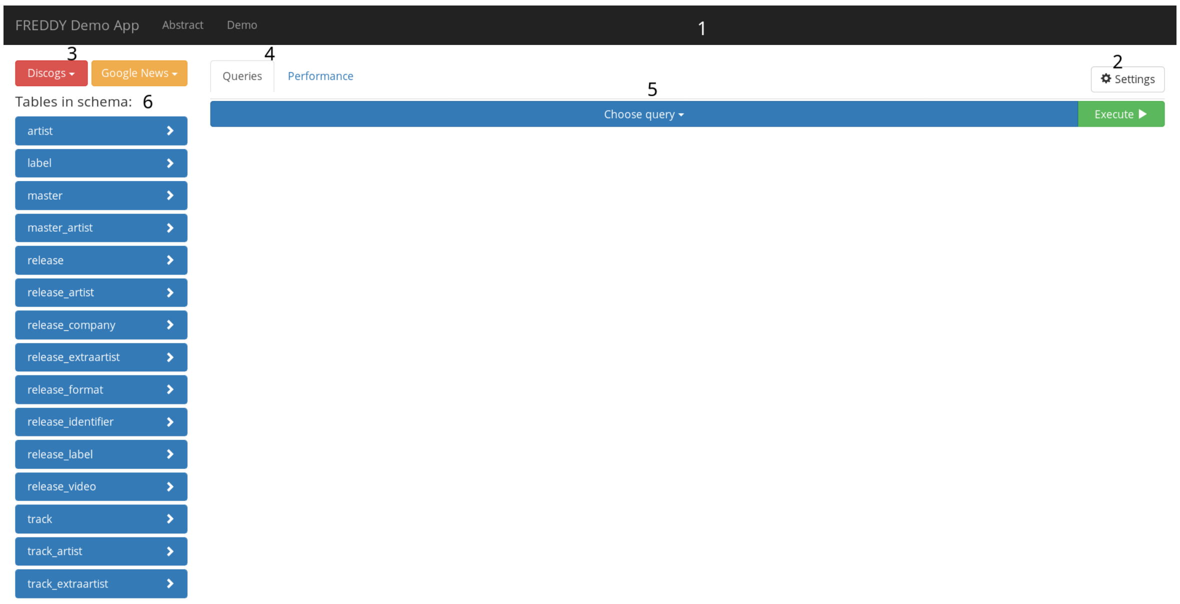
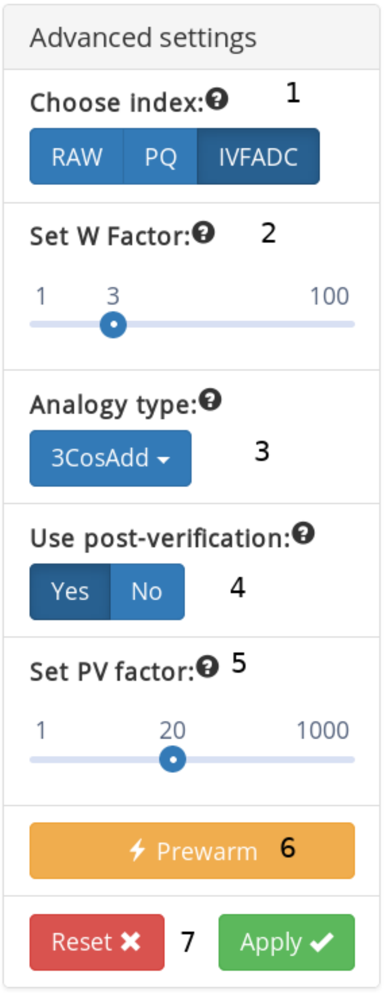
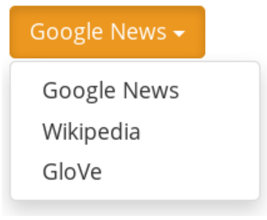


Figure 5.1: A screenshot of the web application's start page.



(a) A screenshot of the advanced settings menu.



(b) A screenshot of the word embeddings picker.

Figure 5.2: Screenshots of the front end's settings widgets.

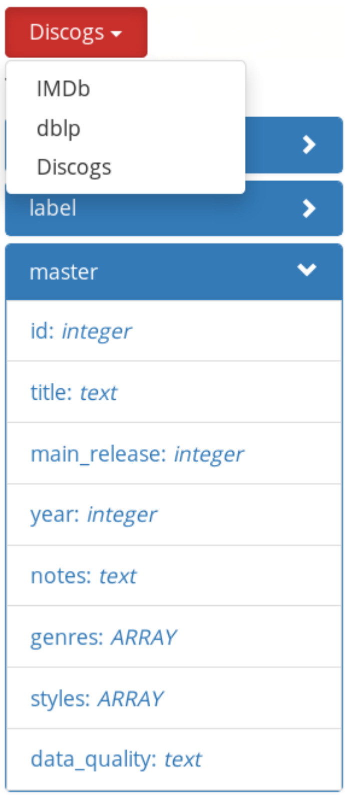


Figure 5.3: A screenshot of the schema picker and browser.

search. By clicking the prewarm button **(6)**, the user can improve the application's performance immediately after its start by sending a request to FREDDY to load the index tables into the database server's main memory. A pair of buttons **(7)** allows resetting FREDDY's settings to their defaults and applying the selected settings to the database, respectively. After the settings are applied to the database, they affect all following word embedding operations.

WORD EMBEDDINGS PICKER

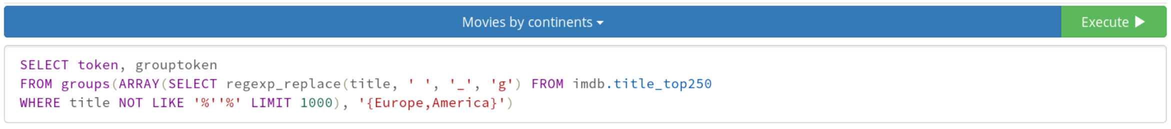
Using the *word embeddings picker*'s drop-down menu in the top left corner of the page (shown in Figure 5.2b), the user can select one of three word embeddings data sets to be used in word embedding operations. This affects all following operations, too.

SCHEMA PICKER AND BROWSER

The leftmost part of the page contains the *schema picker and browser* (shown in Figure 5.3). When a domain-specific data set is selected in the drop-down menu, its schema's tables are displayed as a list below and the list of predefined queries for the specified data set is loaded. Furthermore, when a user clicks on a table's name, a list of columns contained in the table and their datatypes is shown. A user can also conveniently query the data set for the first one thousand values of a specific column by clicking on its name.

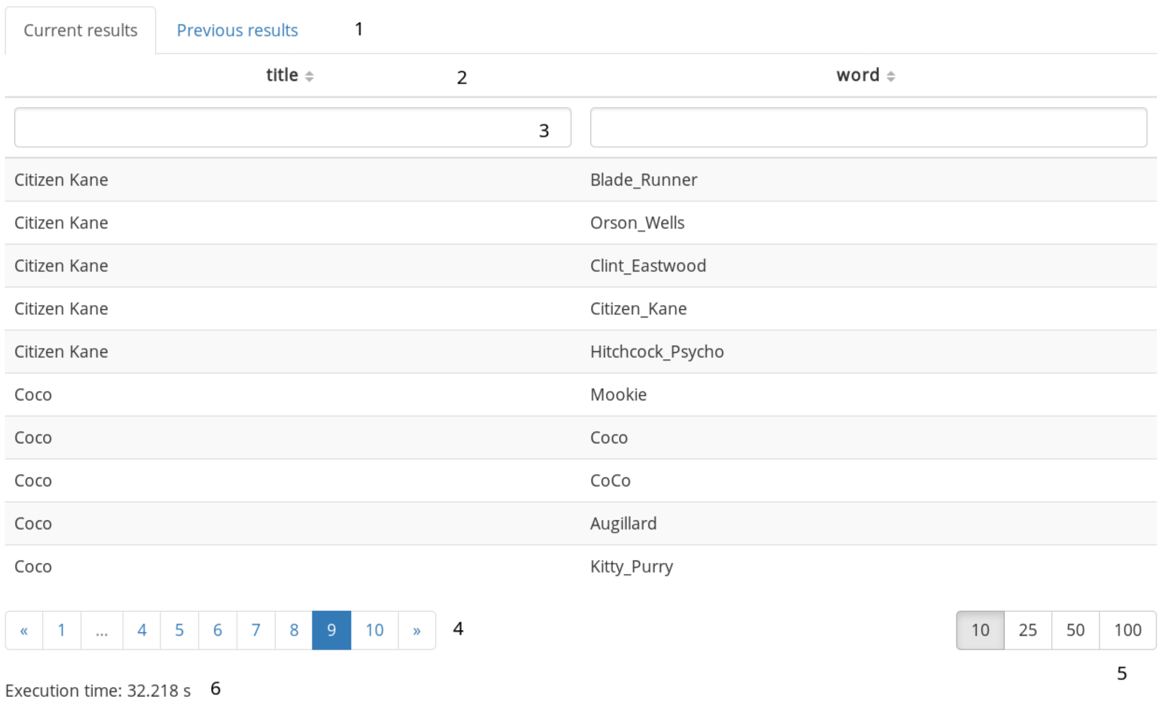
5.1.3 QUERY VIEW

The *query view* is the default view of the web application. It allows the user to send predefined word-embedding-enabled SQL queries to the database, edit them to their preferences and examine their results.



```
SELECT token, grouptoken
FROM groups(ARRAY(SELECT regexp_replace(title, ' ', '_', 'g') FROM imdb.title_top250
WHERE title NOT LIKE '%''%' LIMIT 1000), '{Europe,America}')
```

Figure 5.4: A screenshot of the query picker and editor.



title	word
Citizen Kane	Blade_Runner
Citizen Kane	Orson_Wells
Citizen Kane	Clint_Eastwood
Citizen Kane	Citizen_Kane
Citizen Kane	Hitchcock_Psycho
Coco	Mookie
Coco	Coco
Coco	CoCo
Coco	Augillard
Coco	Kitty_Purry

Execution time: 32.218 s

Figure 5.5: A screenshot of the query results table.

QUERY PICKER AND EDITOR

The query picker and editor are located at the top of the query view (see Figure 5.4). The application's user can select a word-embedding-enabled query by its description from a list of predefined queries in the *query picker*'s drop-down menu. After selecting a query, a syntax-highlighted SQL *query editor* appears under the query picker. Here, the user can customize the query to their own liking. After a user is done with editing the query, they can send the query for execution to the database by clicking on the *Execute* button.

QUERY RESULTS TABLE

The results of a query are displayed in the *query results table* (shown in Figure 5.5). The user can switch between the results of the previous query and the current one by clicking on the respective tab **(1)** and sort the result entries in ascending or descending alphanumerical order by clicking on a column's name **(2)**. Furthermore, options are provided to filter the entries according to a keyword specified by the user **(3)**. Query results are presented in a number of pages through which the user can browse **(4)** and they can also adjust the number of entries displayed in a page **(5)**. Lastly, the query's execution time is shown below the results table **(6)**.

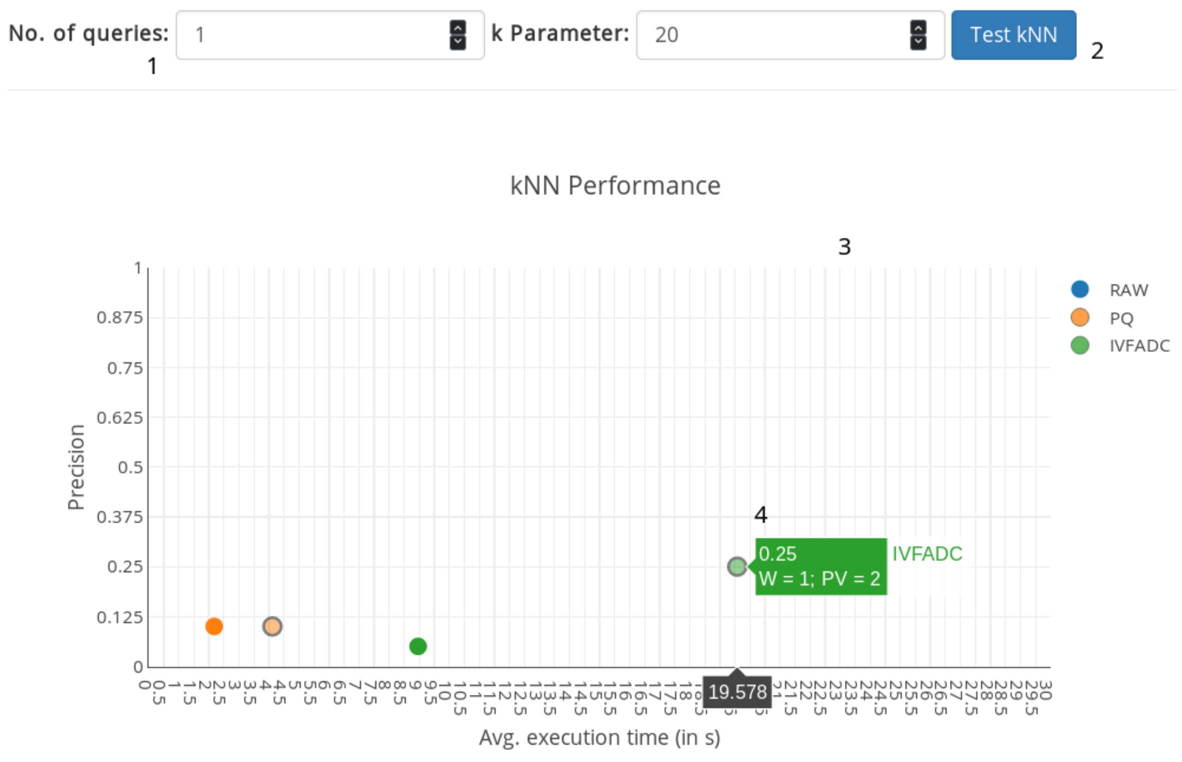


Figure 5.6: A screenshot of the performance view.

5.1.4 PERFORMANCE VIEW

In the performance view (see Figure 5.6), the user can measure the precision and performance of k-NN operations with different parameters. The user can set the parameters for the word embedding operation using the advanced settings menu and can choose the number of operations to be executed and over which the average of the respective metric should be calculated, as well as the parameter *k*, using the provided number boxes **(1)**. By clicking on the *"Test kNN"* button **(2)**, the user can initiate the execution of a performance test by the back end. After its completion, a new data point is added to the performance chart **(3)** which shows the precision (compared to the results of a non-indexed, raw operation) and the average execution time of the operations constituting the performance test. Data points of tests performed using different indexes are shown in a different color and data points of tests using post-verification are displayed semitransparent. Furthermore, when a user place their cursor over a data point, the values of PV and W factors, if they were used in the selected test, are displayed **(4)**.

5.2 FUNCTIONAL REQUIREMENTS

All use cases were tested during the development of the web front end and after its completion. The application fulfills all functional requirements and hence is well-suited for the use cases described in Section 3.1. Table 5.1 lists which application components and front end interface elements are involved in the fulfillment of a particular functional requirement.

Use Case	Requirement	Involved Components	Interface Part
U1	R1.1	Front End, Back End, Database	Settings: Schema Picker
	R1.2	Front End, Back End, Queries File	Query View: Query Picker
	R1.3	Front End	Query View: Query Editor
	R1.4	Front End, Back End, Database	Query View
	R1.5	Front End	Query View: Query Results Table
U2	R2.1	Front End, Back End, Database	Schema Browser
	R2.2	Front End, Back End, Database	Schema Browser
	R2.3	Front End	Schema Browser, Query Editor
U3	R3.1	Front End, Back End, Database	Settings: Advanced Settings Menu
	R3.2	"	"
	R3.3	"	"
	R3.4	"	"
	R3.5	"	Settings: Word Embeddings Picker
	R3.6	Database	n/a
U4	R4.1	Front End, Back End, Database	Performance View
	R4.2	Front End, Back End	n/a
	R4.3	Front End	Performance View
	R4.4	"	"
	R4.5	"	"

Table 5.1: A table containing an evaluation of the functional requirements for the web application.

5.3 NON-FUNCTIONAL REQUIREMENTS

5.3.1 USER INTERFACE DESIGN [N1]

In this section, I evaluate the web application's interface according to the *Laws of UX* [20]. Laws of UX is a collection of the maxims and principles that designers can consider when building user interfaces. This list of well-established design laws rooted in psychology was compiled by front-end designer Jon Yablonski in 2018. Currently, seventeen laws are included in the list. I analyze the graphical user interface according to them and comment on the design decisions that were made. Here, I focus only on six of the laws relevant to the web front end's graphical interface. These are the Doherty Threshold [25], Hick's Law [26] [27], Jakob's Law [28], the Laws of Common Region and of Proximity [29] and the Zeigarnik Effect [30]. The short descriptions below quote the Laws of UX website [20].

DOHERTY THRESHOLD

> *"Productivity soars when a computer and its users interact at a pace (<400ms) that ensures that neither has to wait on the other."*

The web application's interface by itself is fast and responsive, owing much to the high-performance web frameworks and technologies used. However, its quickness is also heavily dependent on the execution time of the word embedding operations used by it and, consequently, on the performance of the machine on which the database server is deployed. Thus, a user may have to wait up to four minutes for the results of a complex word embedding operation.

HICK'S LAW

> *"The time it takes to make a decision increases with the number and complexity of choices."*

The interface elements' design takes Hick's law into consideration. For instance, only queries and tables of the currently selected schema are shown in the query picker, respectively schema browser. Settings not relevant to the currently selected index and other selected settings are hidden from the user to simplify their choices.

JAKOB'S LAW

> *"Users spend most of their time on other sites. This means that users prefer your site to work the same way as all the other sites they already know."*

For this reason, standard user interface design frameworks and components were used. The Bootstrap framework is used by 17.4 percent of all websites as of June 2018 [31], thus its interface components (buttons, radio buttons, text boxes, etc.) should be well-known to most Internet users. Furthermore, other elements of the front end's interface such as sliders for setting a numerical variable's value have been used in graphical user interfaces for decades. I also consulted with several acquaintances who have no prior experience in user interface design to obtain an unbiased opinion about the placement of the *"Apply"* and *"Reset"* buttons in the advanced settings menu in order to determine which option is the most intuitive.

LAW OF COMMON REGION AND LAW OF PROXIMITY

> *"Elements tend to be perceived into groups if they are sharing an area with a clearly defined boundary."*
>
> *"Objects that are near, or proximate to each other, tend to be grouped together."*

The application of these two laws is evident in the grouping of UI elements according to their common task. For example, the *"Execute"* button is grouped together with the query picker and query editor. The schema picker and schema browser occupy the same area in the interface's left side. The advanced settings menu is clearly defined as a panel, containing all advanced settings in one place. However, it was decided to move the word embeddings data set picker out of the advanced settings menu to the opposite part of the interface in order to expose more of FREDDY's functionality to the user. In this case, functionality was chosen over strict adherence to user interface design principles.

ZEIGARNIK EFFECT

> *"People remember uncompleted or interrupted tasks better than completed tasks."*

During the application's testing, it became obvious that the lack of visual feedback during a query's execution or after a failed execution leads to the user's confusion about their action's outcome. Therefore, a loading bar showing the progress of the current request was added to the top of the interface. Furthermore, an error message with details about a request's failure appears under the query picker in the event that a query's execution fails. These two additions help the user to track their progress in achieving the respective use case.

5.3.2 SEPARATION OF CONCERNS [N2]

This requirement is fulfilled by the web application's architectural separation in front end, back end and database system, as described in Section 4.1.

5.3.3 QUERY EXECUTION PERFORMANCE [N3]

The performance of the web application's functionalities was measured in two batches of ten consecutive performance tests. In the first one, a standard grouping query was executed using the web application's query view. The query's execution time and the time from the user initiating the query's execution until the loading of the results in the graphical user interface were measured and compared. On average, the web application causes an overhead of 47.9 ms (median 41 ms) over the database-only execution time of a grouping query.
The other ten performance tests concern the web application's performance view. Again, the database execution time and the time between the initiation of the k-NN performance test and the update of the performance chart were measured. In each test, ten 5-NN queries were executed at a time. Because of the greater complexity of this scenario, a performance test took 302.4 ms longer on average (median 272 ms) than only the execution of its constituent queries. Both overhead values remain under the Doherty threshold of 400 ms and can be considered negligible for the end user. Furthermore, they do not affect the entire system's performance in a negative way.

5.3.4 WEB BROWSER COMPATIBILITY [N4]

The web application's front end was tested using the state-of-the-art (as of June 2018) web browsers Mozilla Firefox (version 60.0.2) and Google Chrome (version 67.0.3396) under the

GNU/Linux operating system. No major differences in none of the web application's functionalities, performance or appearance were found.

6 CONCLUSION AND OUTLOOK

6.1 CONCLUSION

This chapter concludes this bachelor thesis. The thesis detailed the development of a dedicated graphical web front end for a word-embedding-enabled relational database system. The lack of a suitable graphical front end for exploratory use cases featuring a word-embedding-enabled relational database system was identified as an issue. The fundamentals of word embedding techniques, their real-world applications and previous work on integrating their features into a relational database management system were examined. An extensive analysis of the problem in hand followed, from which the requirements for a web front end were derived. Based on the requirements analysis, a three-tier application architecture was chosen for the web application. The front end was then implemented using state-of-the-art web technologies and frameworks and thoroughly tested. Finally, an evaluation of the finished software product showed its suitability for the intended use cases, its conformance with modern user interface design principles and its adequate performance.

Several conclusions could be drawn from this thesis:

- Using a three-tier architecture consisting of several components communicating with each other via messages makes sense for a database system front end and provides high flexibility and modularity.

- When designing a graphical user interface for a web application, trade-offs between the interface's usability and its exposing of the application's functionality should sometimes be made. Even so, UX design principles should always be taken into consideration in the process of creating a graphical user interface.

- Even within the highly-specific context of a word-embedding-enabled database system, adaptation and code reuse of existing external library components remain an important part of web application development. Using state-of-the-art web frameworks both on the front-end and the back-end sides of the application is key to its swift development.

- The execution time of the complex vector operations within a word-embedding-enabled database system is the biggest bottleneck when interacting with it via a graphical front end. Moreover, its performance is highly dependent on available physical resources.

6.2 OUTLOOK

Despite the web application's suitableness for the use cases described in Chapter 3 and its fulfillment of the derived requirements, a few unresolved issues remain. The web application also lacks some desirable functionalities.

First of all, several changes to the web application's front end should be made. Even though responsive web frameworks were used in the application's development and its graphical interface was tested on several different screen sizes and resolutions, it is not yet fully compatible with mobile devices such as smartphones or tablets. Thus, it cannot be considered truly responsive. A code revision of the HTML templates and adjustments to the CSS stylesheets would be of benefit for allowing the comfortable usage of the web application on all kinds of devices. Secondly, taking the web application's accessibility to visually impaired people and people with color vision deficiency into consideration was out of the scope of this thesis. Extensive changes to the user interface should therefore be made in order to ensure its accessibility to people with different kinds of impairments, which may require the implementation of dedicated components to replace some of the external libraries used or modifications to them.

From an implementational point of view, the web application's code could benefit from a refactoring for the purpose of better component modularization. More precisely, it could make better use of AngularJS features such as multiple controllers, modules, services and factories. This would result in the code being more easily maintainable and more extensible.

During this thesis' writing, the unsatisfying performance of some word-embedding-enabled queries was identified as a major issue and a hindrance to the user's smooth interaction with the front end. The web application's back end could therefore be extended by advanced caching functionalities in order to improve the response time of the most frequently used queries.

Furthermore, several security issues were identified during the application's development. SQL injection and execution of denial of service attacks are both possible. Theoretically, the web front end could be exploited as an attack vector, as query customizability was chosen over implementing advanced query parsing to protect against malicious SQL queries. Both the front end and the back end's query and request processing could be improved by a dedicated query parser in order to protect the database system against attacks executed via the front end.

There are several possibilities for adding new functionalities to the web application. For example, an entirely new view focused on word embeddings data sets could be implemented. This could allow the comparison of the results of different word embedding operations using word embeddings trained on various corpora, thus providing further linguistic insights. Another possible feature is a functionality of executing batch word embedding operations on a column of a previous query's results. This is another use case which could benefit from the implementation of an advanced word-embedding-enabled query parser.

BIBLIOGRAPHY

[1] Bernard Marr. *How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read*. May 21, 2018. URL: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read (visited on May 25, 2018).

[2] Seth Grimes. "Unstructured Data and the 80 Percent Rule". In: *Clarabridge Bridgepoints Newsletter* (August 1, 2008). URL: https://breakthroughanalysis.com/2008/08/01/unstructured-data-and-the-80-percent-rule/ (visited on July 3, 2018).

[3] *NLP - natural language processing*. URL: https://www.webopedia.com/TERM/N/NLP.html (visited on May 25, 2018).

[4] Will Y Zou, Richard Socher, Daniel Cer, and Christopher D Manning. "Bilingual word embeddings for phrase-based machine translation". In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. 2013, pp. 1393–1398.

[5] William L Hamilton, Kevin Clark, Jure Leskovec, and Dan Jurafsky. "Inducing domain-specific sentiment lexicons from unlabeled corpora". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing. Conference on Empirical Methods in Natural Language Processing*. Vol. 2016. NIH Public Access. 2016, p. 595.

[6] Alessandro Sordoni, Yoshua Bengio, Hossein Vahabi, Christina Lioma, Jakob Grue Simonsen, and Jian-Yun Nie. "A Hierarchical Recurrent Encoder-Decoder for Generative Context-Aware Query Suggestion". In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. CIKM '15. Melbourne, Australia: ACM, 2015, pp. 553–562. ISBN: 978-1-4503-3794-6. DOI: 10.1145/2806416.2806493. URL: http://doi.acm.org/10.1145/2806416.2806493.

[7] *DB-Engines Ranking*. URL: https://db-engines.com/en/ranking (visited on May 28, 2018).

[8] Michael Günther. *Freddy Homepage on Github*. URL: https://github.com/guenthermi/postgres-word2vec (visited on May 28, 2018).

[9] Jason Brownlee. *What Are Word Embeddings for Text?* October 11, 2017. URL: https://machinelearningmastery.com/what-are-word-embeddings/ (visited on July 5, 2018).

[10] Galina Olejnik. *Word embeddings: exploration, explanation, and exploitation (with code in Python)*. December 3, 2017. URL: https://towardsdatascience.com/word-embeddings-exploration-explanation-and-exploitation-with-code-in-python-5dac99d5d795 (visited on June 1, 2018).

[11] Yoav Goldberg. “Neural network methods for natural language processing”. In: *Synthesis Lectures on Human Language Technologies* 10.1 (2017), p. 92.

[12] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. “A neural probabilistic language model”. In: *Journal of machine learning research* 3.Feb (2003), pp. 1137–1155.

[13] Aaron Geelon So and Roger Huang. *An Introduction to Word Embeddings*. August 28, 2017. URL: `https://www.springboard.com/blog/introduction-word-embeddings/` (visited on June 4, 2018).

[14] Tomas Mikolov, Quoc V Le, and Ilya Sutskever. “Exploiting similarities among languages for machine translation”. In: *arXiv preprint arXiv:1309.4168* (2013).

[15] Erik Bernhardsson. *How did Spotify get so good at machine learning? [...]* January 25, 2017. URL: `https://www.quora.com/How-did-Spotify-get-so-good-at-machine-learning-Was-machine-learning-important-from-the-start-or-did-they-catch-up-over-time/answer/Erik-Bernhardsson` (visited on July 3, 2018).

[16] Jennifer Slegg. *RankBrain: Everything We Know About Google’s AI Algorithm*. October 27, 2015. URL: `http://www.thesempost.com/rankbrain-everything-we-know-about-googles-ai-algorithm/` (visited on July 3, 2018).

[17] Ahmad Abdulkader, Aparna Lakshmiratan, and Joy Zhang. *Introducing DeepText: Facebook’s text understanding engine*. June 1, 2016. URL: `https://code.fb.com/applied-machine-learning/introducing-deeptext-facebook-s-text-understanding-engine/` (visited on July 3, 2018).

[18] Michael Günther. “FREDDY: Fast Word Embeddings in Database Systems”. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. Houston, TX, USA: ACM, 2018, pp. 1817–1819. ISBN: 978-1-4503-4703-7. DOI: `10.1145/3183713.3183717`. URL: `http://doi.acm.org/10.1145/3183713.3183717`.

[19] Michael Günther. “Integration von Word–Embeddings in Datenbanksysteme”. Diploma Thesis. Technische Universität Dresden, February 28, 2018.

[20] Jon Yablonski. *Laws of UX*. 2018. URL: `https://lawsofux.com/` (visited on June 25, 2018).

[21] IMDb.com. *What is IMDb?* 2018. URL: `https://help.imdb.com/article/imdb/general-information/what-is-imdb/G836CY29Z4SGNMK5?ref_=helpsect_cons_1_1#` (visited on June 25, 2018).

[22] Discogs. *About Discogs*. 2018. URL: `https://www.discogs.com/about` (visited on June 25, 2018).

[23] dblp team. *What is dblp?* 2018. URL: `https://dblp.uni-trier.de/faq/What+is+dblp.html` (visited on June 25, 2018).

[24] *AngularJS Developer Guide*. 2018. URL: `https://docs.angularjs.org/guide` (visited on July 5, 2018).

[25] Walter J Doherty and Arvind J Thadhani. “The economic value of rapid response time”. In: *IBM Report* (1982).

[26] William E Hick. “On the rate of gain of information”. In: *Quarterly Journal of experimental psychology* 4.1 (1952), pp. 11–26.

[27] Ray Hyman. “Stimulus information as a determinant of reaction time.” In: *Journal of experimental psychology* 45.3 (1953), p. 188.

[28] Jakob Nielsen. *Jakob’s Law of Internet User Experience*. 2017. URL: `https://www.nngroup.com/videos/jakobs-law-internet-ux/` (visited on July 2, 2018).

[29] Stephen E Palmer. "Common region: a new principle of perceptual grouping." In: *Cognitive psychology* (1992).

[30] Bluma Gerstein Zeigarnik. *Das Behalten erledigter und unerledigter Handlungen, Inaugural-Dissertation... von Bluma Zeigarnik,...* J. Springer, 1927.

[31] W3Techs. *Usage statistics and market share of Bootstrap for websites*. 2018. URL: `https://w3techs.com/technologies/details/js-bootstrap/all/all` (visited on June 19, 2018).

A APPENDICES

A.1 QUERIES FILE FORMAT

The keys of the top-level JSON object are the names of schemas contained in the database. The value of each schema's key is an array of query objects in the format described in Table A.1.

Key	Value type	Description
`query`	String	SQL query string. Formatting (line breaks and tabulators) can be used by entering the corresponding control characters.
`description`	String	A description of the SQL query to be shown in the web application's front end.
`type`	String	Type of the word embedding operation contained in the query. One of `'analogy'`, `'analogy_in'`, `'groups'`, `'similarity'`, `'knn'`, `'knn_in'`, `'knn_batch'` or `'custom'`.

Table A.1: A table containing a description of the query JSON format.

A.2 API

Property	Type	Description
`index`	String	Index to use for word embedding operations: one of `'RAW'`, `'PQ'` and `'IVFADC'`.
`pv`	Boolean	Enable/disable post-verification.
`pvFactor`	Number	Factor for post-verification.
`wFactor`	Number	Factor for IVFADC index.
`analogyType`	String	Analogy implementation to use: one of `'analogy_3cosadd'`, `'analogy_pair_direction'` and `'analogy_3cosmul'`.
`we`	String	Word embeddings data set to use: one of `'Google News'`, `'Wikipedia'` and `'GloVe'`.

Table A.2: A table containing a description of the settings JSON format.

URL	HTTP Verb	Parameters	Action
`/api/similarity`	GET	`keyword, results`	Execute a similarity query on keyword `keyword` with `results` number of results (only for IMDb data set).
`/api/knn`	GET	`query, k`	Find `k` nearest neighbors for word `query`.
`/api/knn_batch`	GET	`query_set, k`	Find `k` nearest neighbors for a list of words `query_set`.
`/api/knn_in`	GET	`query, k, input_set`	Find `k` nearest neighbors for word `query`, searching only in `input_set`.
`/api/analogy`	GET	`a, b, c`	Execute an analogy query with terms `a`, `b` and `c`.
`/api/analogy_in`	GET	`w1, w2, w3, input_set`	Execute an analogy query with terms `w1`, `w2` and `w3`, searching only in `input_set`.
`/api/grouping`	GET	`tokens, groups`	Match a set of tokens `tokens` to a set of grouping tokens `groups`.
`/api/tables`	GET	`schema`	Fetch a list of tables, their columns and column data types for a schema `schema`.
`/api/custom_query`	GET	`query`	Execute a custom user-defined query `query`.
`/api/settings`	POST	See Table A.2	Apply settings in POST request body to the FREDDY database system.
`/api/test_knn`	GET	`query_number, k`	Execute a k-NN performance test with `query_number` number of queries and parameter `k`.
`/api/prewarm`	GET	-	Load index tables into database server's main memory for improved performance.

Table A.3: A table containing a description of the web application's back end's API.