WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

EFMD EQUIS ACCREDITED

Master Thesis

Evaluation of Search Engines in the Context of RIS

Stefan Bauer, 0852385

Wirtschaftsuniversität Wien (WU), Welthandelsplatz 1, 1020 Vienna, Austria

Subject Area: Information Business

Supervisor: Prof. Dr. Axel Polleres, Dr. Jürgen Umbrich

Date of Submission: March 2016

Department of Information Systems and Operations, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

To my parents

Acknowledgements

I would like to thank the people how supported me during my studies and helped me in writing this thesis:

I would like to express my particular thanks to my family, who always supported me during my studies. Moreover, I would like to thank my supervisors Prof. Dr. Axel Polleres and Dr. Jürgen Umbrich, who guided me through the thesis and always contributed valuable input. Also, I would like to express my gratitude towards the Federal Chancellery of Austria, in particular Mag. Brigitte Barotanyi, who made this thesis possible. Finally, I would like to thank my friends, in particular Andreas Curik, who always gave advice in many situations during my studies.

Abstract

The Austrian Legal Information System (RIS) is a web based platform that grants access to national law, the law of the European Union, and the decisions of multiple courts. RIS currently uses a search system that is optimized for storing and retrieving information in a structured form. However, the legal RIS documents are mainly composed of unstructured information (text). Thus, many custom implementations were developed in order to provide the users a variety of different search functionalities. In the first part of the present thesis, we investigate the limits of the current search systems and evaluates alternative solutions. Since RIS was originally developed for professional users e.g. lawyers, judges, we discuss further concepts to improve the overall search experience in the second part of the thesis.

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
BLOB	Binary Large Object
CSV	Comma-Separated Values
CRC	Cyclic Redundancy Check
ETL	Extract Transform Load
FTS	Full-Text Search
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IE	Information Extraction
IR	Information Retrieval
JDBC	Java Database Connectivity
NER	Named Entity Recognition
NLP	Natural Language Processing
RDMS	Relational Database Management System
RIS	Austrian Legal Information System
SDK	Software Development Kit
SQL	Structured Query Language
TFIDF	Term Frequency Inverse Document Frequency
SVM	Support Vector Machine
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 Introduction

The retrieval of information and knowledge from documents and the structured data has always been an important topic. However, with invention of the computer in the last century, the amount of information significantly increased every year. Relational databases, which are optimised for structured data, are still the most important systems to store and retrieve information. Due to the increasing amount of unstructured information, new full-text search systems were developed dealing with textual content in an effective way. Those full-text search systems cover a variety of uses cases e.g. web search, enterprise search, and range from commercial to open-source solutions. Also, relational database vendors started to recognize their limits when dealing with large amounts of unstructured information and started to integrate full-text search functionalities in their solutions.

In this thesis, we investigate full-text search from the viewpoint of a particular use case around storing and searching legal texts. The Austrian Legal Information Systems (RIS), which is hosted by the Federal Chancellery of Austria, is an information system that provides those legal texts through a web interface. The legal content primarily covers the Austrian law and is accessed by professionals e.g. lawyers, judges, as well as non-professionals. RIS uses a relational database to store and retrieve the legal content. However, the current system has some issues. On the one hand, it has plenty of customized implementations that were developed to provide additional full-text search features. Those implementations have evolved historically and are difficult to maintain. On the other hand, the current search systems lacks various full-text search features that are well known from other Web-sites e.g. automatically suggesting alternatives in terms of misspelled words, refining the search through categories. Finally, RIS is a platform that is accessible to everyone. However, retrieving the desired content within a reasonable amount of time is sometimes challenging for non-professionals.

1.1 Motivation

The objective of the present thesis is to find solutions, which allow to mitigate the limitations of the current RIS search system. To this end, we evaluate alternative search systems and compare them to the current solution. Moreover, we discuss various concepts in order to improve the user experience by enriching the search with external information.

1.2 Structure

In chapter 2, we describe the basic full-text search features and evaluation methods as well as the most important features of full-text search systems. Moreover, we discuss methods and techniques, which can be applied to extract additional information from textual content.

In chapter 3, we describe the current RIS search system from different viewpoints. Therefore, we address the access statistics and the technical architecture, as well as the basic full-text search features of current system.

In chapter 4 we evaluate and compare alternative search systems. Therefore, the Federal Chancellery of Austria provided us with a set of requirements that a new search system must meet. Those requirements range from standard full-text search features e.g. wildcard search, boolean search, to RIS specific features e.g. segmentation of specific parts in legal documents. First, we select the alternative search systems and describe the respective technical characteristics. Next, we consider the implementation of the requirements in respective solution and compare them to the current RIS search system. Finally, we evaluate the search systems based on several methods and techniques.

In chapter 5, we describe different possibilities for improving the overall search experience. We discuss concepts, which range from the integration of external sources to the extraction of information from RIS documents.

2 Preliminaries

This chapter describes the basic concepts of Information Retrieval (IR). We explain the concepts of IR based on the following example documents, which we randomly selected from the RIS platform.

Document D1:

> Das Verfahren über die Bestellung eines Sachwalters ist in jeder Lage einzustellen, wenn das Gericht zu dem Ergebnis gelangt, dass ein Sachwalter nicht zu bestellen ist. Dabei besteht zum einen keine Verpflichtung des Gerichts, einen bestimmten Verfahrensabschnitt abzuwarten, zum anderen hat auch noch der Oberste Gerichtshof eine Einstellung auszusprechen, wenn er mit der erforderlichen Sicherheit zu dem Ergebnis gelangt, dass die Voraussetzungen für eine Sachwalterbestellung nicht vorliegen.

Document D2:

> Der gemäß §20 RL-BA 1977 von einem Rechtsanwalt im Falle eines persönlichen Streites aus der Berufsausübung mit einem anderen Rechtsanwalt um Vermittlung anzurufende Ausschuss der Rechtsanwaltskammer ist kein Schiedsgericht im Sinne des §587 ZPO.

Document D3:

> Die gemäß §28 RAO dem Ausschuss der Rechtsanwaltskammer obliegende Aufgabe, bei Meinungsverschiedenheiten zwischen Kammermitgliedern im Rahmen ihrer Berufsausübung zu vermitteln, stellt im Zusammenhalt mit §37 RAO die Grundlage dar, dass der Österreichische Rechtsanwaltskammertag Richtlinien zur Ausübung des Rechtsanwaltsberufes erlassen darf. Bei diesen RL-BA 1977 handelt es sich um Verordnungen.

Information Retrieval (IR) describes the field of retrieving information from different kind of sources. These sources can be music files, pictures, texts. However, in the academic area IR refers often to the sub domain text retrieval, which deals with the retrieval of information from textual content. Therefore, we define the term IR as follows.

"Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers) [25]."

Writing and storing information has a long tradition and goes back to around 3000 BC, when the Sumerian population designed clay tablets with some inscriptions. The Sumerians figured out that efficient use of information is very crucial. To accomplish that goal, they developed some classification techniques in order to identify every tablet and the content it contains. This system allowed them to create information, store it centrally, and retrieve it with the help of intermediaries [32].

This was just a short journey to the past, but reflects very well that the need of storing and retrieving information is very important and dates back to the first civilisations. The simple model of the Sumerians worked well enough at that time. However, in the last centuries, especially when paper and printing press were invented, other archiving techniques had to be developed. With the invention of the computer, people soon realized that it can be used to store and retrieve big amounts of information in an automated way.

Vannevar Bush, an American engineer and inventor, published in 1945 the article "As We May Think". At that time he was director of the Office of Scientific Research and Development, an organisation that carried out almost all research and development activities during World War II. In his article Bush describes his vision of a machine that represents a collective memory and allows users to automatically process and access large amount of knowledge [7].

In the 1950s various ideas emerged, which substantiate the concept of searching text through a computer. H.P. Luhn described in 1957 an important method. He proposed using words for creating an index and measuring the overlap of words for the ranking of documents [21].

In the 1960s more developments in the field of IR took place. One development was certainly the SMART system by Gerard Salton and his students at Harvard University. The SMART is a framework that provides

researchers the possibility to improve search quality through multiple experiments [35]. The other development was the Cranfield evaluation technique by Cyril Cleverdon and his colleges at College of Aeronautics in Cranfield. The Cranfield test is an evaluation method for retrieval systems, which current IR systems still use [11]. Both methods were the reason for quicker progress in the field of IR.

The 1970s and 1980s were characterised by various improvements of the previous methods. The models and techniques were very efficient in terms of smaller data sets. Unfortunately, they did not scale when dealing with big amounts of textual content. The Text Retrieval Conference (TREC) in 1992 changed this issue dramatically. TREC is series of conferences supported by some US Government agencies. The main target of these conferences is to bring researchers together and support them in developing methods that allow to deal with larger text collections in an efficient way [19].

TREC offers in their conferences different tracks, each representing a different domain. These tracks contain domain specific information like document collections, training judgements, evaluation tools. There is also a legal track, that is explicitly designed for legal professionals. On the one hand, it should help them to get insights into current tools and methods, and on the other hand provide them with test collections such that IR systems can be evaluated properly.[1]

Full-text search engines play an important role in the field of information retrieval. We use the term search engine as an short version for a full-text search engine. They are mainly responsible for storing and retrieving data. Figure 1 illustrates an abstract architecture of the two tasks of (i) indexing documents and (ii) retrieving the respective documents corresponding to (concurrent) user search queries. The index is the central data source of all search engines and is created by splitting the documents into a list of unique terms. Once the index is created, information can be retrieved through search queries. Search engines provide functionalities to process search queries concurrently and rank the results accordingly. Retrieving information from the index is efficient, since no full scan of every word in each document has to be performed, which would be the case in a search without an index. Since documents are frequently changed, an index has to be constantly updated. Updates on the index are carried out either dynamically during operation, or in static batch jobs while the search engine is not available.

[1] http://trec.nist.gov/data.html

Search engines must be able to meet several challenges e.g. effectively creating and updating the index, retrieving content within a reasonable amount of time, presenting the relevant content. In the following sections, we describe those challenges in more detail.

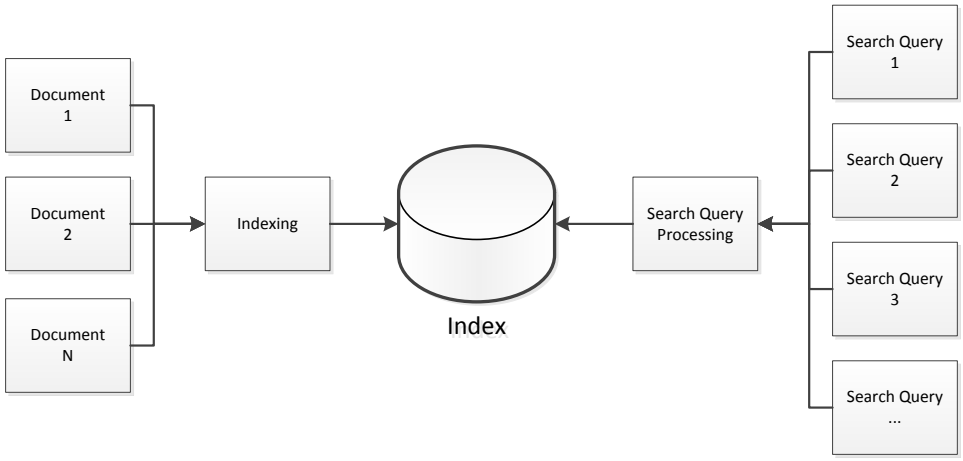

Figure 1: Core Architecture of a Search Engine

2.1 Index

An index is used to store information in such a way that it can be faster retrieved through search queries. Thus, the index is an essential part of a search engine's architecture. In the following section, we describe the structure of a typical index in more detail.

The inverted index is the most common index structure used in the field of textual retrieval. *Dictionary* and *posting lists* are the two main components of an inverted index. The dictionary represents a list of unique terms, extracted from the documents. This dictionary is usually kept in-memory due to performance reasons. It stores the term and the pointer to the posting list for each term. The posting list of each term represents a list of documents the term appears in and is usually stored on disk. Some posting lists contain additional information e.g. position, offset, frequency of a term in the respective document [11].

The dictionary and the posting list, are used to process search queries more effectively. The dictionary provides a logical structure on top of the posting list, which is used to retrieve the terms of the search query in a fast way. The entries in the posting list are then used to retrieve and rank the relevant documents matching to the search query. The two main tasks of a search engine (i) indexing and (ii) search query processing, are usually complementary. This means that the more information is stored in the index, the less work has to be done while processing search queries [11].

We describe an example of an index entry creation based on the document D1 from section 2 in the following part. Creating index entries for the document requires first of all the division of the text into parts, see the box below. These parts are called tokens. In our case, we create tokens based on the white space between the terms. However, other rules can be applied as well for tokenization.

Das | Verfahren | über | die | Bestellung | eines | Sachwalters | ist | in | jeder | Lage | einzustellen | [...]

The tokens are then further processed. This process may include a variety of different filters e.g. removal of specific words, transforming all capital letters to lower case letters. The filters are used to reduce the amount of terms in the index and increase the efficiency of retrieving the appropriate results. The most important filters are described below.

Stop words: A stop word is a word that appears in a phrases, but does not influence the syntax and the semantic of the respective phrase. In most cases theses words are conjunctions like "and", "or", "because". Different languages have different stop words. Thus, applications with a full-text search on different languages must incorporate the list of stop words for the respective language. In most cases, stop words are not indexed, which results in a reduction of the index size. The snowball project provides a common list of German stop words.[2] Snowball is a language that processes natural language through several algorithms. They also integrate a list of German stop words, which many full-text search engines use. The stop words are crossed out in the following example. All those words are not indexed.

[2] http://snowball.tartarus.org/algorithms/german/stop.txt

~~Das~~ Verfahren ~~über~~ ~~die~~ Bestellung ~~eines~~ Sachwalters ~~ist in jeder~~ Lage einzustellen, ~~wenn das~~ Gericht ~~zu dem~~ Ergebnis gelangt, ~~dass ein~~ Sachwalter ~~nicht zu~~ bestellen ~~ist~~. Dabei besteht ~~zum einen keine~~ Verpflichtung ~~des~~ Gerichts, ~~einen~~ bestimmten Verfahrensabschnitt abzuwarten, ~~zum anderen hat auch noch der~~ Oberste Gerichtshof ~~eine~~ Einstellung auszusprechen, ~~wenn er mit der~~ erforderlichen Sicherheit ~~zu dem~~ Ergebnis gelangt, ~~dass die~~ Voraussetzungen ~~für eine~~ Sachwalterbestellung ~~nicht~~ vorliegen.

Stemming: Some search engines make use of algorithms that reduce each word to its stem. Stemming can be defined as followed.

"A stemmer should conflate together all and only those pairs of words which are semantically equivalent and share the same stem [9]."

Stemming allows to retrieve more variations of a word, which normally increases the set of search results. However, stemming has some disadvantages. Stemming algorithms produce sometimes incorrect stems e.g. "ly" can be removed from "cheaply" but not from "reply". Those incorrect stems typically increase in languages that are morphologically rich e.g. German. Morphology describes the study of the forms of words (plural forms, tenses, persons, etc.) [2]. Thus, some search engines integrate a stemming list. The disadvantage is that indexing takes longer, because every word has to be checked against the list. Moreover, creating and maintaining such a list requires a lot of time and effort. A simple and lightweight algorithm was developed by Jörg Caumanns at the Free University of Berlin. It was especially designed for morphologically complex languages like German or Dutch. The most important parts and characteristics of the algorithm are described below [8].

- **Substitution**: If a token is stemmable, substitutions of some characters are performed e.g. "österreich" would be transformed to "osterreich". However, there are also examples where a substitution can lead to false positives e.g. "schönung" (fining) would be transformed

Character	Substitute
ä	a
ö	o
ü	u
ß	ss

Table 1: Stemming Substitutions

to "schonung" (protection). Table 1 shows an extract of the characters being replaced by the algorithm.

- **Suffix**: Suffix-stripping is an important part of the stemming process. The following suffixes are stripped at the end of each term: "e", "s", "n", "t", "em", "er" and "nd". Although, German has a lot more suffixes, only these 7 were chosen because of the lower error rate e.g. "Spieler" would be transformed to "Spiel". However, this simplification still causes various errors and irregular stems, but produces in most cases unique discriminators. The discriminator is the common form of different declensions of the same word. Moreover, it makes the algorithm much more efficient.
- **Prefix**: Analogously to suffix-stripping, prefix-stripping can remove (semantically) redundant prefixes. The algorithms strips all "ge" prefixes e.g. "gegangen" would be transformed to "gangen". However, examples exist where the prefix-stripping can lead to false positives e.g. "genau" (exact) would be transformed to "nau", which does not relate to the original term any more.

We applied the substitution, suffix-stripping and prefix-stripping rules that are previously described on document D2 from section 2. The result can be seen in the following box.

> Der ~~ge~~m**ass** §20 RL-BA 1977 von einem Rechtsanwal~~t~~ im Falle eine~~s~~ pers**o**nliche~~n~~ Streite~~s~~ aus der Berufsaus**u**bung mit einem andere~~n~~ Rechtsanwal~~t~~ um Vermittlung anzurufende Ausschuss der Rechtsanwaltskamme~~r~~ ist kein Schiedsgericht im Sinne des §587 ZPO.

As already described, stemming algorithms produce false stems in some cases due to simplification reasons. Thus, significantly reducing those false stems requires the hard coding of all the exceptions for substitutions, prefix-stripping and suffix-stripping.

Synonyms: A synonym refers to words or phrases that share the same meaning e.g. intelligent can be associated with smart, bright, brilliant, sharp.[3] A list of synonyms can either be created individually or downloaded from Web-sites that maintain synonym lists for the respective language. There are various German synonym lists available. A popular one can be found at Open Thesaurus.[4]

After creating the tokens and applying the filter rules, the index is created and stored. Dictionary and posting lists define the structure of an inverted index. Both concepts are described below.

Dictionary: The task of the dictionary is to provide a list of terms pointing to the relevant posting lists on the disk. The terms extracted from the tokenization process are added to the dictionary. Table 2 shows the structure of such a dictionary based on our example. It contains a list of unique terms, which are ordered lexicographically, and the pointer to respective posting list of the term. Due to simplification reasons we did not apply any filtering rules (stop words, stemming, etc.) and displayed only an extract of the ordered terms. In comparison to the total size of the index, the dictionary is typically small. The structure of the dictionary is essential for the performance of the queries. Most commonly the dictionary is either sort-based or hash-based [11].

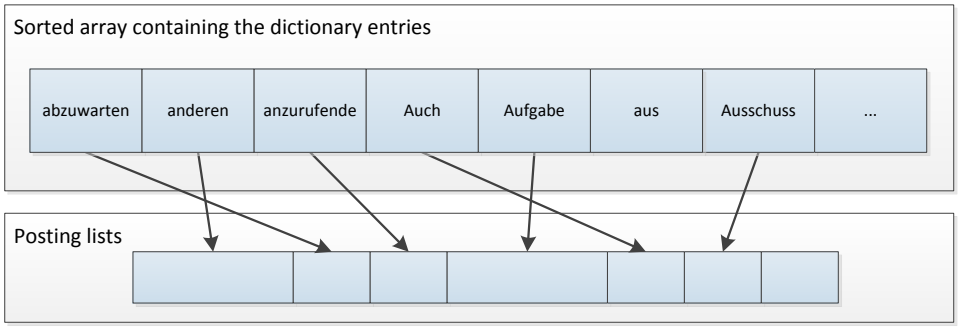

Figure 2: Sort-based Dictionary

[3] http://www.oxforddictionaries.com/de/definition/englisch_usa/synonym
[4] https://www.openthesaurus.de/about/download

Term	Pointer
abzuwarten	1
anderen	2
anzurufende	3
auch	4
Aufgabe	5
aus	6
Ausschuss	7
auszusprechen	8
Ausübung	9
bei	10
Bei	11
...	...

Table 2: Dictionary

- **Sort-based**: All terms of a collection are sorted in a lexicographical order. The terms are arranged either in a sorted array or in a sorted search tree. When using a search tree, the respective information is retrieved through a traversal of the tree. Look-up operations in a sorted list are performed through a binary search. A sorted search tree is usually created, if the sorted list is too big. The effort for the retrieval of information in a sorted list or a sorted search tree is logarithmic. Figure 2 illustrates the dictionary structure based on a sorted array.

- **Hash-based**: Each term in the dictionary has a corresponding entry in a hash table. The hash table consists of hash values, which can be created through various hash functions. These functions can be used to map data of an arbitrary size to data of a fixed size e.g. "John" to "01", "Lisa" to "02". However, it might happen that 2 or more input data share the same hash value, which refers to collisions e.g. "John" to "01", "Lisa" to "01". The problem of collisions can be solved through a linked list of those terms. Figure 3 visualizes the hash-based dictionary based on our example. It displays a hash table, which is used to map keys to an array of buckets. The keys, which are the terms of the dictionary, are used to calculate a hash value through a hash function H(key). There are several hash functions available e.g. CRC32, DJB2, FNF-1. In our example, we selected the hash function CRC32. The Cyclic Redundancy Check (CRC) is a function that creates a check value from the input data through polynomial divisions. It is often used in communication and transmission systems in order to verify

changes in the data. Since CRC creates check values of a fixed length, it can be used as a hash function as well [5]. After creating the hash values through the CRC32 function, we assigned the values through the modulo function to the buckets. The modulo function gets the remainder of a division, which is used to determine the position in the array of buckets. We wrote a simple application in Java, which can be found in appendix A.1, in order to create and assign the hash values. The buckets contain the information, where the data entries can be found on the disk. The data entries are composed of the term itself, the position of the term's posting list and a pointer to the next entry in the linked list, if multiple terms share the same hash value (collision).

Sort-based and hash-based structure are appropriate depending on the type of the query. If a single term is queried, hash-based indexes are much more efficient than sort-based ones. The reason for this is that a less cost intensive operation is required to retrieve the dictionary entry for the relevant term. The operation of retrieving the dictionary entry in constant time, is based on the assumption that only a few or no hash table collisions occur. Search queries retrieving a specific range or terms with a certain prefix are much more efficient in a sort-based index, since a binary search is applied. The search is executed in logarithmic time. Thus, in practice it is common having both index structures [11].

Posting list: The posting list contains the actual data of the index. This data is used during the querying process to retrieve the documents matching to the respective query. Each posting list contains at least the pointer to the corresponding term in the dictionary, and the documents the term appears in. Additionally, the posting list can also contain further information e.g. position, offset and occurrence of a term in the respective document [11]. The position describes the location of the term in the document e.g. "abzuwarten" appears in document D1 on the position 38. The offset describes the start (first character) and end offset (last character) of a term in the respective document e.g. "Das" in document D1 has the start offset 0 and the end offset 3, "Verfahren" has the start offset 4 and the end offset 12. The occurrence determines the frequency of the term in the respective document e.g. "anderen" appears once in document D1 and once in document D2. Table 3 reflects an extraction of the posting lists based on the documents in section 2, where each row represents a posting.

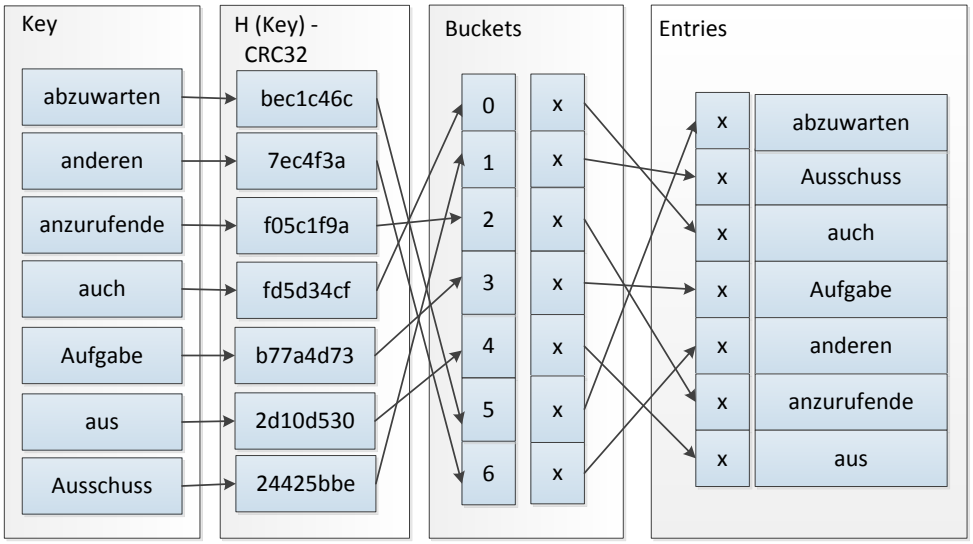

Figure 3: Hash-based Dictionary

2.2 Search Queries

In section 2.1, we described the creation of an index. In this section, we focus on retrieving information from an existing index. Search engines integrate various search features. The most common features, which we explain based on the documents from section 2, are described below.

Boolean Search: A boolean search applies boolean operators for the retrieval of information. The classical boolean operators are AND, OR and NOT. The query *"Verfahren AND Sachwalterbestellung"* retrieves document D1, because the document contains "Verfahren" and "Sachwalterbestellung". In comparison to the AND operator, the OR operators retrieves the documents D1 and D2 with the query *"Verfahren OR Rechtsanwalt"*. The OR operator retrieves documents if and only if they contain "Verfahren" or "Sachwalterbestellung" or both. The query *"Rechtsanwaltskammer NOT Meinungsverschiedenheit"* does not retrieve document D3, since it is only satisfied if and only if a document contains "Rechtsanwaltskammer" but does not contain "Meinungsverschiedenheit". The boolean operators can be combined in order to construct more complex queries according to the classical rules of the boolean algebra [17]. The hash-based approach can be applied in a boolean search, see section 2.1, since less costs are a necessary to retrieve

Pointer	Term	Document	Position	Occurrence
1	abzuwarten	D1	38	1
2	anderen	D1	40	1
2	anderen	D2	20	1
3	anzurufende	D2	24	1
4	auch	D1	42	1
5	Aufgabe	D3	11	1
6	aus	D2	15	1
7	Ausschuss	D2	25	1
7	Ausschuss	D3	7	1
8	auszusprechen	D1	49	1
9	Ausübung	D3	38	1
10	bei	D3	12	1
11	Bei	D3	43	1
...	...	...	...	...

Table 3: Posting List

the information. The reason for this is that the matching term is retrieved in constant time through the hash value. In the sort-based approach, the dictionary has to be scanned in logarithmic time before retrieving the relevant information, which results in more costs.

Wildcard Search: Wildcards are place-holders, which allow the user to find different variations of a term. Thus, wildcard queries are mostly used if the user is uncertain about the spelling of a term e.g. Sidney vs. Sydney, or different variations of a term. In many search engines the character "*" is used to match any sequence of characters. However, it is also possible to determine the amount of characters to be matched. For the latter characters like "?" or "_" are applied. The different types of wildcard queries are listed below [25]. The sort-based approach, see section 2.1, for the retrieval of information is more applicable for the wildcard search. The reason for this that the entire dictionary is scanned in logarithmic time in order to retrieve the matching documents. This is less cost intensive compared to the constant time effort of the hash-based approach.

- **Right Wildcard**: In the right wildcard search the user wants to match any sequence of characters at the end of the term. Thus, the query *"Sachwalter*"* would retrieve results like "Sachwalters", "Sachwalterbestellung" and "Sachwalter" in document D1. The right wild-

card search is very efficient, since it makes use of the inverted structure of the index.

- **Left Wildcard**: This type of wildcard is also known as leading wildcard. It is used if the user wants to match any sequence of characters at the beginning of the term. The query *"*stellen"* would match in document D1 also the terms "einzustellen", "bestellen" and "Einstellen". The left wildcard search is typically less efficient, since the query terms and the terms in the dictionary have to be reversed while executing the query. Effective search engines meet the performance issue by storing each term in the dictionary in an additional reversed form.

- **Middle Wildcard**: This type of wildcard sets the placeholder in the middle of a term. So if the user queries for *"Recht*kammer"* the documents D2 and D3 would be returned, since they both contain "Rechtsanwaltskammer". The middle wildard behaves similar to the right wildcard.

- **Precise Wildcard**: The precise wildcard is used to determine single characters that should be replaced by any sequence of characters. In our example, if a user searches for *"Verfahrens__schnitt"* the term "Verfahrensabschnitt" in document D1 would match. The precise wildcard behaves similar to the right wildcard.

Phrase Search: In a phrase search the user looks for multiple terms that must appear in documents according to the sequence in the query. Since the order of the terms for such queries is essential, the position of each term must be stored in the index. In many search engines a phrase search is determined by placing one quote at the beginning and one quote at the end of the phrase. So if a user is looking for *"Das Verfahren über die Bestellung"* document D1 would be retrieved. On the contrary, if the query *"Das die Bestellung Verfahren über"* is carried out, document D1 would not be retrieved.

Near Search: As in the phrase search, the near search requires the position of each term to be stored in the index. In a near search, terms or phrases that are close to each other are retrieved. Whereas, the query *"Gericht NEAR Ergebnis"* would retrieve document D1, the query *"Verfahren NEAR Sachwalterbestellung"* would not do so. Furthermore, some search engines enable specifying the distance between two terms or phrases.

Range Search: Many search engines support range queries on ordered values. The order could be based on numbers or the alphabet. When dealing with range queries, a dictionary with a sorted list or a sorted search tree structure is essential. The reason for this is that a binary search can be applied on the dictionary in logarithmic time, which requires less costs compared to the hash-based approach. In our example, we can assume that the documents additionally store the following paragraph numbers.

Document D2: "§20"
Document D3: "§28"

If the user searches for *"§20 TO §28"* documents D2 and D3 would be returned, since it matches the range specified in the query. However, if the user searches for *"§31 TO §41"* no documents would be returned, since the range from the query does not match to the range from the documents.

Auto Correction: Automatic corrections support the user in case of misspelled terms. The search engine provides additional suggestions through "Did you mean?" if it recognizes differences of a user entered term. If the user is looking for *"Gerucht"*, the search engine would suggest for example "Gericht". The suggestions are based on a list of correct terms. This list can be either created and maintained manually or automatically. The manual generation is exact, since the correct terms are manually chosen, but also resource and time intensive. That's why many search engines supports to automatically generate the list from the index.

The terms of the search query are then checked against this list. Suggesting alternative terms requires the measurement of the similarity between the query term and the list of correct terms. The most important technique in this field is the edit distance. The edit distance between the terms T1 "Buch" and T2 "Tuch" is the minimum number of edit operations necessary to transform T1 into T2. The most common edit operations are

- insert a character into a string,
- delete a character from a sting,
- replace a character from a string.

The less operations are necessary, the more similar are the terms T1 and T2. In this case, one operation would be necessary that replaces the character "T"

	$i(\rightarrow)$	B	U	C	H
$j(\downarrow)$	0	1	2	3	4
T	1	1	2	3	4
U	2	2	1	2	3
C	3	3	2	1	2
H	4	4	3	2	**1**

Table 4: Levenshtein distance

in T2 by "B" in order to match both terms. The edit distance, which is also known as Levenshtein distance can be adapted by using different weights for the operations [25]. The similarity is calculated through the following algorithm.

$$d_{i,j} = min(d_{i,j-1} + d_{weight}, d_{i-1,j} + d_{weight}, d_{i-1,j-1} + d_{x_i=y_j}) \quad (1)$$

The algorithm compares each character of the two terms in order to measure the minimum edit operations. Table 4 illustrates the steps of the algorithm. In our case we use use a d(weight) of 1. If two characters match d(xi=yj) we apply 1 and if they don't match we apply 0. The algorithm loops through all characters of both terms. It starts by comparing the first character of T1 "B" with the first character of T2 "T". In this case d(i-1,j)+d(weight) is 1+1, d(i,j-1)+d(weight) is 1+1 and d(i-1,j-1)+d(xi=yj) is 0+1 since "T" and "B" don't match. The algorithm proceeds until the last position, which determines the edit distance, which is 1 in our case.

Taxonomy: A taxonomy describes a structure that organises information in a hierarchical order [39]. It is used to group information based on categories e.g. date, time, topics, and is applied in various search features e.g. faceted search. A faceted search uses the taxonomy in order to display the number of documents according to the search query in each group. A taxonomy is structured similar to the index, described in section 2.1. The difference, however, is that the dictionary of a taxonomy is ordered according to the respective hierarchy. Figure 4 illustrates a faceted search from the job portal Xing.[5] We executed the query "data scientist" and received 1,067 results. On the right side a taxonomy can be seen, which maps the results to the pre-defined hierarchy. The first level of the hierarchy is "Tätigkeitsfeld", "Karrierestufe", etc. The second level of the hierarchy contains in "Tätigkeitsfeld" the facets "Forschung, Lehre und Entwicklung (597)", etc. The numbers in the brackets determine the amount of results according to the query in the respective facet e.g. 597 results in the facet "Forschung, Lehre und Entwicklung".

[5] https://www.xing.com/jobs/

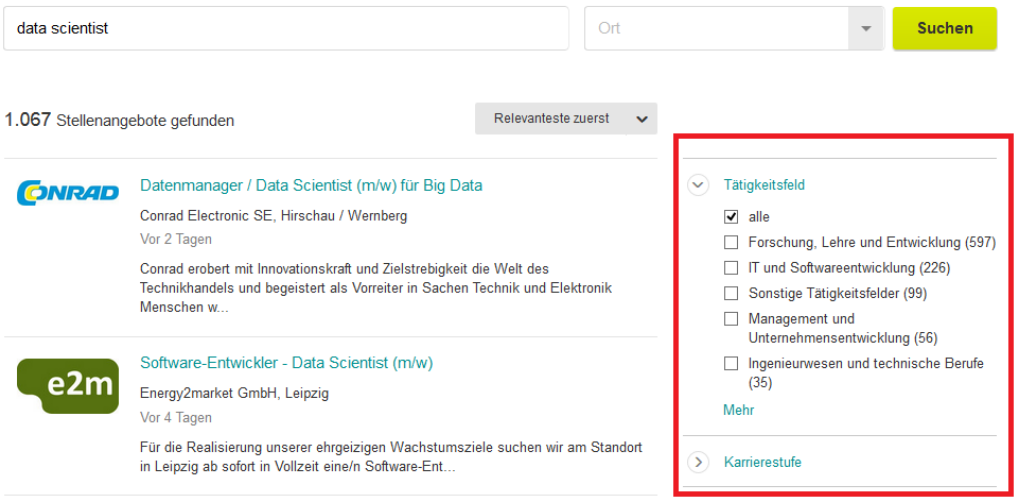

Figure 4: Faceted Search

2.3 Ranking

The ranking of documents is an important task of a search engine, since it's purpose is to retrieve the matching documents according to it's relevance. In the field of IR, several models and techniques for the ranking of documents exist. However, two 2 major categories can be identified: semantic and statistical ranking methods. Whereas the semantic approaches try to understand the natural language text of a human user, the statistical approaches rank the documents, which contain the information, based on some statistical measures that match the user query most closely [17]. In the present thesis, we focus on the statistical models and describe the main models below.

Early IR systems used complex boolean operations in order to provide the user information. The disadvantage is that those systems don't have sophisticated document ranking functions, since they simply express whether a document matches a query or not. Although, boolean systems have ranking options e.g. ordered by date, their rankings are very limited. However, in large document collections, it is not reasonable for the user to look through the whole list of resulting documents. Consequently, those boolean systems are more and more replaced by other solutions, which rank the the documents matching to the query of the user. To do so, many IR systems assign a score to every document/query combination. The results are then ranked based on this score [36].

Before computing the score, a weight has to be assigned to each term in a document. This weight depends on the number of occurrences of the term in the document. The simplest approach, to compute the score of a query *term* (t) and a *document* (d), is to assume that the weight is equal to the number of occurrences of (t) in (d) e.g. "das" appears twice in document D1, thus the weight for the term "das" would be 2. This approach is called *term-frequency* (tf) [34].

Unfortunately, it is not always the case that all terms are equally important, which the term-frequency approach assumes. Thus, some terms are more relevant than others e.g. the term "Gesetz" (law) appears in many RIS documents and is therefore not so important than the term "Sachwalter" (trustee), which only appears in specific RIS documents. A common approach to determine the relevance of terms is the *document-frequency* (df), which computes the number of documents that contain the respective term t e.g. term "Gesetz" appears in 181,861 RIS documents, whereas the term "Sachwalter" can be found in 4,664 RIS documents. Consequently, the term "Sachwalter" is weighted higher, since it appears in less documents [34].

The term-frequency weights terms higher that appear in more documents. By contrast, the document-frequency weights terms higher that appear in less documents. To combine both approaches, the *inverse-document-frequency* (idf) has to be calculated [34].

$$idf_t = log \frac{N}{df_t} \tag{2}$$

In the formula above, N describes the total number of documents. RIS has currently 1,466,207 document. Thus, the idf for the term "Sachwalter" is 2,50 and the idf for the term "Gesetz" is 0,91. This means that the higher the idf the higher the importance of the term. The term-frequency (tf) and the inverse-term-frequency (idf) can now be combined to assign to term (t) a weight in document (d). These weights are essential for calculating a score for each query/document pair, such that the most relevant documents according to a query are ranked first. Since, query and document can be represented as vectors, many search engines apply the vector space model for measuring the score for each query/document combination [34].

Vector Space Model: The vector space model represents documents and queries a vectors. A *vector* (V) is a list of (N) *numbers*, which can be repre-

sented in form of a column [23].

$$V = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \tag{3}$$

Each document becomes an independent dimension in the vector space. The tf or tf-idf weights can be determined for every term in each document and inserted in the vector space. Since not every term from the dictionary appears in all documents, most vectors are sparse. This means that the number 0 is assigned to a term, if it doesn't occur in the respective document [23]. Below is a simple example of a vector space with 3 documents, 3 terms and the corresponding the tf-weights.

$$\begin{array}{l} \\ Gesetz \\ Sachwalter \\ neu \end{array} \begin{array}{c} \begin{array}{ccc} Doc1 & Doc2 & Doc3 \end{array} \\ \begin{pmatrix} 5 & 20 & 1 \\ 2 & 3 & 0 \\ 15 & 2 & 8 \end{pmatrix} \end{array} \tag{4}$$

But not only the documents are vectors, also the query itself can be represented as a vector. A typical approach to calculate the score for each query/document pair, is the cosine similarity. This approach computes the similarity between the *query-vector* V(q) and the *document-vector* V(d). The numerator is the inner product (dot product) of the vectors V(q) and V(d), while the denominator represents the product of the Euclidean lengths [23].

$$score(q,d) = \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)||\vec{V}(d)||} \tag{5}$$

The inner product multiplies is the sum of the product of two vectors e.g. $V(Doc1) \cdot V(Doc2) = (5 \cdot 20) + (2 \cdot 3) + (15 \cdot 2) = 136$.

$$InnerProduct(q,d) = \sum_{i=1}^{n} V(q)_i \cdot V(d)_i \tag{6}$$

The Euclidean length is the distance is the distance between two vectors e.g. $|V(Doc1)||V(Doc2)| = (5^2 + 2^2 + 15^2)^2 \cdot (20^2 + 3^2 + 2^2)^2 = 323,89$.

$$EuclideanLength(q,d) = \sqrt{\sum_{i=1}^{n} V(q)_i^2} \cdot \sqrt{\sum_{i=1}^{n} V(d)_i^2} \tag{7}$$

Term	Query				Doc1		
	tf	df	idf	tf-idf(t,q)	tf	idf	tf-idf(t,d)
Gesetz	1	181,861	0,91	0,91	5	0,91	4,55
Sachwalter	1	4,664	2,50	2,5	2	2,5	5
Neu	0	76,844	1,28	0	15	1,28	19,2

Table 5: Vector Space

Inserting the values of the inner product and the Euclidean length in the score formula produces the result of 0,42. The score (cosine similarity) is bound between 0 and 1, if all values of the vectors are positive. A cosine similarity of 1 means that the vectors have the same orientation, which means that the vectors perfectly match. By contrast, a cosine similarity of 0 means that the vectors are orthogonal and therefore not similar.

Table 5 shows an example of a possible vector space. In the example, we assume the query (q) "Sachwalter Gesetz", which we use to measure the score of document (Doc1) from the previous example. In the example we assume a total number of 1,466,207 documents. We compute the tf-idf weight for each term, such that we can create a query vector tf-idf(t,q) and a document vector tf-idf(t,d). Applying the score formula on both vectors produces the score 0,67. This means that the query (q) is similar to document (Doc1).

2.4 Evaluation Methods

Analysing and evaluating the quality of search results requires a test collection. TREC provides such test collections for different domains. The search results are then compared to those test collection. The most common techniques are precision and recall. Those two concepts are used to determine the accuracy of the search results [38].

Recall relates the relevant retrieved results (Ra) to the set of relevant documents (R).

$$Recall = \frac{Ra}{R} \tag{8}$$

In contrast to recall, precision is defined as the proportion of the relevant

retrieved results (Ra) to all documents (A).

$$Precision = \frac{Ra}{A} \tag{9}$$

The F-measure is a way to calculate the effectiveness of a search application. It is based on recall and precision and used to evaluate the classification performance.

$$F = \frac{2RP}{R + P} \tag{10}$$

Assuming that a test collection has 100 relevant retrieved results (Ra), 200 relevant documents (R) and a total number of 500 documents (A), would produce a recall of 50 %, a precision of 20 % and an F-measure of 0,29 %. This means that half of the results are positive, but only 20 % of the results are accurate. The low F-measure of 29 % indicates that the search engine does not retrieve results effectively.

The relationship between precision and recall is in general inverse. The more items are retrieved by the search engine the greater is the probability that relevant documents from the overall collection are retrieved. The recall has the value 1 when all documents are retrieved. However, this effect results in a retrieval of more irrelevant documents. Thus, the precision will decrease, the more documents are retrieved [38].

The test collections are manually created by judges and domain experts. Unfortunately, in case of larger data sets this process takes too long and is simply not feasible. Thus, several methods were developed that allow to determine a smaller set of documents. This set is then manually judged, such that the probability of relevant documents outside this set is low. Pooling, which is one possible method, constructs the set of relevant documents by putting together the top N results extracted from a set of n systems. In TREC the value of N is 100. The documents of the pool are then manually judged by humans, which is feasible due to the reasonable amount of documents [38].

Precision and recall are quantitative evaluation techniques. Nevertheless, also qualitative techniques exist. One additional way is conducting interviews with domain experts. It is a question-answering method, where the experts are given a set of queries. In the next step, they have to define the

documents they would expect from the corresponding query. The retrieved results are then compared to the expected document set. However, queries might return a vast amount of results. Thus, usually a smaller set of documents is defined and checked within the first N results of the document set retrieved by the query [41].

The previously discussed methods evaluate the quality of the search results. Though, it is important to mention that there are many other ways of evaluating IR systems like indexing times, size of the index, stability, scalability.

2.5 Additional Methods and Techniques

In the previous sections, we talked about the basic IR concepts. In this section, we focus on further methods and techniques, which can be used to improve the search experience and enrich the search.

On possibility to enrich the search experience is to extract additional information from texts e.g. categories, metadata. We refer this concept to Information Extraction (IE).

"Information extraction is the task of finding structured information from unstructured or semi-structured text [1]."

In the field of information extraction, Natural Language Processing (NLP) methods are applied in order to extract structured information from the respective documents. NLP combines the fields computer science and linguistics by focusing on the interaction between computers and natural language [22]. In the following part of this section, we describe two important NLP concepts: Entity Extraction and Classification.

Entity Extraction: An important NLP task, which is called Named-entity Recognition, is the extraction of entities from documents. Those entities are typically metadata e.g. dates, locations, names, which can be automatically extracted and used for different search features e.g. faceted search [12].

However, NLP methods are not the only way to extract information from documents. Regular expressions are another way of extracting information from texts by applying certain rules. Regular expressions are specific characters, which describe search patterns. The name is derived from the mathematical theory they are based on. They can be used to execute certain actions when

being detected in a data set e.g. extract information, split text.[6]

Example: E-mail recognition

- Document: "The E-mail address max.mustermann@wu.ac.at"
- Regular Expression: *"\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\"*
- Description: The regular expression would identify and extract the E-mail address of the document above.

Example: Paragraph recognition

- Document: In this example, we refer to the documents in section 2.
- Regular Expression: *"(§[0-9][0-9])|(§[0-9][0-9][0-9])"*
- Description: Applying the regular expression on the documents in section 2, would detect the following paragraphs: "§20", "§587", "§28", "§37".

Classification: Another important NLP task refers to the classification of documents. Classification methods are used to identify patterns within or between documents. This information can be used to identify similar documents and derive categories for those documents e.g. labour law, environmental law. Those categories can be used to provide the user similar documents when searching in a specific domain.

The Support Vector Machine (SVM) is a popular supervised-learning model that can be used to classify data. As it is a supervised-learning model, it requires a labelled training data. This training data is composed of categories and data, in our case documents, assigned to those categories e.g. "environmental law". The documents in the categories have certain features (weights) e.g. documents containing the terms "environment", "nature" are assigned to the category "environmental law", and are represented as a vector of those features, see section 2.3. The algorithm matches the input data against the training data in order to classify the data. The classification is performed through support vectors and hyperplanes. Each category in the training set has certain support vectors. These support vectors are used to determine the optimal hyperplane. The hyperplane is the median between two support

[6] http://www.regular-expressions.info/

vectors and is used to separate data sets. The optimal hyperplane is the one with the highest margin (distance between the two support vectors) and without data between the support vectors [18].

Extract Transform Load: The Extract, Transform and Load (ETL) process is used to extract the relevant data from potentially various data sources, transform the data to the relevant schema or format and load it to the desired location. This process can be used to get the relevant data, which is then indexed by a search engine. There are various tools available ranging open-source e.g. Talend Open Studio, to commercial ones e.g. SAS ETL. Most of those tools have a graphical interface in order to design the ETL process [40].

2.6 Implementations

Relational Database Management Systems (RDBMS) have been used to manage structured and unstructured data for a long time. Relational databases are very efficient on structured data, but lack performance when dealing with large sets of unstructured data. With the rising amount of unstructured data, new solutions were developed handling those data in an efficient way. Open-source search engines emerged ranging from Application Programming Interfaces (API) e.g. Lucene, to standalone search environments e.g. ElasticSearch. But not only open-source solutions were developed, also commercial search engines e.g. Mindbreeze InSpire, HP Autonomy. were launched. The classical relational database vendors IBM, Microsoft, Oracle. reacted to those upcoming full-text search engines and started integrating full-text search features in their relational databases [3]. Below are the most important search solutions listed.

- **Relational database vendors**: IBM DB2, Microsoft SQL Server, MySQL, Oracle, PostgreSQL
- **Open-source search engines**: Lucene, Solr, ElasticSearch
- **Commercial search engines**: Google, Mindbreeze InSpire, HP Autonomy, IBM

Google, which is known for its large scale web search engine, offers the Google Search Appliance. A Search Appliance is a package of IR software and optimized hardware. But Google is not the only vendor of those Search Appliances. Other vendors like Mindbreeze, started to develop such Search

Appliances as well. Since hardware and software are offered together, the customers don't need to provide additional resources and capacity in their existing IT infrastructure.

Tab	Description
Bundesrecht	Federal law
Landesrecht	National law
Gemeinderecht	Municipal law
EU-Recht	European Union law
Judikatur	Judicature
Erlaesse	Decrees
Gesamtabfrage	Query on all collections
Uebersicht	Site-map

Table 6: RIS Tabs

3 RIS - The Austrian Legal Information System

The Austrian Legal Information Systems, in German "Rechtsinformationssystem" (RIS), is hosted by the Federal Chancellery of Austria and represents one of its longest running projects in the digital age. RIS is a collection of the Austrian legislation and jurisdiction. It was initially developed only for the public administration. Since 1998 the general public has access to the system free of charge and without any registration. Furthermore, the relevant legal information can be retrieved through all common web browsers at any time [6].

Figure 5 illustrates the main page of the RIS Web-site, which classifies the Austrian legal information in 5 categories. Table 6 describes those categories. The other parts of the Web-site contain information about RIS and links to Web-sites that provide legal information.

The Austrian Legal Information System collects a lot of information on Austrian and European law. Nevertheless, it is important to mention that RIS only provides information and the authentic version of the Federal and State law. It does not offer any legal advices [4].

3.1 Access and Design

RIS is accessed by professionals like judges, lawyers. as well as non-professionals. The Federal Chancellery provided us with some access statistics that were taken from the logs of the web server. The observation period was 1 month, from the 16th of April 2011 to the 16th of May 2011. The analysis showed that the majority of page visits occur during the standard working week from

Figure 5: Screenshot of the Austrian Legal Information System

Monday to Friday. At the weekend RIS is not used on a frequent basis. According to the statistics only eleven percent of the total visits fall on Saturday and Sunday. The peak of visits during the observation period was reached on Monday. Figure 6 illustrates an analysis of the page visits and queries on an hourly basis, accumulated over the whole observation period. It can be seen that the majority of visits and requests fall on the period from 06:00 a.m. to 5:00 p.m. Outside the regular working time from 5:00 p.m. to 06:00 a.m. only a small percentage visited the page or retrieved some information. During this 1 month around 75 million page visits were recorded. In total the web server identified 78 million visits. The difference of 3 million can be attributed to spiders and bots mostly coming from web search engines like google or yahoo. About half of the page visits, namely 43 million can be related to search queries. On average this means 1,4 million queries per day. Table 7 reflects these queries in more detail. According to the statistics 0,3 percent of the daily queries were incorrect due to users' applying an invalid query syntax.

The analysis of the statistics highlights that the page visits and user queries are mostly spread over a regular working week. Moreover, the statistics indicate that there are no high peaks of user queries that might lead to issues in terms of search performance.

Figure 7 illustrates the search mask of the Consolidated Federal Law. Every

Queries per day	Amount
Successful	1.407.600
Cached	1.058.309
Redirected	50.815
Incorrect	4.005

Table 7: RIS Statistics

search mask looks slightly different since they are all based on a different underlying database. Section 3.3 reflects how the data is stored and queried in more detail. Nevertheless, this search mask is a good representative, because it contains all common elements. Typically a user can search over all content of this data set through the field "Suchworte". The other search fields reference to metadata allowing the user to refine the search and present the content in a structured form. The user can specify a date range, select from a drop down list or enter the search terms in an open text field.

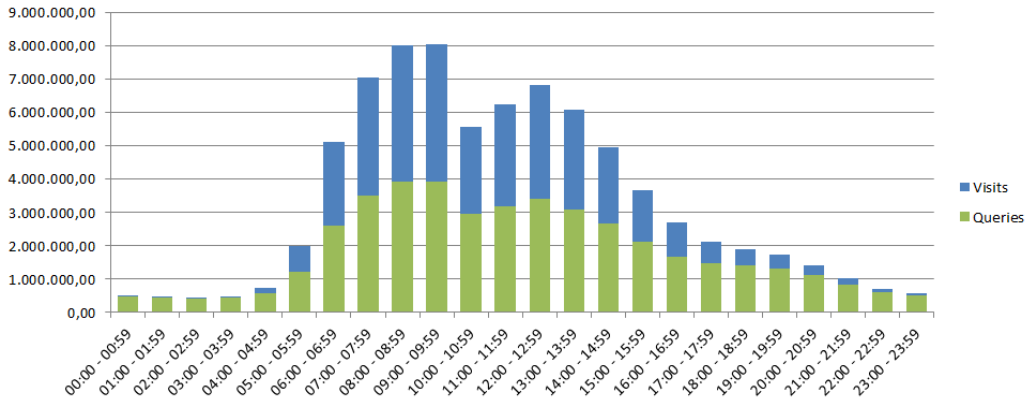

Figure 6: Web-site visits and user queries

3.2 Architecture

In this section we describe the technical architecture of the information system. Figure 8 shows the architecture that is composed of 4 basic layers.

- **Data**: RIS has 31 databases, which differ in attributes, tables, amount of documents and content. Each of the databases contains legal information focusing on a specific part of the Austrian Law.
- **Database Server**: To guarantee fail safety, two database servers exist that have redundant data stored and run on physically different ma-

Figure 7: Screenshot of the Consolidated Federal Law search mask

chines. The database server Microsoft SQL Server 2008 R2 is currently used at this layer.

- **Application Server**: The application server is responsible for load-balancing and fail-over processes.
- **Web Server**: Each of the physically distributed server clusters has two web server running.

Figure 9 describes the steps that are necessary to load the data to the relevant database. The legal content is delivered in files with various formats. The dispatcher is used to regulate the process of these incoming files. In the next step the converter transforms the metadata XML into the RIS-Import XML. Moreover, the server creates the User Data XML out of word documents or parts of the metadata XML. Additionally, the server creates Word and PDF documents, which can be retrieved by the user in the search process. Finally, the respective information is loaded to the relevant database.

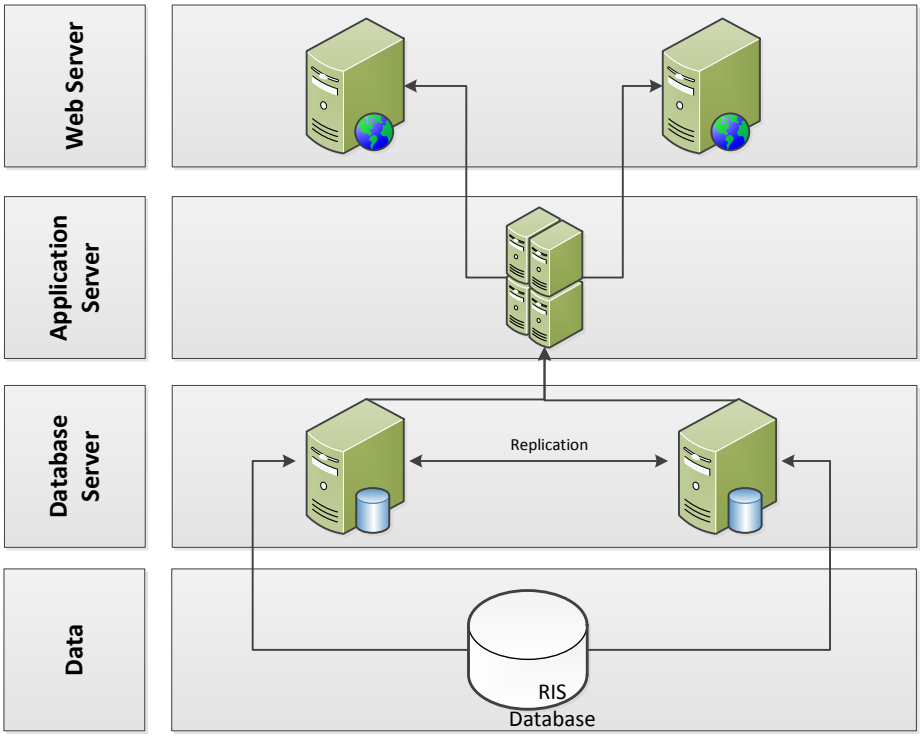

Figure 8: RIS Architecture

3.3 Data and Search

Each database contains different data structures. However, they all share a similar core structure. A detailed description of the database "Bundesnormen" can be found in appendix D.1. The names of the tables are always prefixed with the current name of the database, followed by the description of the respective table. The metadata of a document can be found in the table Dokument and the text itself is stored in the table "Nutzdaten". The other tables contain additional information to the format and history of each document. The views are queries over several columns in the table, which are relevant for a full-text search.

As described in section 3.2, RIS uses Microsoft SQL Server 2008 R2 in order to store and query data. RIS makes use of several technologies that are natively supported by Microsoft SQL Server to retrieve information. The full-text search uses the native full-text search engine. This engine is com-

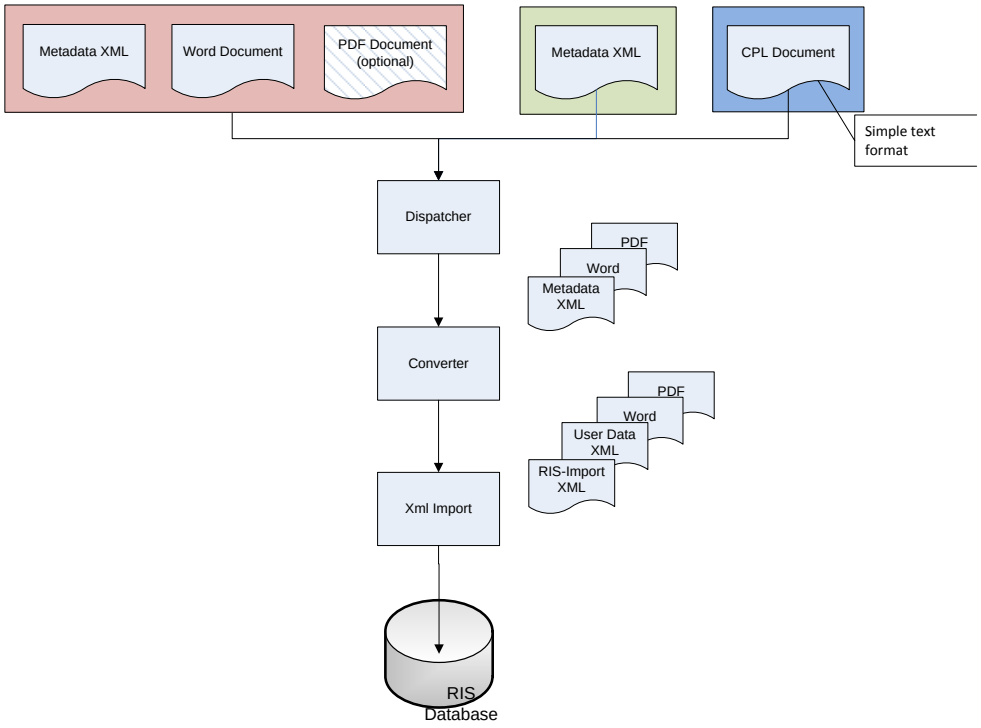

Figure 9: RIS Import

posed of the SQL Server process and the filter daemon host process. Figure 10 shows the full-text search architecture of the SQL Server. The user tables contain the actual data in the database to be full-text indexed. The crawler is responsible to trigger the indexer, which actually creates the full-text index. The full-text engine, responsible for the full-text compilation and execution of queries, also starts the filter daeomon host process. This process deals with the accessing, filtering, word breaking and stemming of the textual content [27].

As described above, SQL Server offers build in full-text search functionalities. Full-text queries make use of the predicates CONTAINS and FREETEXT and functions CONTAINSTABLE and FREETEXTTABLE. The predicates CONTAINS and FREETEXT are specified in the WHERE or HAVING clause of a SELECT statement. Both of them can be combined with other SQL predicates. The full-text predicates return a TRUE or FALSE value. The result set lists the rows that matches to the user query. Furthermore, the language to be searched for can be specified. This is relevant for stem-

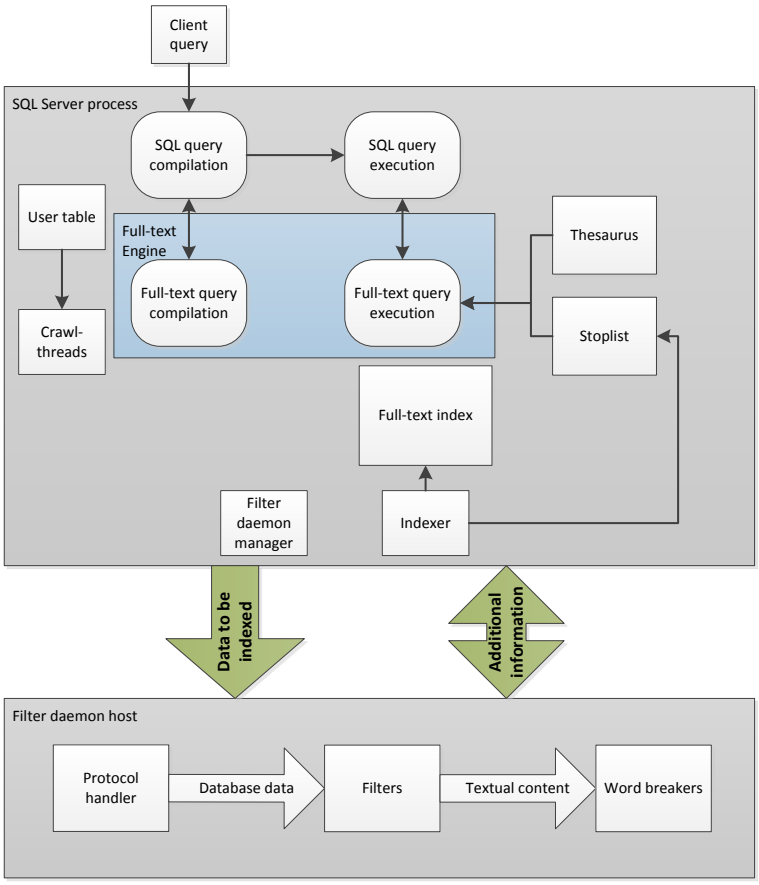

Figure 10: SQL Server Full-text Architecture (based on [27])

ming, stop words or thesaurus lookups. The predicate CONTAINS can be used for precise search on single terms or phrases as well as prefix, proximity or weighted search. It is necessary to specify at least one condition in which column the user is searching in and the search query itself. Logical operators can be used to combine search conditions. In contrast to CONTAINS, FREETEXT returns results if any term matches the query. While CONTAINS looks for exact matches of the phrase, FREETEXT breaks the phrase into individual words. CONTAINSTABLE and FREETEXTTABLE are similar to CONTAINS and FREETEXT. The difference is that they return an additional RANK column. This column is used to specify a ranking value for each row based on several statistical indicators, see listing 4. Table 8 summarizes the full-text search features and describe by which predicate they are supported [26].

Query	Description	Supported by
Simple term	One or more specific terms or phrases	CONTAINS and CONTAINSTABLE, FREETEXT and FREETEXTTABLE
Prefix term	A term starting with a specified text, also known as wildcard search	CONTAINS and CONTAINSTABLE
Generation term	Inflectional forms of a specific term, also known as stemming	FREETEXT and FREETEXTTABLE by default, CONTAINS and CONTAINSTABLE via optional INFLECTIONAL argument
Proximity term	Terms or phrases close to each other, also known as near search	CONTAINS and CONTAINSTABLE
Thesaurus	Synonyms for terms	FREETEXT and FREETEXTTABLE by default, CONTAINS and CONTAINSTABLE via optional THESAURUS argument
Weighted term	A weighting value defines the importance of a term or a phrase	CONTAINS and CONTAINSTABLE

Table 8: Microsoft SQL-Server FTS Features

The following examples are intended to explain the basic concepts of the SQL Server full-text search. All examples are based on the documents from section 2.

Listing 1: SQL CONTAINS with Boolean and Wildcard Search

```
1 SELECT *
2 FROM Table_1
3 WHERE CONTAINS(Column_1, ' "Oberste Gerichtshof" AND "Sachwalter*"
    ')
```

The query from listing 1 uses the CONTAINS predicate in combination with a boolean and wildcard search. It selects all columns from "Table_1" and carries out a full-text search on "Column_1". The full-text query would retrieve document D1, since it matches the phrase "Oberste Gerichtshof" and the terms "Sachwalterbestellung", "Sachwalters", "Sachwalter".

Listing 2: SQL CONTAINS with Proximity Search

```
SELECT *
FROM Table_1
WHERE CONTAINS(Column_1, ' Rechtsanwalt NEAR Vermittlung')
```

Listing 2 represents a full-text query using a proximity search. It applies the CONTAINS predicate and searches for the term "Rechtsanwalt" being close to the term "Vermittlung". This is the case in document D2.

Listing 3: SQL CONTAINS with Stemming

```
SELECT *
FROM Table_1
WHERE CONTAINS(Column_1, ' FORMSOF (INFLECTIONAL, Sachwalters) ')
```

Listing 3 describes a full-text query that uses the native stemming functionality of SQL Server. Microsoft natively supports a German stemmer.[7] However, the documentation does not provide any information which stemming algorithm is being used. If the German stemming algorithm from section 2.1 was applied, the above query would retrieve the terms matching to "Sachwalter" and "Sachwalters".

Listing 4: SQL CONTAINSTABLE with Term Weights

```
SELECT *
FROM Table_1
INNER JOIN CONTAINSTABLE (Column_1, ' ISABOUT (Rechtsanwalt WEIGHT
    (0.8), Sachwalter WEIGHT (0.2) ) ') AS ColumnR
ON Table_1.ID = ColumnR.[KEY]
ORDER BY ColumnR.RANK
```

The query from listing 4 applies a full-text search with the CONTAINSTABLE predicate, which additionally allows to rank the retrieved results. The ranking is based on various statistical values e.g. Occurrence, IndexRowCount.[8] In the full-text query we make use of the term weight feature, which allows determining the importance of specific terms. The weights range from

[7] https://technet.microsoft.com/en-us/library/ms142509%28v=sql.105%29.aspx

[8] https://technet.microsoft.com/en-us/library/ms142524%28v=sql.105%29.aspx

0 (lowest importance) to 1 (highest importance).[9] In our case, we give the term "Rechtsanwalt" more importance than the term "Sachwalter". Thus, document D2 would be ranked before document D1.

Listing 5: SQL FREETEXT

```
SELECT *
FROM Table_1
WHERE FREETEXT(Column_1, 'Meinungsverschiedenheit
    Kammermitgliedern')
```

Listing 5 illustrates a full-text query using the FREETEXT predicate. In contrast to the CONTAINS predicate, the FREETEXT predicate looks for each term individually. In our example, the query retrieves documents containing the term "Meinungsverschiedenheit", or "Kammermitgliedern" or both. Document D3 would be returned, since both terms appear in this document.

RIS makes also usage of the LIKE predicate on some metadata fields. However, the LIKE predicate has some drawbacks compared to full-text search. First of all, LIKE works on character patterns. This means that it scans every character in a text in order to find patterns matching to the query. Also, a LIKE query lacks performance when it is carried out against a large amount of unstructured text. Performance issues arise, because LIKE always scans the whole data whereas a full-text search uses the full-text index to retrieve information, see section 2.1. A LIKE query against millions of rows of textual content can take minutes to return, whereas a full-text query can take only seconds or less against the same data, depending on the number of rows being returned [28].

Listing 6: SQL LIKE

```
SELECT *
FROM Table_1
WHERE Column_1 LIKE 'Oberste Gerichtshof' AND Column_1 LIKE
    "Sachwalter%"
```

Listing 6 illustrates a LIKE query, which is equivalent to the query from listing 1. The LIKE operator supports all types of wildcard queries, see section

[9] https://technet.microsoft.com/en-us/library/ms142577%28v=sql.105%29.aspx

2.2. Moreover, a range for single characters can be defined e.g. [a-f]. Finally, the LIKE predicate can be combined with other SQL predicates e.g. AND, OR, NOT, BETWEEN.[10]

Listing 7 shows a RIS query using the full-text search functionalities of SQL Server. The user searched for *"Fahrstreife* nahe Radfahrstreifen*"* in the field "Suchworte". The query retrieved results matching to term "Fahrstreife" being close to term "Radfahrstreifen". Furthermore, the user applied a right wildcard search on both terms. The full-text search predicate CONTAINSTABLE in order to rank the retrieved results.

Listing 7: RIS Full-text Search

```
1 SELECT [...]
2 FROM BundesnormenDokument
3      INNER JOIN CONTAINSTABLE (BundesnormenSuchworteView,
           (Suchworte), @suchworte) AS VolltextSuchworte ON
           BundesnormenDokument.[ID] = VolltextSuchworte.[Key]
4      WHERE
5        ((BundesnormenDokument.Inkrafttretedatum is null or
             BundesnormenDokument.Inkrafttretedatum <= @vonDatum)) AND
6        ((BundesnormenDokument.Ausserkrafttretedatum is null or
             BundesnormenDokument.Ausserkrafttretedatum >= @bisDatum))
7   ) AS [Selection] ON [Complete].[ID] = [Selection].[ID] AND
       [RowNumber] BETWEEN @firstRow AND @lastRow ORDER BY [RowNumber]
8   OPTION (OPTIMIZE FOR UNKNOWN);',N'@suchworte
       nvarchar(38),@vonDatum datetime,@bisDatum datetime,@firstRow
       int,@lastRow int',@suchworte=N'"Fahrstreife*" NEAR
       "Radfahrstreifen*"',@vonDatum='2015-01-31
       00:00:00',@bisDatum='2015-01-31
       00:00:00',@firstRow=1,@lastRow=10
```

3.4 Requirements

The SQL Server has various built in full-text search features, see section 3.3. However, some full-text search features are not supported by the SQL Server. The full-text predicates enable middle and right wildcards, but do not allow left and precise wildcards. Though, this can be solved with the LIKE predicate, massive performance issues would arise on large texts. Moreover, faceted search, auto corrections and range search are not supported by the

[10] https://msdn.microsoft.com/en-us/library/ms179859%28v=sql.105%29.aspx

SQL Server. Ranges queries can be performed in the SQL Server, but not on the full-text index.

Due to those limitations, we developed together with the Federal Chancellery of Austria requirements for a full-text search in RIS. Table 9 illustrates those requirements. The requirements range from standard full-text search features as described in section 2, to specific RIS functionalities. The further evaluations of alternative search solutions are based on those requirements.

ID	Requirement	Description
1	Wildcard	Support of left, middle, right and precise wildcard search
2	Phrase Search	Exact full-text search in text and metadata fields
3	Near Search	Near search on terms and phrases with the possibility to specify the distance between those terms and phrases
4	Boolean Search	Support of logical search operators
5	Combined Search	A combination of wildcard and near search must be supported
6	Special Characters	Since RIS has many special characters, the information which characters are indexed or used for term splitting is mandatory
7	Numbers	Search for numbers with leading zeros has to be supported
8	Non-Breaking Space	Non-breaking spaces have to be treated as breaking spaces
9	Metadata and Full-text Search	Concurrent search on metadata and full-text index
10	Stemming	Support of stemming features
11	Validation of Queries	Validation of the user query before it is sent to the server
12	Current Search Functions	The current search functions, which make the retrieval of information easier for the user, must be able to implement
13	Scalability	The search solutions must be scalable
14	Performance	The query times have to be in the two digits area
15	Updates	Updating the index must be possible
16	Auto Correction	Suggestions like 'Did you mean?' in case of misspelled queries
17	Range Search	Range queries on the full-text index
18	Group of Words	Definitions of group of words that belong together and are being indexed as one term
19	Group of Words including Stop Words	Definition of group of words, which must include stop words
20	Faceted Search	Definition of categories, such that facets can be created
21	Segmentation	Formulating multiple segments in a field of a document in order to retrieve content that belongs together
22	Canonisation of Group of Words	Transforming group of words into a canonical format such that different spellings are found
23	Synonyms	Integration of a synonym list must be possible
24	Global Search	Search over different indices

Table 9: Requirements

4 Evaluation of Search Engines

As discussed in section 3.2, RIS uses the Micorosoft SQL Server 2008 R2 for indexing and searching legal documents. The SQL Server is a relational database from the Microsoft Corporation. With the rising demand of retrieving high volumes of textual content, new search solutions were developed. These search solutions are optimized for textual content and cover a wide range of vendors and technologies, see section 2.6. With the SQL Server 2005, Microsoft started to provide their customers with full-text search features.[11] Although, the SQL Server 2008 R2 offers various full-text search features, refer to section 3.3, not all requirements of a RIS full-text search can be met. Those requirements cover a wide range of full-text search features tailored to the needs of RIS platform, see section 3.4.

This chapter describes the evaluation of 2 full-text search engines. Since there are various search solutions on the market, we discuss in section 4.1 the selection of the respective solution. The implementation of the RIS full-text search requirements is described in section 4.3. If requirements are not supported out of the box by the respective search solution, we describe in detail one possible workaround. However, it is important to mention that there might be different alternative solutions to implement the requirements as well. Finally, in section 4 we evaluate both solutions.

4.1 Selection of Search Engine

There are plenty of different search engines on the market. Those search engines are used in many domains e.g. web search, enterprise search. Google and Yahoo are famous web search engines, which crawl and index millions of Web-sites around the world. Search engines can also be incorporated in Web-site to make the content of the site retrievable. Furthermore, companies use enterprise search engines in order to index and search different data sources within the company. Thus, search engines play an important role by making the different kind of unstructured data retrievable. In this thesis, we exemplary evaluate one open-source and one commercial search engine. Comparing an open-source with a commercial search engine allows us to get insights into two solutions with different strength and target customers. The selection of those two search engines is described in this section.

[11] https://msdn.microsoft.com/en-us/library/ms142571%28v=sql.90%29.aspx

4.1.1 Lucene - Open-Source Search Engine

In this section, we describe the selection of the open-source search engine. There are plenty of powerful open-source search engines available ranging from APIs to standalone search engines. The most popular ones are Lucene/-Solr and ElasticSearch.

Lucene is a scalable, high performance IR library released under the Apache Software License. It is an open-source project implemented in Java. Lucene enables enriching your applications with many full-text search features. Those features can be implemented through a powerful core API [24].

Solr is a standalone full-text search engine, which was developed in Java. It is built on top of Lucene and runs as a servlet in a container like Apache Tomcat or Jetty. Solr was originally developed to provide search functionalities for corporate Web-sites. It is an open-source project that got officially released as a project of the Apache Software Foundation in 2006. Solr provides many full-text search features, distributed search and index replication. The strength of Solr is its scalability and fault tolerance. Lucene represents the core of Solr and is responsible for indexing and searching. Solr and Lucene were merged in 2010 [37].

Elasticsearch was developed by Shay Banon and is a standalone full-text search engine developed in Java. It is an open-source project released under the Apache License. Elasticsearch is also based on Lucene, which is responsible for indexing and searching processes. Thus, it enriches Lucene similar to Solr with additional functionalities like multitenancy and scalability [13].

We decided to use Lucene for the evaluation, because a vast amount of search engines are based on it. Moreover, it is widely spread and supported by a huge community.

4.1.2 Mindbreeze - Commercial Search Engine

This section deals with the selection of the commercial search engine. Since there are various commercial search engine vendors on the market, we used the Gartner Magic Quadrant for Enterprise Search to get a better overview of the available solutions.

Gartner, Inc. is an Information Technology (IT) research and advisory company, which provides insights mainly for Chief Information Officers (CIO)

and leading IT staff. In August 2015 Gartner published the Magic Quadrant for Enterprise Search. The Magic Quadrant is a research in a specific market, which compares different competitors based on pre-defined evaluation criteria. Every vendor gets a specific score at the end of the evaluation determining his position in the graphical representation [15].

Each of the 4 quadrants has specific characteristics that are described below.

- **Leaders**: Vendors in this quadrant are able to execute their current vision and are well positioned for future challenges.
- **Visionaries**: Vendors have a good feeling for the market, but are not able to execute their vision well.
- **Niche Players**: Vendors in this quadrant are specialised in a small segment, or do not even focus on any segment. Additionally, they usually do not outperform other competitors.
- **Challengers**: Vendors might be able to dominate a large segment, but do not really understand in which direction the market moves.

Figure 11 shows the Gartner Magic Quadrant for Enterprise Search. Gartner compared 15 vendors based on multiple criteria. They contacted the company Mindbreeze and were able to initiate a cooperation. Mindbreeze is an Austrian based firm that specialized on enterprise search and information access. Mindbreeze provides their customers with the Mindbreeze InSpire, which is a Search Appliance applicable for enterprise and web search. Mindbreeze achieved a very good result in the Gartner Magic Quadrant on Enterprise Search, since it was positioned as challenger with the highest ability to execute. This means that company can react to market changes very quickly [16].

Gartner described the strength and weaknesses of each vendor. They pointed out that Mindbreeze performs well in terms of search capabilities going beyond enterprise search e.g. automatically classifying incoming documents. Moreover, Mindbreeze offers responsive design and mobile app capabilities, which cover many different devices. Gartner also identified that Mindbreeze has many connectors, which allows their customers to index various data sources. According to Gartner one caution is the rather difficult usage of relevancy models based on user behaviour. Furthermore, Mindbreeze is not well known outside Europe and has a small but improving network of partners. The number of documents indexed determine the price. Hence, the

client needs to know the amount of repositories and documents they would like to index [16].

Figure 11: Magic Quadrant on Enterprise Search (based on [16])

4.2 Technical Characteristics

In this section, we describe the basic architecture of Lucene and Mindbreeze. The technical characteristics are essential in order to understand the main differences between both solutions.

4.2.1 Lucene

As described in section 4.1.1, Lucene is a very powerful search engine that can be used to index and search data stored in different files like web pages,

Microsoft Word documents, PDF documents, etc. Lucene was originally developed by Doug Cutting. In 2001 Lucene entered the Apache Software Foundation and was part of Jakarta. Jakarta is a family of high quality open source Java products.[12] The first release under the Apache Software License was made available in June 2002. Lucene was developed due to the lack of open-source search engines. It has a strong community, which contributes to the further improvement of the software [20].

In 2005 Lucene turned from a sub project of Jakarta to a top level Apache project. While sub-projects like Solr merged into the Lucene project, other projects like Hadoop, Mahout, Nutch or Tika became own top level projects. Lucene is used in a variety of applications that range from social networks and financial services to governmental applications. Although Lucene was developed in Java, it is also available in other languages like Perl, C++, Python and Ruby [31].

Indexing and searching are the core functionalities of Lucene. Those functionalities are provided by Lucene through a powerful API. We use Lucene 4.0 as it was the most stable version and offered plenty of documentation, which was necessary to implement the RIS requirements.[13] In the next step, we downloaded the Integrated Development Environment (IDE) Eclipse Luna (Version 4.4).[14] Eclipse is a programming environment, primary used for developing Java applications.

We describe the implementation of the RIS requirements in section 4.3. Understanding the implementation of the requirements, requires a knowledge about the core architecture of Lucene. We describe the architecture of Lucene through an example covering the indexing of documents and the retrieval of information from the index. Figure 12 illustrates this architecture. The retrieval of information is divided into the indexing and the searching process. Below we describe those 2 parts and provide the code for a simple application in the appendix. The indexing is described in appendix B.1 and the searching in appendix B.2.

The indexing process starts with extracting the relevant data from different sources and file formats e.g. PDF files, relational databases. Since, Lucene is a full-text search engine, the data is mainly textual content. However, it

[12] http://jakarta.apache.org/
[13] http://archive.apache.org/dist/lucene/java/4.0.0/
[14] https://eclipse.org/downloads/

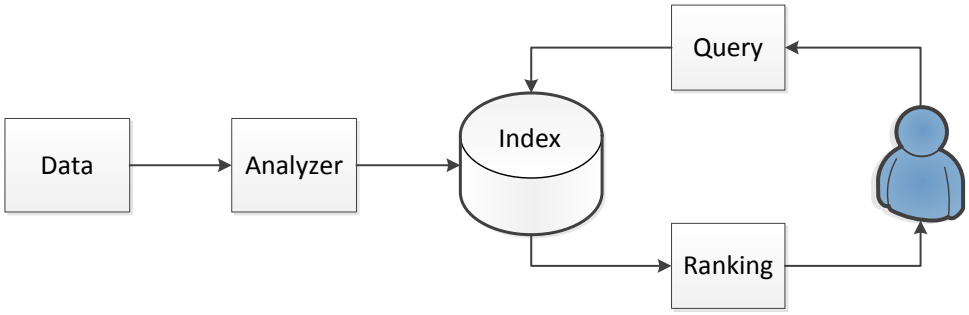

Figure 12: Lucene Architecture

is also possible to add metadata that describes the initial data. In RIS such metadata is release data of a law, norms, paragraphs. The data is then sent to the analyzer, which is responsible for preparing the data such that it can be indexed. The choice of the analyzer very much influences the indexing of data. The analyzer transforms the incoming data into a stream of tokens, see 2.1. A token stream is composed of the tokenizer and the token filter. The tokenizer creates the tokens out of the incoming data based on pre-defined rules and patterns e.g. the WhiteSpaceTokenizer creates tokens based on the white space between the incoming text.[15] The tokenizer is followed by the token filter, which applies additional rules on the tokens e.g. the StopFilter removes all stop words of the token stream.[16] Lucene provides a wide range of analyzers that are appropriate for different use cases e.g. the GermanAnalyzer for German texts, the StandardAnalyzer for English content.[17] The GermanAnalyzer is optimized for the German language. It integrates German stopwords and a German stemming filter.[18] However, the analyzers of Lucene can be customized if non of the standard ones is applicable for the respective application [20].

In the next step, the tokens of each token stream are added to Lucene documents. These documents are a collection of fields. The fields are used to

[15] https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/analysis/Tokenizer.html

[16] https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/analysis/TokenFilter.html

[17] https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/analysis/Analyzer.html

[18] https://lucene.apache.org/core/4_0_0/analyzers-common/org/apache/lucene/analysis/de/GermanAnalyzer.html

add further information to a Lucene document. In case of a legal text in RIS, the text itself and additional metadata can be stored in separate fields in the Lucene document. Figure 13 represents Lucene documents, based on the example documents in section 2. In this example each Lucene document consist of the legal text, the date and the paragraph. The original RIS documents, by contrast, contain more metadata information. This example is intended to illustrate the logical structure of a Lucene document and its fields. Each field in Lucene is composed of 3 parts: name, type and value.[19] Lucene offers different types of fields depending on the underlying data e.g. IntField, StringField. Values may be text (String), binary (byte[]), or numeric. Additionally, the following information can be specified: Analyzer, Storage, TermVector. The information can be used to determine whether to use an Analyzer, store the values of the fields, or create a vectors space for the terms in the field [20].

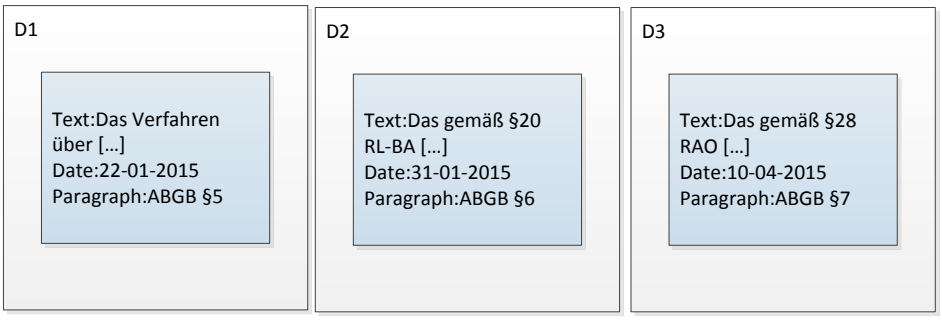

Figure 13: Lucene Document

The last step of the indexing process deals with creating and storing the index. Lucene uses an inverted index that can be stored either on disk or in memory. Whereas, the storage on disk is more appropriate for larger indexes, in memory storage is applicable for smaller indexes. There are two main reasons for this. On the one hand memory is more expensive than disk space. On the other hand, memory can be accessed much faster than a disk. However, Lucene supports possibility that exploit the advantages of both storage solutions. Therefore, Lucene can automatically load multiple documents in memory, before writing them to disk. Additionally, Lucene provides options

[19] https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/document/Field.html

File	Description
Term Dictionary	Stores the terms in a alphabetical order with the respective pointers to other files (frequency, position). Refer to section 2.1 for more information
Term Frequency	Contains the frequency of each term in the respective document
Term Position	Contains the list of positions of each term in each document
Stored Fields	This file contains a mapping of the document id's to the field where the data is actually stored
Term Vectors	This is an optional file that contains additional information about a field in Lucene e.g. position, offset, frequency.
Norms	Norms represent factors, which are relevant for scoring
Deleted Documents	This file stores markers for all documents that were deleted. The standard configuration of Lucene deletes documents from disk in case of index updates, which re-merges the documents

Table 10: Lucene Document

that allow to further configure the indexing and storing process in order to improve the performance e.g. increase the number of documents added in memory. [20].

The index is the core of every search engine. Lucene uses the concepts of an inverted index, as explained in section 2.1. It divides an index into sub indexes, which are described as segments. Each of those segments stores various files, see table 10.[20] The segments are built for the purpose of handling the indexing and updating of documents in an effective way. Since, the segments are smaller than the entire index, resources and operations of machine that is responsible for creating and storing the index can be balanced in an optimal manner.

The second part of the Lucene architecture deals with the search for information. The user formulates and executes a query. Lucene makes it possible to create own queries with its API e.g. BooleanQuery, TermQuery, PhraseQuery, RangeQuery. Those queries are restricted to a specific type of search.

[20] http://lucidworks.com/blog/2009/03/18/exploring-lucenes-indexing-code-part-2/

However, Lucene also provides a query language through the QueryParser. This query parser reads a string and maps it to a Lucene query.[21] The query parser supports all the described search features from section 2.2. The index is used to retrieve the results, which match to the user query. In section 2.3, we describe common methods that rank the retrieved results. Lucene uses the Term Frequency Inverse Document Frequency (TFIDF) Similarity to compute a score for each retrieved document.[22] TFIDF combines Boolean Models with Vector Space Models.

4.2.2 Mindbreeze

Mindbreeze is an Austrian Search Appliance vendor, which specialized on making different kind of data sources in enterprises retrievable. Mindbreeze delivers its product InSpire in form of an appliance, which includes hardware and software. The hardware is optimized on the amount of documents indexed. Mindbreeze InSpire is able to analyse structured as well as unstructured data. Apart from various full-text search features, Mindbreeze InSpire enables defining access rights and recognizing the subject of each document. The latter can be used for classifying information and describing semantic correlations between documents [29].

Mindbreeze InSpire offers a service-oriented architecture for indexing and searching. Figure 14 shows the core architecture of Mindbreeze. The InSpire Search Appliance has multiple connectors to a wide range of different data sources. The index is created by loading the data to the Index Service. In case of updates of the index, the Crawler Service of the InSpire Search Appliance is used. The Crawler Service allows to search different systems if content is changed or has to be added to the index. There are two possibilities how the Crawler Service detects changes in the source system. A source system can be any kind of system e.g. database, content management system, which stores the relevant data. One way is that the Crawler Service actively recognizes changes, which refers to the pull principle. If changes in the source system are detected, the Crawler Services forwards those changes to the Index Service, which then updates the index. The other possibility is that the source system automatically identifies the changes and forwards this information to the Crawler Service. This concept is referred to the push principle. The Filter Service comes after the Crawler Service and is used to

[21] https://lucene.apache.org/core/4_0_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html#package_description

[22] https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html

extract the textual content from the different data sources e.g. PDF documents, HTML sites. The search for information is the second part of the Mindbreeze architecture. The Query Service is used to handle the retrieval of information [30].

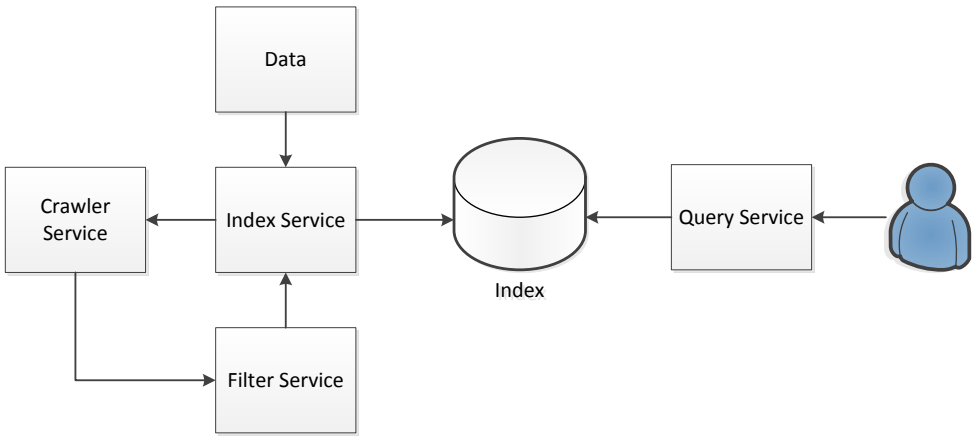

Figure 14: Mindbreeze Architecture

4.3 Implementation of Requirements

The Federal Chancellery of Austria provided us with a complete database dump of the RIS system. We restored the entire dump in Microsoft SQL Server 2014 using the SQL Server Management Studio. The backup files of the 26 RIS databases had a size of 300 GB. After we restored the back file, the consumed disk space increased to 900 GB. Thus, the full restored files including also the SQL Server full-text index were 3 times higher than the back up files. We defined together with the Federal Chancellery of Austria the databases, which are the basis for the implementation and evaluation of the RIS requirements, see table 9. We decided in favour of the 3 databases "Bundesnormen", "Justiz" and "Landesrecht Niederösterreich". In a particular system, we compare Microsoft SQL Server (current RIS solution) with Lucene and Mindbreeze, see table 11. Below are the terms we used in the comparison.

- **Yes**: It means that the requirement are implemented out of the box, with the on-board functions of the respective system.
- **No**: This means that it is not possible to implement the requirement with the respective system.

- **Custom**: We used this term to determine that the requirement could be implemented, but required some additional programming or configuration effort.

In the following part of this section we describe the procedure of indexing the 3 databases in Lucene and Mindbreeze.

Lucene: We used Microsoft SQL Server 2014 to restore the backup files the Federal Chancellery of Austria provided us with. We restored the files on the Hard Disk (HDD), connected to our local machine through USB 2.0, see table 12 for more details on the hardware. We established a connection to the Microsoft SQL Server for accessing and loading the data. We used a Java Database Connectivity (JDBC) connection, which is a standard for connecting the Java programming language to plenty of databases.[23] In the next step, we extracted the relevant data from the 3 databases with SQL scripts and indexed the data. We created an index for each database and stored them on the HDD. The code is made available in appendix B.3, where we index the database "Bundesnormen". The indexing of the other databases follows the same schema, but requires the adaptation of the database connection, the SQL script and Lucene fields accordingly. Since not all requirements could be met with the on board features of Lucene e.g. canonisation, segmentation, we created a custom analyzer, see appendix B.11. This analyzer is tailored to the German language and the RIS requirements. The index was then used for retrieving information. We set the default field in Lucene to "Suchworte", which is a view on all relevant metadata (text, date, etc.) of a RIS document.

Mindbreeze: Mindbreeze provided us with one of their InSpire Appliances. We restored the three databases "Bundesnormen", "Justiz" and "Landesrecht Niederösterreich" with the Microsoft SQL Server 2014 on a virtual machine running on a Mindbreeze server. For extracting and loading the data from the database, we used the tool Talend Open Studio. This tool is an open source product, which allows to Extract, Transform and Load (ETL) data from various sources through a graphical user interface.[24] We created 3 Talend jobs, one for each of the 3 databases. Figure 15 illustrates the Talend job for the database "Bundesnormen". The other jobs follow the same schema, but differ in terms of databases and mappings.

[23] http://www.oracle.com/technetwork/java/javase/jdbc/index.html
[24] https://www.talend.com/products/talend-open-studio

ID	Requirement	SQL Server	Lucene	Mindbreeze
1.1	Wildcard: Right	yes	yes	yes
1.2	Wildcard: Middle	custom	yes	custom
1.3	Wildcard: Left	custom	custom	yes
1.4	Wildcard: Multiple	custom	yes	yes
1.5	Wildcard: Precise	custom	yes	custom
2.0	Phrase Search	yes	yes	yes
3.0	Near Search	yes	yes	yes
4.0	Boolean Search	yes	yes	yes
5.0	Combined Search	yes	yes	yes
6.0	Special Characters	custom	yes	yes
7.0	Numbers	**no**	yes	yes
8.0	Non-Breaking Space	**no**	yes	yes
9.0	Metadata and Full-text Search	**no**	yes	yes
10.0	Stemming	yes	yes	yes
11.0	Validation of Queries	**no**	yes	yes
12.1	Current Search Functions: Word Delimiter	yes	yes	yes
12.2	Current Search Functions: Different Spelling	custom	custom	custom
13.0	Scalability	yes	yes	yes
14.0	Performance	yes	yes	yes
15.0	Updates	yes	yes	yes
16.0	Auto Correction	**no**	yes	yes
17.0	Range Search	custom	yes	custom
18.0	Group of Words	custom	custom	yes
19.0	Group of Words including Stop Words	custom	custom	yes
20.0	Faceted Search	**no**	yes	yes
21.0	Segmentation	custom	custom	yes
22.0	Canonisation of Group of Words	custom	custom	yes
23.0	Synonyms	yes	yes	yes
24.0	Global Search	yes	yes	yes

Table 11: Implemented Requirements

Specification	Details
Central Processing Unit (CPU)	3.2 GHz Dual Core
Random Access Memory (RAM)	16 GB
Operating System	Windows 7, 64 Bit
Hard Disk (HDD)	4 TB

Table 12: Lucene Hardware Details

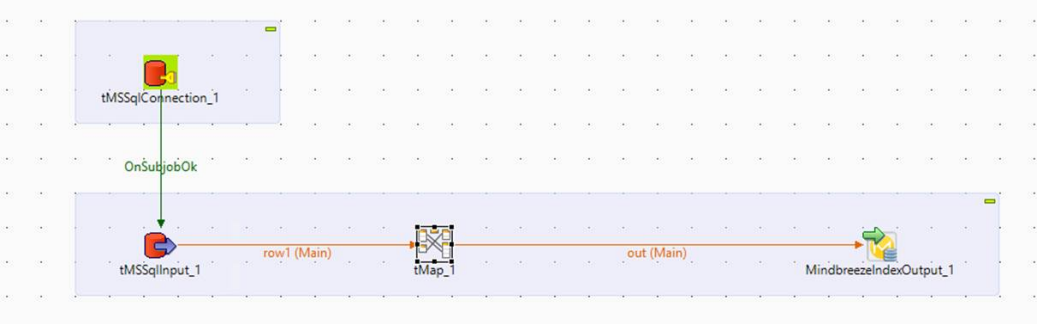

Figure 15: ETL Talend Job

Each Talend job starts with a connection to the respective database. In the next step, we applied the same SQL scripts we used for Lucene to extract the relevant RIS attributes from the SQL Server. We mapped those attributes to the Mindbreeze InSpire framework. This framework includes the mandatory attributes: key, title, content, extensions and categoryClass. Those attributes had to be filled with data. Thus, we assigned the relevant RIS attributes accordingly and created additional attributes for the remaining RIS attributes. If an assignment to a mandatory Mindbreeze attribute did not make sense, we just entered a string describing this attribute. Table 13 reflects parts of the mapping.

Since we added further attributes in the Talend jobs to the Mindbreeze framework, we had to add them to the Index Service as well. This was done by adding those additional attributes e.g. "Unterzeichnungsdatum", to the categoryDescriptor file. An extract of the file is described in listing 8. The last step included the configuration of the index according to the RIS requirements. The Mindbreeze InSpire Appliance offers a graphical web interface, see figure 16, for configuring and starting the indexing. The index was then used for retrieving information. If no field is explicitly set by the user in a query, the Mindbreeze searches by default over all fields.

RIS	Mindbreeze
ID	key
Kurztitel	title
Langtitel	title
"EXT"	extension
"RIS"	categoryClass
Paragraphnummer	Paragraphnummer
Unterzeichnungsdatum	Unterzeichnungsdatum
...	...

Table 13: Mindbreeze Mapping

Listing 8: Category Descriptor

```
<category>
   <metadatum aggregatable="true" id="Unterzeichnungsdatum"
       visible="true" selectable="true">
   <name xml:lang="de">Unterzeichnungsdatum</name>
   </metadatum>
</category>
```

The previous part of this section describes the steps for Lucene and Mindbreeze that were necessary to index the 3 databases. The following part reflects the implementation of the RIS requirements, see table 11, in more detail.

[1.1] Wildcard: Right

- Description: Right wildcard is a place holder at the end of a term that allows to search for multiple variations.
- Example: The query "Gericht*" would also retrieve results like "Gerichts", "Gerichtshof", etc.
- SQL Server: The right wildcard query is supported out of the box. The predicates CONTAINS and CONTAINSTABLE in combination with the "*" character can be used to perform this wildcard search. Query: *CONTAINS(column, 'Gericht*')*
- Lucene: The right wildcard is supported out of the box and indicted through the "*" character. The WildcardQuery or the QueryParser

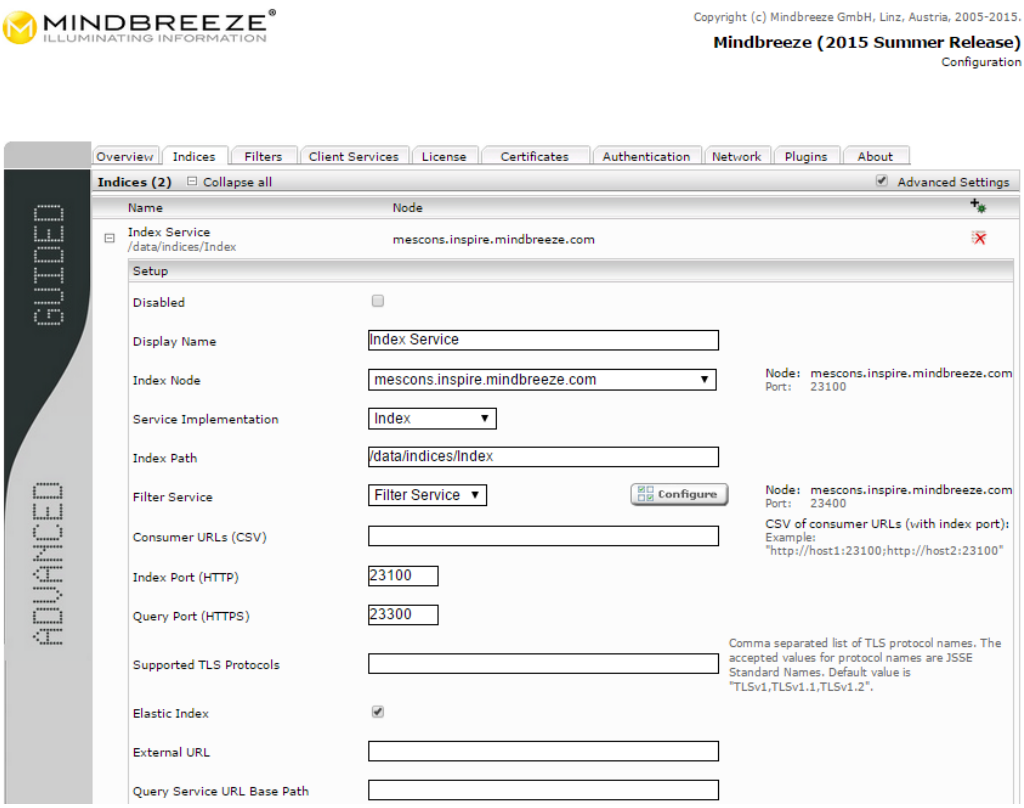

Figure 16: Mindbreeze Web Interface

can be used to perform this wildcard search.
Query: *"Gericht*"*

- Mindbreeze: The right wildcard is supported out of the box. The standard configuration applies always a right wildcard search by default. However, the "*" character can be used as well.
  Query: *"Gericht*"*

[1.2] Wildcard: Middle

- Description: Middle wildcard is a place holder in the middle of a term that allows to search for multiple variations.
- Example: The query "Rad*streifen" would also retrieve results like "Radbegegnungsstreifen", "Radfahrstreifen", etc.
- SQL Server: The middle wildcard search is not supported by the full-text services of the SQL Server. However, it could be solved with a

LIKE query, which causes performance issues on larger texts.
Query: *LIKE 'Rad%streifen'*

- Lucene: The middle wildcard is supported out of the box and indicted through the "*" character. The WildcardQuery or the QueryParser can be used to perform this wildcard search.
Query: *"Rad*streifen"*

- Mindbreeze: Being able to perform a middle wildcard search requires the implementation of a custom query transformation plugin, see appendix C.1. 37-41 reflect the respective parts in the code. We applied the middle wildcard functionality on the metadata title. If the query parser recognizes the "*" character, it will replace it by the regular expression ".*". This regular expression indicates a wildcard search and is recognized by the Query Service.
Query: *"Rad*streifen"*

[1.3] Wildcard: Left

- Description: Left wildcard is a place holder at the beginning of a term that allows to search for multiple variations.

- Example: The query "*walter" would also retrieve results like "Sachwalter", etc.

- SQL Server: The left wildcard search is not supported by the full-text services of the SQL Server. However, there are several possibilities to provide this feature. One option would be to perform a full table scan through a like query. This drawback of this option is the lack of performance. The other option would be to store the text reversed in a different field or in a meterialised view and create a full-text index on the column. In case of a left wildcard query, the query is reversed and carried out on the reversed column.
Query: *CONTAINS(revColumn, ' "retlaw*" ')*

- Lucene: The left wildcard is deactivated by default. It has to be explicitly allowed in the WildcardQuery or the QueryParser, since this wildcard search is not performant. Lucene reverses the terms in the dictionary at query time, which is resource and time consuming. The lack of performance can be solved with a custom implementation, which

we took from GitHub.[25] This filter creates a reversed token from every incoming token. It takes an incoming token, reverses this token and adds the "#" character at the end of the token. This new token is additionally stored at the current position of the original incoming token. This special character is then used to identify reversed tokens when performing a left wildcard search. On the one hand this filter doubles the amount of tokens in the index, but on the other hand it makes left wildcard queries more efficient. The better performance is achieved due to the fact that the cost intensive reversal of terms at query time isn't necessary any more.
Query: *"*walter"*

- Mindbreeze: The left wildcard is supported out of the box. The standard configuration applies always a left wildcard search by default. However, the "*" character can be used as well.
Query: *"*walter"*

[1.4] Wildcard: Multiple

- Description: Multiple wildcards represent 2 or more placeholders within a term.
- Example: The query "Arbeit*schutz*" would also retrieve results like "Arbeitsschutz", "Arbeitnehmerschutzvorkehrung", etc.
- SQL Server: The multiple wildcard is not supported by the full-text services of the SQL Server. However, it could be solved with a LIKE query, which causes performance issues on larger texts.
Query: *LIKE 'Arbeit%schutz%'*
- Lucene: The middle wildcard is supported out of the box and indicted through the "*" character. The WildcardQuery or the QueryParser can be used to perform this wildcard search.
Query: *"Arbeit*schutz*"*
- Mindbreeze: Being able to perform a middle wildcard search requires the implementation of a custom query transformation plugin, see appendix C.1. 37-41 reflect the respective parts in the code. We applied the middle wildcard functionality on the metadata title. If the query parser recognizes the "*" character, it will be replaced by the regular

[25] https://github.com/thihy/lucene/blob/master/solr/core/src/java/org/apache/solr/analysis/ReversedWildcardFilter.java

expression ".*". This regular expression indicates a wildcard search and is recognized by the Query Service.
Query: *"Arbeit*schutz*"*

[1.5] Wildcard: Precise

- Description: Precise wildcards are 1 or more placeholders that determine the amount of characters to be replaced.
- Example: The query "Bund_ _" would retrieve results like "Bundes", but NOT "Bundesrecht".
- SQL Server: The multiple wildcard is not supported by the full-text services of the SQL Server. However, it could be solved with a LIKE query, which causes performance issues on larger texts.
  Query: *LIKE 'Bund_ _'*
- Lucene: The precise wildcard is supported out of the box and indicted through the "?" character. The WildcardQuery or the QueryParser can be used to perform this wildcard search.
  Query: *"Bund??"*
- Mindbreeze: Being able to perform a precise wildcard search requires the implementation of a custom query transformation plugin, see appendix C.1. 37-41 reflect the respective parts in the code. We applied the precise wildcard functionality on the metadata title. If the query parser recognizes the "_" character, it will be replaced by the regular expression ".". This regular expression indicates a single placeholders and is recognized by the Query Service.
  Query: *"Bund_ _"*

[2.0] Phrase Search:

- Description: The phrase search is used to search for multiple terms, where the order of the terms matters.
- Example: The query "Arbeitnehmer in Wien" would only retrieve results where the terms "Arbeitnehmer", "in", "Wien" occur consecutively.
- SQL Server: Phrase queries can be formulated with the full-text search predicates of the SQL Server, see section 3.3.
  Query: *CONTAINS(column, ' "Arbeitnehmer in Wien" ')*

- Lucene: Phrase search is supported out of the box. Queries can be either formulated with the QueryParser through quotes at the beginning and at the end of the phrase, or explicitly through the PhraseQuery.
  Query: *"Arbeitnehmer in Wien"*

- Mindbreeze: Phrase search is supported out of the box. Queries can be formulated through quotes at the beginning and at the end of the phrase.
  Query: *"Arbeitnehmer in Wien"*

[3.0] Near Search:

- Description: The near search looks for terms or phrases that are close to each other.

- Example: The query "Arbeitnehmer NEAR Wien" would only retrieve results if the term "Arbeitnehmer" is close to the term "Wien".

- SQL Server: Near queries can be formulated with the CONTAINS and CONTAINSTABLE full-text predicates, see section 3.3. The distance between terms can't be specified in SQL Server 2008 R2. However, newer versions of the SQL Server allow the specification of the distance.
  Query: *CONTAINS(column, 'Arbeitnehmer NEAR Wien')*

- Lucene: Near search is supported out of the box. Queries can be either formulated with the QueryParser, or explicitly through the SpanNearQuery. The tilde character " " indicates a near search in the QueryParser. Furthermore, the distance between terms and phrases can be set.
  Query: *"Arbeitnehmer Wien" 10*

- Mindbreeze: Near search is supported out of the box. The distance between terms and phrases can be set.
  Query: *Arbeitnehmer NEAR Wien*

[4.0] Boolean Search:

- Description: The boolean search is composed of the boolean operators AND, OR, NOT.

- Example: The query "Bund AND Gericht" would only retrieve results if the term "Bund" and the term "Gericht" occur in the same document.

- SQL Server: Boolean queries can be formulated with the CONTAINS and CONTAINSTABLE full-text predicates, see section 3.3. Furthermore, combinations of boolean operators can be specified to generate nested queries.
  Query: *CONTAINS(column, 'Bund AND Gericht')*

- Lucene: Boolean search is supported out of the box. Queries can be either formulated with the QueryParser, or explicitly through the BooleanQuery. Any combinations of boolean operators are possible.
  Query: *Bund AND Gericht*

- Mindbreeze: Boolean search is supported out of the box. Combinations of boolean operators can be made.
  Query: *Bund AND Gericht*

[5.0] Combined Search:

- Description: Combined search defines the combination of the requirements 1-4 in a query.

- Example: The query "Fahr* NEAR Rad*" would retrieve results where any terms starting with "Fahr" are close to any terms starting with "Rad".

- SQL Server: Combinations of the above search features can be made. But not all requirements are supported by the full-text features of the SQL Server. Thus, performance issues may arise when combining special queries, especially in connection with the LIKE predicate.
  Query: *CONTAINS(column, ' "Fahr*" NEAR "Rad*" ')*

- Lucene: All combinations of the above requirements can be made with the full-text search functions.
  Query: *"Fahr* NEAR Rad*"*

- Mindbreeze: All combinations of the above requirements can be made with the full-text search functions.
  Query: *"Fahr* NEAR Rad*"*

[6.0] Special Characters:

- Description: Legal documents have various special characters included e.g. "§", "/", "\". Those characters are important for the retrieval of legal content.

- Example: The paragraph "§" character has to be indexed.
- SQL Server: The "§" character is not indexed by default. RIS uses a workaround to solve this problem. It converts the "§" character to a specific string e.g. "Paragraph", which is then indexed by the SQL Server. When the user applies the "§" character in his query, this character is converted to the string "Paragraph". This approach enables the search for the paragraph with the "§" character.
- Lucene: The "§" character is indexed by default. Lucene provides a description of the characteristics of every analyzer, tokenizer and filter. The description can be used to identify the text segmentation and filtering rules. The StandardTokenizer uses unicode text segmentation rules.[26] However, if special characters need to be treated in a way that is not compliant to the standard rules, Lucene enables to fully customize the token stream through various tokenizers and filters e.g. WhitespaceTokenizer (creates tokens based on the whitespace), WordDelimiterFilter (custom splitting rules can be created). If special characters are used on query side, they can be escaped if necessary. The query parser of Lucene uses the following special characters. Escaping Special Characters: + - && || ! ( ) { } [ ] ^ " ~ * ? : \
- Mindbreeze: Mindbreeze uses the standard unicode rules for word splitting. Thus, the "§" character is indexed and can be searched for.

[7.0] Numbers:

- Description: Some metadata fields e.g. "Gliederungszahl" in the database "Landesrecht Niederösterreich" contain numbers with leading zeros. Those leading zeros are essential for the retrieval of RIS documents.
- Example: A number with leading zeros would be "0000123".
- SQL Server: The leading zeros are automatically cut off by the SQL Server. Thus, the number "0000123" would be transformed to "123".
- Lucene: Leading zeros are not cut off by Lucene. We developed a little test case to prove this requirement. Listing 9 shows an HTML document, which we indexed. We used Apache Tika to extract the content between the HTML tags and indexed it with Lucene. Apache Tika is an open-source tool under the Apache Software License, which

[26] http://unicode.org/reports/tr29/

allows to extract metadata and text from many different file types [14]. In the next step, we searched in the index for "0000123" and retrieved the document.

- Mindbreeze: Tokenizer profiles can be set in the configuration process, which allow to define the type of a metadata e.g. string, numeric value. We set the tokenizer profile to nonnumeric and indexed the document from listing 9. In the next step, we searched in the index for "0000123" and retrieved the document.

Listing 9: Test Document

```
<!DOCTYPE html>
<html>
   <body>
      <p>Leading zeros 00001234</p>
      <p>Gericht wurde vertagt</p>
   </body>
</html>
```

[8.0] Non-Breaking Space:

- Description: Non-breaking spaces are special characters that prevent automatic line breaks at the position they are used. Those non-breaking spaces, however, should be recognized as breaking spaces by the search system.

- Example: In HTML non-breaking spaces are encodes with " ". A phrase in a document might contain "Gericht wurde vertagt", such that "Gericht" and "wurde" occur in the same line.

- SQL Server: Non-breaking spaces are not treated like breaking spaces. Thus, based on the example above, the query "Gericht wurde" would retrieve no results, since both terms are indexed as one term "Gerichtwurde"

- Lucene: Non-breaking spaces are treated like breaking spaces. We indexed the HTML document from listing 9 and queries for "first paragraph". The query retrieved the relevant document.

- Mindbreeze: Non-breaking spaces are treated like breaking spaces. We indexed the document from listing 9 and searched for "first paragraph". We were able to retrieve the respective document.

[9.0] Metadata and Full-text Search:

- Description: A legal RIS document is composed of the textual content and various metadata describing the document e.g. publication date, paragraph number. The search system should allow to apply a full-text search on all metadata fields.
- Example: The metadata of a RIS document consists of different types e.g. "Suchworte" (string), "Paragraphnummer" (int), "Norm" (string). The full-text search should be available over all those metadata fields.
- SQL Server: The current search system does not provide a full-text search over all metadata fields of a RIS document. The reason for this is that the full-text search features of the SQL Server do not meet all the search functionalities RIS is currently using. Thus, full-text search is partly combined with standard SQL features e.g. LIKE, BETWEEN for range queries.
- Lucene: A full-text index can be created on all RIS metadata fields, since Lucene supports various types of fields e.g. StringField, IntField.
- Mindbreeze: A full-text index can be created on all RIS metadata fields.

[10.0] Stemming:

- Description: Stemming refers to the process of reducing words to their word stem.
- Example: The term "laufen" would be reduced to "lauf" depending on the stemming algorithm,.
- SQL Server: Stemming can be activated in the SQL Server.[27] A German stemming algorithm is available.
- Lucene: Depending on the choice of the Analyzer different stemmers for various languages exist. The GermanAnalyzer contains a stemming algorithm optimized for the German language. Lucene integrated different stemming algorithms in their previous versions. Lucene 4.0, the version we used, includes a light German stemmer from the Universite de Neuchatel.[28]

[27] https://msdn.microsoft.com/en-us/library/ms142509%28v=sql.105%29.aspx
[28] http://members.unine.ch/jacques.savoy/clef/germanStemmer.txt

- Mindbreeze: The stemming feature can be added through the StemmerTransformer plugin. A basic stemming algorithm is integrated in the plugin. Moreover, Mindbreeze offers additional dictionaries with vocabularies for the most common languages e.g. German, English.

[11.0] Validation of Queries:

- Description: The search system should be able to validate queries according to their syntactical correctness before they are executed.
- Example: The query "(Gericht AND Bund" would return an error, since the closing bracket is missing.
- SQL Server: Query validations are not supported by the SQL Server. However, RIS uses a custom approach that sends an initial query to the server, without retrieving any results. If the query is valid, results are retrieved.
- Lucene: Validation of queries can be added with Regular Expressions. Furthermore, Lucene throws automatic exceptions when a non valid query was detected by the query parser.
- Mindbreeze: Queries can be validated using regular expressions. In contrast to Lucene, Mindbreeze integrates less validation services. This is done on purpose, because Mindbreeze wants to offer the user immediate results even at the risk of allowing a syntactically non valid query.

[12.1] Current Search Functions: Word Delimiter

- Description: The search system is not supposed to split words on slashes.
- Example: The phrase "ABG/§2" should be indexed as one term, and not split on the "/" character.
- SQL Server: A slash is not a word breaker in SQL Server.
- Lucene: The unicode text segmentation rules are used in Lucene. Those rules do not use the "/" character to split words.[29]
- Mindbreeze: A slash is not a word breaker in Mindbreeze, because the unicode rules are applied.

[29] http://unicode.org/reports/tr29/#Default_Word_Boundaries

[12.2] Current Search Functions: Different Spelling

- Description: The search system should allow different spellings of a search term.
- Example: The query Paragraph:"1a" and the query Paragraph:"1 a" should retrieve equivalent results.
- SQL Server: RIS solved this requirement through a custom approach.
- Lucene: We created a custom approach to solve this requirement in Lucene. Therefore, we created a new field, which merges the fields "Paragraphnummer" and "Paragraphbuchstabe". In the next step, we cut the whitespace, if existing, and executed the query. This solution allows to retrieve both spellings, since it transforms "1 a" to "1a".
- Mindbreeze: We created a custom approach, similar to the Lucene approach, to solve this requirement. We merged the fields "Paragraphnummer" and "Paragraphbuchstabe" to one field in the Talend job. In the next step, we implemented an additional solution in the QueryExpansion plugin, see appendix C.1. Part 28-38 describes the relevant code, which cuts the whitespace, if existing, and forwards the query to the Query Service.

[13.0] Scalability:

- Description: The search system has to be scalable when creating and updating index with a large amount of data.
- SQL Server: The current search system has a scalable architecture, see section 3.2.
- Lucene: We indexed the database "Bundesnormen" and recorded the times after indexing 10,000 documents. Figure 17 illustrates those times. We indexed 346,000 documents, which took around 8 minutes (470 seconds). The amount of documents is described on the x-axis, the times are indicated on the y-axis. On average Lucene indexed 725 documents per second. The figure shows an almost linear curve, which means that Lucene scales well with big amount of documents.
- Mindbreeze: The InSpire Appliance is a scalable solution. Firstly, the hardware is optimized on the amount of documents being indexed.

Secondly, further Appliances can be added, which can be combined to a cluster. This cluster distributes the index over multiple Appliances, which can be used to parallelize the creation or update of the index.

[14.0] Performance:

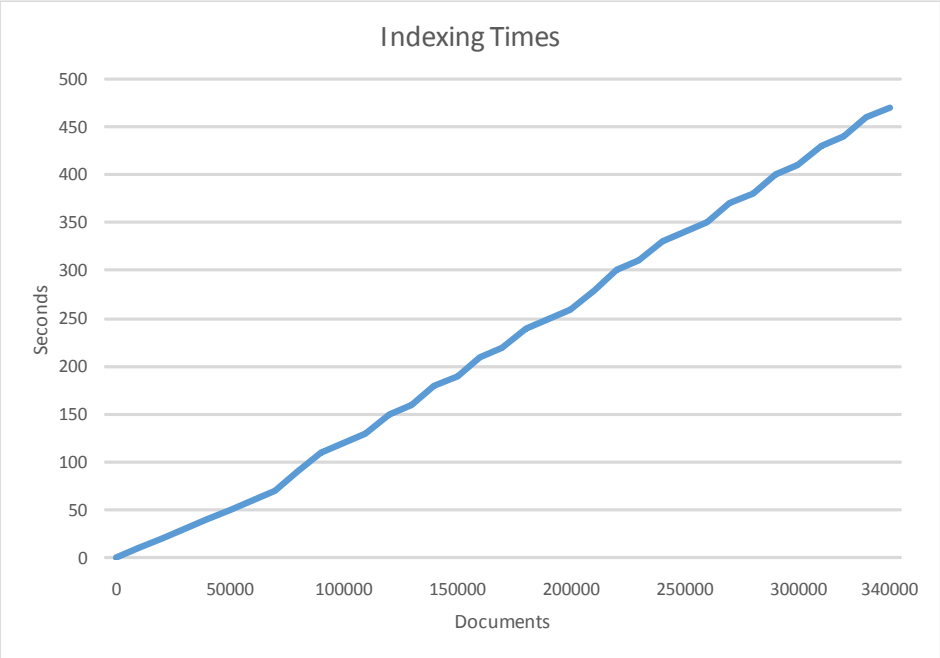

Figure 17: Indexing Times

- Description: Results should be retrieved under 100 milliseconds.
- SQL Server: The current search system is efficient, because it quickly retrieves the documents matching to a query.
- Lucene: The requirement is met by Lucene. An average query took around 91 milliseconds. For more information, see section 4.4.
- Mindbreeze: The requirement is met by Mindbreeze. An average query took around 11 milliseconds. For more information, see section 4.4.

[15.0] Updates:

- Description: Updates on the index should be possible with the pull or the push principle. In the pull principle, the search system automatically detects the documents that were added, deleted, or changed. By contrast, the push principle refers to an approach where the source system submits the search system the documents that have to be added, deleted, or changed.
- SQL Server: Updates in SQL Server are performed through the UPDATE predicate. The index is replicated on 2 separate machines. When updates are carried out, the first machine is set off-line and the index updates. When the update on the first machine is finished, the machine is set on-line and the same process is carried out on the second machine.
- Lucene: The index can be updated through the push and the pull principle. Lucene enables updates on the entire index as well as updates on single documents. The update of a specific document in the Lucene index is described in appendix B.5.
- Mindbreeze: Updates on the index are carried out through the Crawler Service. Mindbreeze supports updates with the pull or the push principle. Moreover, the entire index but also single documents can be updated.

[16.0] Auto Correction:

- Description: Auto corrections refer to the automatic detection of misspelled terms in a query. The search system offers the user alternative suggestions.
- Example: The query for "Gerucht" could produce the suggestion "Did you mean? Gericht".
- SQL Server: The auto correction is not supported by the SQL Server.
- Lucene: Lucene supports automatic suggestions of corrections. The basis of those suggestions is a dictionary. This dictionary can be created out of the index or added through an additional list of words. We implemented a dictionary on the field "Suchworte" and placed the code in appendix B.6. Including the suggestions in the search process,

requires opening the directory and selecting an appropriate string similarity method. Lucene supports various of those methods. However, we selected the JaroWinklerDistance, because it is faster and gives better results than the standard Java string distance method or the LevensteinDistance [10].

- Mindbreeze: Mindbreeze offers a solution, which displays suggestions in case of misspelled words. The alternative search terms are based on internal index statistics and analysis. This feature is called 'Did you mean?' and can be added as an additional plugin.

[17.0] Range Search:

- Description: The range search describes a search over a specific range. This range can be in a lexicographical or numerical order. Lexicographical means an alphabetical order of words e.g. "aa", "ab", "ba".

- Example: Range searches are mostly used for dates and paragraphs in RIS e.g. "§1a TO §3d".

- SQL Server: Range searches are not supported by the full-text search features of the SQL Server. However, they can be constructed with SQL range queries.
  Query: *column BETWEEN (1 AND 3) AND column BETWEEN ('a' AND 'd')*

- Lucene: Range queries on the full-text index are supported by Lucene. Range queries can be constructed through the QueryParser, or explicitly through the RangeQuery e.g. NumericRangeQuery, TermRangeQuery. We provide the code for the example in appendix B.4. The QueryParser could also be used to construct the same range query.
  Query: *paragraphnummer:"1 TO 3" AND paragraphbuchstabe:"a TO d"*

- Mindbreeze: Range search is supported by Mindbreeze out of the box. However, in the InSpire Appliance we got access to, only one connector for range queries was implemented. Since our example required a range query on two metadata fields, we had to come up with a workaround. We solved the requirement by making use of the Crawler Service of the InSpire Appliance. We modified the SQL script from the ETL job in a way, which stored the metadata in an XML structure on the server. In

the next step, we added the relevant metadata fields, "paragraphnummer" and "paragraphbuchstabe", to the Crawler Service and indexed the XML document. The data was extracted from the XML documents through XPath expressions. XPath is a syntax, which is used to select elements and attributes in XML documents.[30]
Query: *paragraph:"1a TO 3d"*

[18.0] Group of Words:

- Description: RIS has a list of words, which belong together. Thus, they have to be indexed as one token.
- Example: The term "BGBl" and "I" should be tokenized as "BGBlI".
- SQL Server: RIS uses a custom approach for this requirement.
- Lucene: The requirement is not supported by Lucene out of the box. However, we managed to solve it using an AutoPhrasingTokenFilter provided by Lucidworks on GitHub.[31] This filter takes one or multiple terms as an input. Whenever it detects those terms in a token stream, it tokenizes them as a single token.
- Mindbreeze: The requirement is supported by Mindbreeze out of the box. Entity recognition patterns can be used to define pattern rules in order to tokenize group of words. The following regular expression, gives an example how such rules can be implemented. In this case "BGBl" and "I" are grouped together.
  Regular expression: Wortgruppe =/BGBl\.\sI/.

[19.0] Group of Words including Stop Words:

- Description: This requirement is similar to the previous requirement. The difference is that the group of words include stop words.
- Example: The term "der" and the term "Bürger" should be tokenized as "der Bürger".
- SQL Server: RIS uses a custom approach for this requirement.

[30] http://www.w3schools.com/xsl/xpath_intro.asp
[31] https://github.com/LucidWorks/auto-phrase-tokenfilter/blob/master/src/main/java/com/lucidworks/analysis/AutoPhrasingTokenFilter.java

- Lucene: The difference to the previous requirement is that the filter must be set before the removal of stop words. This can be achieved through a custom analyzer.

- Mindbreeze: As in the previous requirement, we used the entity recognition service. This service can be used, since it is applied before the removal of stop words. The following regular expression, gives an example how such rules can be implemented. In this case "der" and "Bürger" are grouped together.
  Regular expression: Wortgruppe =/der \s Bürger \s/.

[20.0] Faceted Search:

- Description: In a faceted search the amount of matching documents to a query is measured for pre-defined categories.

- Example: A faceted search can be build on specific metadata fields e.g. date, paragraph.

- SQL Server: The faceted search is not supported by the SQL Server.

- Lucene: The faceted search is supported by Lucene out of the box. Implementing this full-text search feature in Lucene requires the creation of a taxonomy. A taxonomy is a classification of a content into ordered categories.[32] We build a category on the field "Gericht", see appendix B.7. Any other categories like dates, paragraphs, etc. can be selected as well.

- Mindbreeze: The faceted search is supported by Mindbreeze out of the box. Mindbreeze builds a taxonomy on every metadata field by default.

[21.0] Segmentation:

- Description: Specific fields in RIS are segmented. This means that documents are only retrieved, if the query matches to the content of the segment.

- Example: The field "Norm" is such a segmented field. As an example we assume that field "Norm" has the following two segments: "KartG 2005 §50" and "OG 2005 §53". The segments are separated through

[32] http://dictionary.reference.com/browse/taxonomy

a new line. If the user searches for "KartG §53" the document would not be retrieved, since "KartG" and "§53" don't occur in the same segment.

- SQL Server: RIS uses a custom approach for this requirement.
- Lucene: We solved the segmentation of a field in Lucene using a custom implementation, which can be found in appendix B.9. If a new line is detected, we virtually expand the distance to the following token. In combination with a SpanTermQuery, which allows to specify the distance between two terms or phrases, only terms close to each other are found. Thus, if the user searches for "KartG §53" he would not retrieve the document, since the terms are not close to each other.
- Mindbreeze: Segments in a metadata field are automatically created in Mindbreeze when they are loaded in form of an array list. Thus, we split the segments of the field "Norm" on a new line and added those segments to an array list. The respective field is then automatically handled by Index Service as a segmented field.

[22.0] Canonisation of Group of Words:

- Description: RIS uses a list of group of words that have to be indexed in a canonical form, such that different variations are retrieved.
- Example: The terms "ABGB" and "§1" should be tokenized as "ABGB §1". Additionally, the terms should be tokenized in a second variation "§1 ABGB". This allows the user to retrieve different variations of a group of words.
- SQL Server: RIS uses a custom approach for this requirement.
- Lucene: This requirement is not supported by Lucene out of the box. Consequently, we implemented a custom solution, see appendix B.10. If a specific term from a canonical list is detected in the token stream, we store the position of this token and add the token to a temporal list. If the next token is also identified in the respective list, we add the token from the temporal list to the current position. This transforms a specific set of terms to a canonical form.
- Mindbreeze: This requirement is supported by Mindbreeze out of the box. We solved it by defining entity recognition patterns based on the following rules. The regular expressions transform "§1" and "ABGB"

to a canonical form, such that different variations are retrieved.
Regular Expression 1: Paragraph=/§\d{1,4}/. Gesetz=/(ABGB)/.
Regular Expression 2: Wort=Gesetz+" "+Paragraph./
Regular Expression 3: Wort2=Paragraph+" "+Gesetz./

[23.0] Synonyms:

- Description: Synonyms are words or phrases that mean the same as other words or phrases.
- Example: The term "fast" means the same as "rapid".
- SQL Server: A thesaurus, which implies a list of synonyms, can be integrated in the full-text search.
- Lucene: Synonyms can be added to the index through the SynonymFilter. A synonym map is created and added to the SynonymFilter, which applies the synonyms on the incoming token stream, see appendix B.8. In combination with the AutoPhrasingTokenFilter, mappings of entire phrases can be built.
- Mindbreeze: A list of synonyms can be added with the SynonymTransformer plugin.

[24.0] Global Search:

- Description: A global search refers to a search over all databases.
- Example: Search over the databases "Bundesnormen" and "Justiz".
- SQL Server: RIS provides the search over all databases.
- Lucene: We created an index for each of the 3 databases. Queries on all of those indexes are supported by Lucene. The relevant indexes are read by different IndexReader, which are assigned to the MultiReader. The query is executed on the MultiReader.
- Mindbreeze: We created 1 index with Mindbreeze, which was composed of 3 sub indexes. The DetaIntegration service creates those sub indexes based on the different data sources being added. In our case, we added the 3 Talend jobs to this service.

Supported	SQL Server	Lucene	Mindbreeze
Yes	12	23	25
Custom	11	6	4
No	6	-	-

Table 14: Support of RIS Requirements

4.4 Evaluation

SQL Server, Lucene and Mindbreeze are very different search systems, each having advantages and disadvantages. The implementation of the requirements, see section 4.3, shows that Lucene and Mindbreeze can meet all requirements. By contrast, SQL Server can't meet 6 requirements at all, and has plenty of custom solutions. Table 14 shows the distribution of the requirements according to our classification.

In this section, we evaluate the search systems in order to get further insights. There are various forms how search systems can be evaluated. The range from precision and recall methods, see section 2.4, to the consideration of index size, index updates, query times, etc. We did not use the precision and recall methods, because no ground troth in terms of a reference data set was available. So we tried different approaches to evaluate the search systems, which are described below.

Index Size: First of all, we compared the size of the full-text index, see table 15. The index sizes is composed of the 3 databases "Bundesnormen", "Justiz" and "Landesrecht Niederösterreich". Table 15 compares full-text index size of Lucene and the SQL Server for each database. The size of the SQL Server full-text index was retrieved through the SQL statement "SELECT FULLTEXTCATALOGPROPERTY ('LrNoVolltext','IndexSize')", which returns information about various properties of the full-text index e.g. size of the index (IndexSize), number of indexed items (ItemCount).[33] The property "IndexSize" displays the logical size of the full-text index, which defines all fragments (parts) that can be queried. Fragments are internal tables that store the inverted index data.[34] However, the fragments of the index that are not relevant for querying are not considered e.g. marked for deletion (documents that are marked for deletion, but not yet deleted from disk yet).

[33] https://msdn.microsoft.com/en-us/library/ms190370%28v=sql.105%29.aspx

[34] https://technet.microsoft.com/en-us/library/cc280700%28v=sql.105%29.aspx

Database	SQL Server (GB)	Lucene (GB)
Bundesnormen	0,773	1,91
Justiz	1,40	3,70
LrNo	0,04	0,12

Table 15: Index Size

The size of the Lucene index was taken from the hard disk. It can be seen that the full-text index of Lucene is around 2,5 times bigger than the full-text index of the SQL Server. There are various reasons for this difference. Firstly, Lucene stores additional content, see table 10, which are used for the effective retrieval of information e.g. term vectors. Secondly, the index size of Lucene covers all segments and files, which is not the case in the logical index size of the SQL Server. Moreover, we created a Lucene field for each relevant metadata of the respective RIS database, such that a full-text search can be applied on all fields in the search mask of the RIS Web-site. This is not the case in SQL Server, since the full-text search is limited and therefore compensated through custom solutions or standard SQL features e.g. range search, LIKE predicate. The range search in the SQL Server is performed with the BETWEEN predicate on the respective metadata e.g. paragraph number. Since the range search on the full-text index is not supported by the SQL Server, the particular metadata field is not indexed. By contrast, Lucene supports the range search on the full-text index, which is why we added the field to the index. Due to those differences, it is not possible to conclusively compare the size of both indexes. We could not consider the size of the Mindbreeze index, since the information was not disclosed.

Query Times: In the second approach, we compared the query times of Lucene and Mindbreeze. The Federal Chancellery of Austria provided us with queries for the respective RIS requirements, see appendix D.2. We carried out those queries on the Lucene and Mindbreeze index and recorded the times. We only considered the query times, no network times were taken into account. The Lucene index was stored on the hard disk of our local machine, see table 12. By contrast, Mindbreeze provided us with an InSpire Appliance, where the information of the technical characteristics was not disclosed. Each of the queries was executed 3 times. We calculated the average of those values in order to get a reliable time record for each query. Table 16 shows min, mean and max statistics in order to give an indication of the distribution. It can be seen that Mindbreeze performs far better than Lucene. However, it is important to mention that this is just an indication,

Statistics	Lucene (ms)	Mindbreeze (ms)
Minimum	43	8
Maximum	320	24
Average	91	11

Table 16: Query Times

since different hardware was being used.

Overlap: In our last approach, we compared the overlap of the RIS and Lucene results through a custom application. The Federal Chancellery of Austria provided us with log files from their server. Among other information, a log contains the time stamp of the user request and the Uniform Resource Locator (URL). The URL is an unique identifier, which is used to retrieve and publish any resource on the web. This resource can be a HTML page, an image, etc. It is composed of mandatory and optional parts.[35] Table 17 reflects the structure of a RIS URL, which retrieves results matching to a query. The URL is composed of the protocol, the domain name, the path to the result file on the server and the query. The query itself includes the respective database, the metadata and the user entries. Due to the fact that some queries retrieve plenty of results, RIS uses paging in order to reduce the amount of results displayed on each HTML page. The results are limited to 100 documents per page. "Position=1" at the end of the URL refers to the first result page. Additional pages are added in steps of 100. So the second page would have the number 101, the third page 201, and so forth. Since we wanted to compare all results from Lucene and RIS, we had to take the paging into account in our application. In the next step, we extracted the queries from the log files. We developed a script that parses all log files and extracts the search terms from the field "Suchworte" of the databases "Justiz" and "Bundesnormen". We were able to extract 108,390 queries from the "Justiz" database and 65,861 queries from the "Bundesnormen" database. We stored the queries in 2 separate text files and executed them on the respective Lucene index. Unfortunately, this approach produced many errors due to encoding problems and different search syntax. RIS uses a custom search syntax, which is then translated to the SQL Server syntax e.g. RIS near query *Rad nahe Zug*, would be in Lucene *"Rad Zug"~*. As a consequence, we randomly took a set of 100 queries from the "Justiz" database, which we translated to the Lucene query parser syntax. At the end, we had

[35] https://developer.mozilla.org/en-US/Learn/Understanding_URLs

Type	Example
Protocol	http:// or https://
Domain	www.ris.bka.gv.at/
Path	Ergebnis.wke
Query	?Abfrage=Justiz&Gericht= &Rechtssatznummer=&Rechtssatz= &Fundstelle=&AenderungenSeit=Undefined& SucheNachRechtssatz=True&SucheNachText= False&GZ=&VonDatum=&BisDatum=30.10.2015& Norm=&ImRisSeit=Undefined&ResultPageSize= 100&Suchworte=&Position=1

Table 17: RIS URL

one file composed of the original RIS queries and one file with the equivalent queries in the Lucene syntax.

Those queries were the input for our application, which compares the result set of RIS with the result set of Lucene. The application reads in each of the RIS queries and each of the equivalent Lucene queries. Retrieving the result set of Lucene, we simply carried out the query on the respective index stored on our machine. Retrieving the result set of RIS, required to insert the search term in the metadata "Suchworte" of the RIS URL and request the result page from the RIS server. Paging was integrated in our application, because we wanted to retrieve all results. In the next step we parsed through the result pages with the JSOUP HTML parser and extracted the document ID of each RIS document.[36] Finally, we we added the RIS and Lucene results to 2 separate lists and analysed the overlap based on the document ID.

Table 18 illustrates the key findings. From the 100 queries, 95 queries retrieved results in Lucene, whereas 82 queries retrieved results in RIS. Furthermore, we identified that 78 queries had an overlap of results, which means that there was at least one document that appeared in both result sets. We took a closer look at the overlap and identified that in only 7 queries the results perfectly matched. In the remaining 71 queries, RIS produced in 12 queries more results, whereas Lucene generated in 59 queries more results. There are various reasons for the difference in the result sets. Firstly, RIS uses plenty of custom implementations e.g. segmentation, group of words, etc., which results in different indexing characteristics. Secondly, we applied

[36] http://jsoup.org/

Queries	Lucene	RIS
Total queries	100	100
Queries with results	95	82
Queries with common results	78	

Table 18: Result Overlap

the German specific features in Lucene e.g. German stemming, German stop word removal. An example for the second argument is the query "pistenraupe" that retrieved 15 results in RIS and 21 results in Lucene. We took a look at the result sets of both queries and figured out that the RIS results are a sub set of the Lucene results. So, Lucene retrieved 6 more documents, which we further analysed. We figured out that in each of those documents the plural form "pistenraupen" appeared, but not the singular form "pistenraupe". Thus, due to the stemming feature, which reduces terms to their root forms, Lucene retrieved 6 additional documents.

5 Accessibility and Enrichment of Search

RIS is an information system that is available to all people. However, it is mainly used by professionals, as described in section 3.1. For non professional users it is challenging to retrieve the desired information within a reasonable amount of time. As a consequence, we discuss in this chapter possibilities that allow to improve the user experience through the integration of external sources and additional search concepts.

5.1 Integration of External Sources

The integration of external sources is a good way to enrich the search with additional information. We took a closer look at some options, which are illustrated in figure 18. The arcs define the information flow and are labelled with a number. This number is used to reference to the concepts behind the arcs, which are described below.

In this thesis, we define Web-sites as external sources. We selected the following Web-sites, since they offer legal content that relates to documents in RIS.

- **HELP**: HELP is a portal of the Austrian Federal Administration. The information platform was launched in 1997 offering citizens a vast amount of administrative content.[37]
- **Wikipedia**: Wikipedia is an Internet encyclopedia, that is freely available and covers a huge amount of different content.[38]

[1] In-Links: Wikipedia and HELP provide among other content plenty of articles talking about legal content. This legal content partly relates to the Austrian law and references in some cases to concrete RIS documents through an URL. Consequently, we analysed the URLs and amount of URLs that reference to RIS documents. In our analysis, we considered URLs that contain the domain name "www.ris.bka.gv.at".

At first, we analysed the Wikipedia links that reference to RIS documents. We identified 2 possibilities to get those links. The first option was to crawl the whole German Wikipedia and extract the relevant links from the HTML

[37] https://www.help.gv.at/Portal.Node/hlpd/public
[38] https://de.wikipedia.org/wiki/Wikipedia:Hauptseite

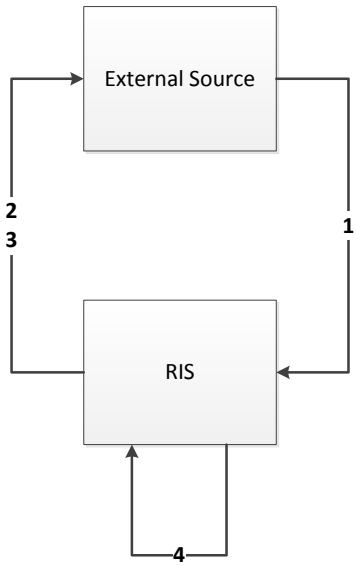

Figure 18: Integration of External Sources

documents. Since there are plenty of Wikipedia pages, we went for the second option. In this option we made use of MediaWiki, which is a free server-based software for WikiMedia content. WikiMedia relates to a group of projects, including Wikipedia, Wiktionary, Wikiquote, etc. MediaWiki processes and displays WikiMedia data that is stored in a MySQL database. Whenever a user edits a page, Mediawiki recognizes it and writes it to the database.[39] WikiMedia provides different dumps of those databases as download on their Web-site. We downloaded the external links and page SQL dump of the German Wikipedia.[40] We restored the dumps with the open source database MySQL.[41] The external links table stores and tracks all external links. It contains the following attributes: "el_id", "el_from", "el_to" and "el_index". Since URLs are not sufficient enough, we also include the page database. This database can be considered as the core of Wikipedia, containing metadata about WikiMedia pages. Listing 10 shows the SQL query retrieving the external RIS links and the title of the respective pages. We joined the two tables based on the page identifier and selected the links containing the RIS domain name. Since the attributes "el_to" and "page_title" were stored in a Binary Large Object (BLOB), we had to cast them to UTF-8, a character

[39] https://www.mediawiki.org/wiki/Manual:What_is_MediaWiki%3F
[40] https://dumps.wikimedia.org/dewiki/20150826/
[41] https://www.mysql.de/

Link	HELP	Wikipedia
In-links	1,448	5,825
Direct In-links	1,173	5,126

Table 19: In-Links

encoding for all common characters. BLOB is a data type that stores a variable amount of data in a binary format.[42] Listing 10 shows the SQL query that we used to extract the relevant RIS links.

Listing 10: MediaWiki SQL Query

```
1 SELECT el_from, cast(el_to as char(1000) character set utf8), cast(
      p.page_title as char(1000) character set utf8)
2 FROM delinks.externallinks e INNER JOIN page.page p ON e.el_from = p
      .page_id
3 WHERE e.el_to LIKE '%ris.bka.gv.at%';
```

We stored the results from the SQL query in a CSV file and prefixed every name with the Wikipedia domain name "https://de.wikipedia.org/wiki/" in order to get the URL of the WikiMedia page. In total, we identified 5,825 RIS links. However, not all of those links were relevant for us, because not all of them referenced to concrete RIS documents. However, around 88 % of the links can be referenced to concrete RIS documents, see table 19.

Extracting the RIS links from the HELP platform required crawling all HTTP pages. To do so, we used the web crawler crawler4j. This crawler is an open source web crawler implemented in Java. We decided in favour to this crawler, because it offers a simple interface and perfectly served our purpose for crawling web pages.[43] We were able to download 37,479 HELP pages. In the next step we used the JSOUP parser to extract the RIS links, title and the HELP links from the HTML pages. We stored the information in a CSV file, which we further analysed. We found out that there are in total 1,448 RIS links. 81 % of those links can be related to concrete RIS documents, see table 19.

The analysis pointed out that there are around 6,000 links that can be related to concrete RIS documents, see table 19. Those concrete links, which we describe as direct in-links, can be related to RIS documents based on specific metadata e.g. "Dokumentnummer" or "Gesetzesnummer". Table 20

[42] https://dev.mysql.com/doc/refman/5.7/en/blob.html
[43] https://github.com/yasserg/crawler4j

Web-site	Direct In-link
HELP: "Hacklerregelung"	https://www.ris.bka.gv.at/GeltendeFassung.wxe?Abfrage=Bundesnormen&Gesetzesnummer=10008422
Wikipedia: "Hacklerregelung"	https://www.ris.bka.gv.at/Dokument.wxe?Abfrage=Justiz&Dokumentnummer=JJR_20100727_OGH0002_010OBS00103_10K0000_001

Table 20: Direct In-link

illustrates an example of a direct in-link. We searched for the term "Hacklerregelung" in Wikipedia and HELP. We retrieved documents that contained links to concrete RIS documents. The direct in-links as well as the corresponding link and content of the source article in Wikipedia or HELP can be assigned to the relevant documents in RIS. This information can be added to additional metadata fields and provided to the user at query time.

[2] Search on HELP Content: Another option to improve the user experience is to provide the user in RIS a search on HELP content. We considered HELP and not Wikipedia, because the Federal Chancellery of Austria mentioned HELP as a source that provides valid content about the Austrian law. We implemented the HELP Search in Lucene and Mindbreeze. Mindbreeze already crawled and indexed HELP for another project. Thus, queries on the HELP index could be performed without any configuration.

In Lucene we had to crawl and index HELP, which required the development of a custom application. We crawled the HELP pages with the crawler crawler4j. In the next step, we used Apache Tika to extract the relevant content e.g. text, URLs, from the HTML pages. Apache Tika is an open source tool, which allows to extract metadata as well as text from a variety of file types.[44] We created a Lucene index on the content, we extracted with Apache Tika. The index was composed of 2 fields. The first field was filled with the title of each HTML page. For the second field we automatically extracted the text between the HTML tags. This index could be used to provide the user with additional information, since HELP provides plenty of summaries and descriptions about legal concepts.

[44] https://tika.apache.org/

Results	Queries
1 or more results	11,968
No results	3,115

Table 21: Second Search in HELP

[3] Second Search in HELP: This option assumes that the user already executed a query. Refining the search, RIS could provide the user the possibility to execute a second search on metadata e.g. title, of the respective result document in the HELP index. We implemented this option in Lucene and Mindbreeze. However, we also analysed the feasibility of this option, since it could have happened that hardly any results are retrieved from a second search on the HELP index. Thus, we developed an application that extracts the title of the database "Bundesnormen" and executes a query for each title on the HELP index. We were able to extract 14,960 different queries from the database. We identified that 80 % of those queries retrieved at least one result, see table 21. Thus, the option is feasible due to the high rate of retrieved results.

5.2 Further Solutions

Whereas, section 5.1 focuses on possibilities of integrating additional external sources, this section discusses a variety of other possibilities to enrich the search.

[4] Second Search in RIS: Figure 18 integrates also another option, which deals with the execution of a second query on an already existing RIS metadata field. We implemented this functionality in Lucene and Mindbreeze. In both cases, the user already executed a query and retrieved results. Refining the search, the user could now execute a second search on a metadata field e.g. paragraph number, of a specific document in the result set.

User Interface: Mindbreeze offers an editor and pre-defined HTML integration services, that allow to integrate Mindbreeze search features in already existing Web-sites. Consequently, we built a user interface that covers the previously discussed options. We downloaded the HTML Web-site of the search mask "Bundesrecht konsolidiert", see figure 7, and the result page of this search mask. We integrated the Mindbreeze search in this HTML pages

and executed them on a local Web-server. Figure 19 illustrates the user interface of the result page. In this case, the user searched for "Radfahrstreifen" in the field "Suchworte". We designed the result page similar to the current RIS result page and added new features. When the user clicks on the HELP symbol of the first document, the query "Bodenmarkierungsverordnung" is executed on the HELP index. However, when the user clicks on the paragraph symbol of the first document, a second search with "§20" is executed. Moreover, we integrated the faceted search on "Datum" and "Rechtsgrundlage", such that the user can filter the search. In this case, the 44 results of the query are distributed over the categories 2013, 2012, 2011 and 1900. The value next to each year indicates the amount of documents in the respective year e.g. the year 2013 contains 5 documents matching to the user query.

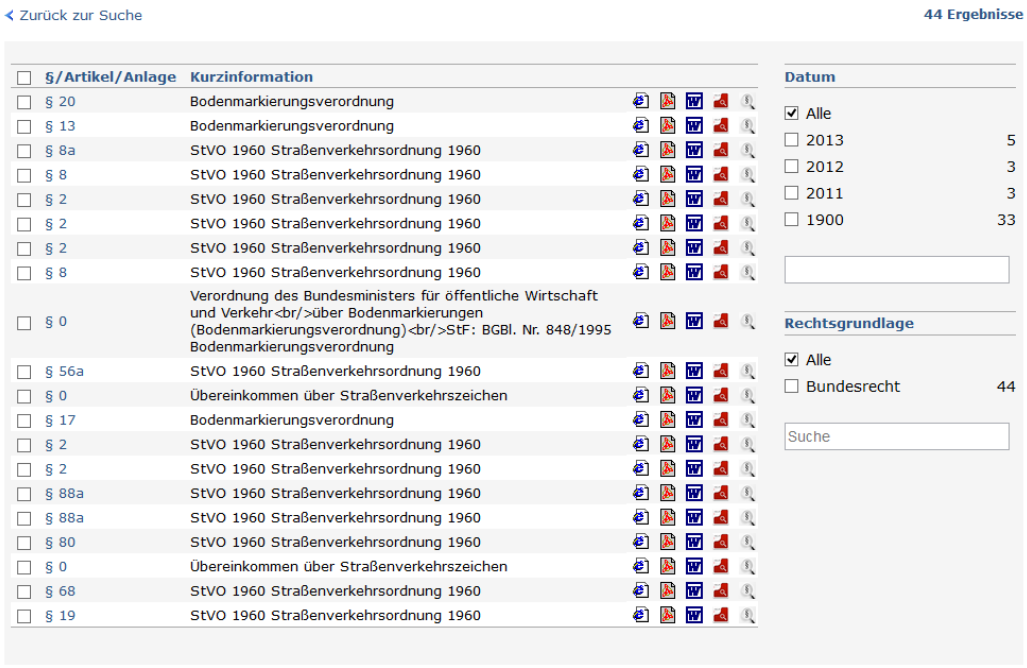

Figure 19: User Interface

Entity Extraction: The extraction of entities is another possibility to enrich the search. The entities e.g. date, name, location, can be used for various scenarios. These scenarios range from using the entities for categories in a faceted search to enriching the entities with additional information e.g. links to other Web-sites. We discuss some scenarios below and provide possible solutions on how to integrate them.

There are plenty of ways in order to extract entities out of textual content. One way is to use regular expressions for this task, see section 2.5. Mindbreeze offers this service through the Entity Recognition Parameter. Entities are automatically generated by applying pattern rules based on regular expressions. The extracted entities reference to metadata that can be used for a faceted search.

Lucene does not offer this service out of the box. However, it can be implemented in Lucene with a custom solution. The PatternTokenizer can be used to define regular expression that are applied on the textual input.[45] This tokenizer takes the arguments pattern and group. Pattern refers to the generation of regular expressions. Group describes how the tokens are being generated. The group with the value -1 splits the tokens based on the regular expression. By contrast, the group with the value 0 selects the tokens matching to the regular expression. Since we want to extract the matching tokens, we used the 0 group. We created a custom analyzer with the PatternTokenizer that extracts dates from textual content, see appendix B.12. We applied the tokenizer on a specific field in Lucene, which can be used to create categories for the faceted search, see appendix B.7. Since we did not want to apply the custom analyzer to all other fields, we had to integrate the PerFieldAnalyzerWrapper.[46]

Extracting entities through regular expressions is one option. Still, there are more possibilities that can be applied e.g. through the use of Natural Language Processing (NLP) tools and methods. Named Entity Extraction, which is such a NLP task, uses techniques that allow to automatically extract entities out of textual content. Lucene can be combined with different NLP tools. OpenNLP is a machine learning library for the processing of unstructured natural language text. It supports the NLP task described in section 2.5.[47] The main disadvantage of OpenNLP is that it can't collaborate with Lucene out of the box. Thus, NLP4L was developed to tackle this collaboration issue. NLP4L uses NLP methods that can be applied on the Lucene index.[48]

[45] https://lucene.apache.org/core/4_0_0/analyzers-common/org/apache/lucene/analysis/pattern/PatternTokenizer.html

[46] https://lucene.apache.org/core/4_0_0/analyzers-common/org/apache/lucene/analysis/miscellaneous/PerFieldAnalyzerWrapper.html

[47] http://opennlp.apache.org/

[48] https://github.com/NLP4L/nlp4l

Classification: The classification refers to the automatic generation of categories e.g. criminal law, procedural law, based on entire documents. These categories can be used to provide the user similar cases and documents depending on the content he is looking for.

Mindbreeze offers such classification services out of the box through the Support Vector Machine (SVM) concept, see section 2.5. First of all, a training set is created. This training set is composed of pre-defined categories and documents relating to the respective categories. Each document contains specific features and is represented as a vector of those features. The features are weighted according to the number of occurrences of the feature in the document, the existence or non existence of the feature in the document and the probability the feature occurs within a certain category. The input data is matched against the training data and classified through linear classifiers.

Lucene offers some classification techniques starting from version 4.2. It implements Naive Bayes and k-NN classification algorithms. However, Lucene can be combined with other solutions like Apache Mahout as well.[49] Apache Mohout provides an environment that enables integrating machine learning functionalities in applications. It offers a vast amount of algorithms ranging from filtering to classification and clustering of data.[50]

Reference Reading: Many research was done on classification and entity extraction. However, juridical documents are usually more complex in terms of phrase structure and terminologies being used. This makes it more difficult for NLP and classification tasks to produce reliable results. However, a scientific work from the University of Evora deals with the information extraction of legal documents [33]. In this work, the researchers applied linguistic information and machine learning techniques. Their approach considered document classification for describing legal concepts based on SVM and Named Entity Extraction through a natural language parser. The legal documents were retrieved from the EUR-Lex Web-site. SVM was chosen because it is computationally efficient and very robust in case of larger data sets. SVM light, an implementation of SVM in the programming language C,

[49] http://soleami.com/blog/comparing-document-classification-functions-of-lucene-and-mahout.html

[50] http://mahout.apache.org/

was selected for the classification task.[51] Instead of using statistical machine learning approaches, the researchers used linguistic approaches. PALAVRAS, a natural language parser, developed by the Institute of Language and Communications of the University of Southern Denmark was used for this task.[52]

The experiment was carried out on a data set covering International Agreements from EUR-Lex.[53] The downloaded data was composed of 2,714 agreements from 1953 to 2008 in four different languages (English, German, Italian and Portuguese). An EUR-lex documents is described by the following metadata: title, reference, dates, classifications and miscellaneous information. There are three different types of classifications: EUROVOC descriptor, subject matter and directory code.

The first approach was to classify the documents. The researchers selected the directory code in order to assign the legal documents to the relevant categories. Those categories were used to train the SVM algorithm. The analysis of the experiment showed that the German and the English language had similar results with a precision of 90 % and a recall of 80 %. The results were slightly better than those of the Italian and Portuguese language. However, the experiment demonstrated a high amount of correctly classified documents over all languages [33].

In the second approach, the researchers carried out an experiment on the extraction of entities of an English data set through a natural language parser. They extracted the categories location names, organization names, dates, references to documents and document articles. Personal names were not extracted since they hardly showed any convenient result. The analysis of the results showed a high number of references to other documents and articles, on average 28 entities per document. This information could be used to demonstrate a chain of legislative cases of a specific legal document. The categories organizations and countries showed also high numbers of entities per document. However, taking the precision into account only 35 % of the organizations and references were correct. In terms of the organizations this could be caused by natural language parser itself, since all non assigned entities were transferred to the organizations category. The high error rate of references could be explained by the relatively complex structure of sentences in legal documents [33].

[51] http://svmlight.joachims.org/
[52] http://beta.visl.sdu.dk/constraint_grammar.html
[53] http://eur-lex.europa.eu/homepage.html

6 Conclusion

In this chapter, we summarize the key finding of the present thesis and provide suggestions for future work. We evaluated the following search systems: SQL Server, Lucene and Mindbreeze InSpire. The SQL Server, a relational database of the vendor Microsoft, is the current search system of RIS. Whereas, Lucene is an open-source full-text search engine, Mindbreeze is an Austrian based company that offers the InSpire Search Appliance, a combination of software and optimized hardware.

The evaluation of the search systems was based on the requirements that were provided by the Federal Chancellery of Austria, and additional indicators e.g. query times, index size. The requirements range from standard full-text search features, see chapter 2, to RIS specific features. The implementation and evaluation of the requirements produced valuable information. On the one hand, it demonstrated the limits of the SQL Server, the current search system of RIS. Despite the integrated full-text search features of the SQL Server, only 41 % of the requirements could be met. Furthermore, in 38 % of the requirements custom implementations were required and even 20 % of the requirements could not be met at all. By contrast, Lucene and Mindbreeze could meet all requirements and only a few requirements had to be implemented through custom solutions. Furthermore, the evaluation of Lucene and Mindbreeze according to the specific indicators, pointed out that both solutions are very scalable and efficient in the retrieval of content. Finally, we identified that the search systems are very different and characterised by strength and weaknesses.

Since Lucene is an open-source project under the Apache Software License, the entire source code is available and can be adapted accordingly. Furthermore, Lucene provides plenty of documentation and has a huge community, which constantly contributes to the improvement of the search engine. However, Lucene has also some drawbacks. Lucene is not a standalone solution. Thus, it requires programming effort to create a search solution. Moreover, the Lucene developers don't undertake any guarantees or contractual obligations for issues arising with the use of the Lucene code. Finally, migrating from an older version to a newer version in Lucene requires to adapt the code according to the changes, which might cause issues.

On the contrary, Mindbreeze is a commercial software vendor that provides a fully functional search environment through the InSpire Search Appliance. Configuring and implementing Mindbreeze can be done in an efficient and

fast way through a web interface. Furthermore, Mindbreeze has plenty of different connectors to various sources and offers a variety of additional plugins e.g. stemming, synonyms, that can be easily added. Also, Mindbreeze provides plenty of additional features e.g. permission services, app.telemetry. Mindbreeze implemented a permission and access right system, which enables limiting the accessibility to specific content based on pre-defined user groups. App.telemetry is a powerful framework that can be used for Application-Service-Level-Management and Resource-Optimization. It collects and analyses response and processing times of the InSpire Search Appliances. The service provides information along the whole query process. However, Mindbreeze has also some weaknesses. The licensing is based on the amount of documents and not transparent above a certain amount of documents. This price model requires to know the amount of documents in advance and doesn't consider the actual size of documents, since only the amount of documents are relevant. Finally, if features and functionalities are desired that go beyond the standard implementation of the InSpire Appliance, cost-intensive individual solutions are necessary.

In the last part of the thesis, we discussed various concepts according to the enrichment of the search in order to improve the search experience of the user. Whereas, some of those concepts are feasible e.g. integration of external sources, other concepts proved to be not useful e.g. Natural Language Processing (NLP) tasks.

Finally, there are other platforms and projects similar to RIS. The OpenLaws platform is a project supported by the European Union, which helps to find legal information more easily as well as organize and share it with others.[54] The objective of this project is to makes legal information more accessible through a network of legislation, case law, literature and legal experts. Since OpenLaws is an open data platform, it could be integrated in RIS.

[54] http://www.openlaws.eu/

References

[1] C.C. Aggarwal and C.X. Zhai. *Mining Text Data*. Springer, 2012.

[2] Mark Aronoff and Kirsten Fudeman. *What is Morphology?* Blackwell, 2007.

[3] Ahmet Arslan and Ozgur Yilmazel. *Quality Benchmarking Relational Databases and Lucene in the TREC4 Adhoc Task Environment*. Proceedings of the International Multiconference on Computer Science and Information Technology, 2010.

[4] Barotanyi, Behr, Eibl, Freitter, Gottwald, Herwig, Havranek, Karning, Klauser, Kustor, Kraut, Ledinger, Leitold, Medimorec, Niedermueller, Pirker, Posch, Regenspurger, Reichstaedter, Rupp, Scheidbach, Tauber, Vock, and Wagner-Leimbach. *Behoerden im Netz. Das oesterreichische E-Government ABC*. Kny and Partner, 2014.

[5] Chris Borrelli. *IEEE 802.3 Cyclic Redundancy Check*. 2001.

[6] Bundeskanzleramt. Rechtsinformationssystem. https://www.ris.bka.gv.at, October 2015.

[7] Vannevar Bush. *As We May Think*. Atlantic Monthly, 1945.

[8] Jörg Caumanns. *A fast and simple stemming algorithm for German words*. 1999.

[9] Paice C.D. *Method for Evaluation of Stemming Algorithms Based on Error Counting*. Journal of the American Society for Information Science, 1996.

[10] Peter Christen. *A Comparison of Personal Name Matching: Techniques and Practical Issues*. Manning Publications Co., 2006.

[11] C.W. Cleverdon. *The Cranfield tests on index language devices*. Aslib Proceedings, 1967.

[12] Ronan Collobert, Jason Weston an Leon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. *Natural Language Processing (Almost) from Scratch*. Journal of Machine Learning Research, 2010.

[13] ElasticSearch. https://lucene.apache.org/core/, October 2015.

[14] Apache Software Foundation. Apache tika. https://tika.apache.org/, October 2012.

[15] Gartner. Gartner magic quadrant. http://www.gartner.com/technology/research/methodologies/research_mq.jsp, September 2015.

[16] Gartner. Magic quadrant for enterprise search. http://www.gartner.com/technology/reprints.do?id=1-2KSMU7V&ct=150806&st=sb, August 2015.

[17] Ed Greengrass. *Information Retrieval: A Survey.* 2000.

[18] Steve R. Gunn. *Support Vector Machines for Classification and Regression.* 1998.

[19] D. K. Harmann. *Overview of the first Text REtrieval Conference (TREC-1).* NIST Special Publication, 1993.

[20] Erik Hatcher and Otis Gospodnetic. *Lucene in Action.* Manning Publications Co., 2005.

[21] Luhn H.P. *A statistical approach to mechanized endocing and searching of literary information.* IBM Journal of Resarch and Development, 1957.

[22] Karen Sparck Jones. *Natural language processing: a historical review.* 2011.

[23] DIK L. LEE, HUEI CHUANG, and KENT SEAMONS. *Document Ranking and the Vector-Space Model.* 1997.

[24] Lucene. Apache lucene core. https://lucene.apache.org/core/, August 2015.

[25] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schuetze. *An Introduction to Information Retrieval.* Cambridge University Press, 2009.

[26] Microsoft. Full-text predicates and functions overview. https://technet.microsoft.com/en-us/library/ms142583%28v=sql.105%29.aspx, October 2015.

[27] Microsoft. Full-text search architecture. https://msdn.microsoft.com/en-us/library/ms142541%28v=sql.105%29.aspx, October 2015.

[28] Microsoft. Full-text search overview. https://technet.microsoft.com/en-us/library/ms142547%28v=sql.105%29.aspx, October 2015.

[29] Mindbreeze. The european enterprise search appliance. https://www.mindbreeze.com/the-european-enterprise-search-appliance.html, August 2015.

[30] Mindbreeze. System overview. http://help.mindbreeze.com/index.php?topic=doc/Product-Information---Mindbreeze-InSpire-eng/system-overview.htm, September 2015.

[31] Walker Morgan. Ten years of the lucene search engine at apache. http://www.h-online.com/open/news/item/Ten-years-of-the-Lucene-search-engine-at-Apache-1350761.html, September 2011.

[32] MyHistoryLab. *The Beginnings of Civilization*. Person, 2009.

[33] Paulo Quaresma and Teresa Goncalves. *Using Linguistic Information and Machine Learning Techniques to Identify Entities from Juridical Documents*.

[34] Juan Ramos. *Using TF-IDF to Determine Word Relevance in Document Queriesl*.

[35] Gerard Salton. *The SMART Retrieval System - Experiments in Automatic Document Retrieval*. Prentice Hall Inc., 1971.

[36] Amit Singhal. *Modern Information Retrieval: A Brief Overview*. Google Inc.

[37] Solr. Apache solr. http://lucene.apache.org/solr/, September 2015.

[38] Simone Teufel. *An Overview of Evaluation Methods in TREC Ad Hoc Information Retrieval and TREC Question Answering*. Springer, 2007.

[39] Daniel Tunkelang. *Faceted Search*. Morgan and Claypool, 2009.

[40] Panos Vassiliadis and Alkis Simitsis. *Extraction, Transformation, and Loading*. 2007.

[41] Addison Wesley. *Evaluating Search Engines*. 2008.

A Preliminaries Content

A.1 Hash Function

Listing 11: Hash Function

```
public class HashFunction {
   public static void main(String[] args) {
      // terms of dictionary
      String[] input = {"abzuwarten", "anderen", "anzurufende",
          "auch", "Aufgabe", "aus", "Ausschuss"};
      long[] positions = new long[input.length];

      for (int i = 0; i < input.length; i++) {
         // get bytes from string
         byte bytes[] = input[i].getBytes();

         // create object
         Checksum checksum = new CRC32();

         // create crc32 value
         checksum.update(bytes, 0, bytes.length);

         // get crc32 value
           long checksumValue = checksum.getValue();

           // assign crc32 value to array through modulo function
           int pos = (int) (checksumValue % 7);
           positions[pos] = checksumValue;
           System.out.println("position: " + pos + ", term: " +
               input[i] + ", hash value: " +
               Long.toHexString(checksumValue));
      }
   }
}
```

B Lucene Content

B.1 Indexer

Listing 12: Lucene Indexer

```
1  public class Indexer {
2     public static void main(String[] args) throws IOException {
3        String indexDir = "E:\\Lucene\\Index";
4        String dataDir = "E:\\Lucene\\Data";
5
6        File dirD = new File(dataDir);
7        File[] files = dirD.listFiles();
8
9        for (int i = 0; i < files.length; i++) {
10          // (1) Create Lucene index
11          File f = files[i];
12          GermanAnalyzer analyzer = new
                GermanAnalyzer(Version.LUCENE_40);
13          FSDirectory dirI = FSDirectory.open(new File(indexDir));
14           IndexWriterConfig config = new
                 IndexWriterConfig(Version.LUCENE_40, analyzer);
15           IndexWriter writer = new IndexWriter(dirI, config);
16
17           // (2) Index file content and filename
18           Document doc = new Document();
19           doc.add(new Field("text", new FileReader(f)));
20           doc.add(new Field("name", f.getName(), Field.Store.YES,
                 Field.Index.ANALYZED));
21           writer.addDocument(doc);
22           writer.close();
23       }
24    }
25 }
```

B.2 Searcher

Listing 13: Lucene Searcher

```
1  public class Searcher {
2     public static void main(String[] args) throws IOException,
          ParseException {
```

```
3       String indexDir = "E:\\Lucene\\Index";
4       GermanAnalyzer analyzer = new
            GermanAnalyzer(Version.LUCENE_40);
5
6       // (1) Open index
7         IndexReader reader =
              DirectoryReader.open(FSDirectory.open(new
              File(indexDir)));
8        IndexSearcher searcher = new IndexSearcher(reader);
9
10       // (2) Parse query
11       QueryParser parser = new QueryParser(Version.LUCENE_40,
             "text", analyzer);
12       Query query = parser.parse("user query");
13
14       // (3) Search index
15       ScoreDoc[] hits = searcher.search(query, null, 20).scoreDocs;
16
17       for (int i = 0; i < hits.length; i++) {
18         // (4) Retrieve filename
19         Document hit = searcher.doc(hits[i].doc);
20         System.out.println(hit.get("name"));
21       }
22         reader.close();
23    }
24 }
```

B.3 Database Connection and Indexing

Listing 14: Database Connection and Indexing

```
1  // Connect to SQL Server
2  Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
3  conn = DriverManager.getConnection("jdbc:sqlserver://localhost;
4     databaseName=Ris_Bundesnormen", "USER", "PASSWORD");
5
6  // SQL on database Bundesnormen
7  String sqlBundesnormen =
8  "SELECT bd.ID,
9     ISNULL(bd.Dokumentnummer, '') Dokumentnummer,
10    ISNULL(bs.Suchworte, '') Suchworte,
11    ISNULL(bd.Kurztitel, '') Kurztitel,
```

```
12      ISNULL(bd.Langtitel, '') Langtitel,
13      ISNULL(bd.Abkuerzung, '') Abkuerzungen,
14      Paragraphnummer,
15      ISNULL(bd.Paragraphbuchstabe, '') Paragraphbuchstabe,
16      Artikelnummer,
17      ISNULL(bd.Artikelbuchstabe, '') Artikelbuchstabe,
18      ISNULL(bd.Anlagennummer, '') Anlagennummer,
19      ISNULL(bd.Anlagenbuchstabe, '') Anlagenbuchstabe,
20      ISNULL(bd.Anlagenteil, '') Anlagenteil,
21      ISNULL(bd.StammnormPublikationsorgan, '')
            StammnormPublikationsorgan,
22      ISNULL(bd.NovellenPublikationsorgan, '')
            NovellenPublikationsorgan,
23      ISNULL(bd.StammnormBgblnummer, '') StammnormBgblnummer,
24      ISNULL(bd.NovellenBgblnummer, '') NovellenBgblnummer,
25      ISNULL(bd.Typ, '') Typ,
26      ISNULL(bd.IndexText, '') IndexText,
27      ISNULL(bd.Unterzeichnungsdatum, '') Unterzeichnungsdatum,
28      ISNULL(bd.Inkrafttretedatum, '') Inkrafttretedatum,
29      ISNULL(bd.Ausserkrafttretedatum, '') Ausserkrafttretedatum,
30      ISNULL(bd.Veroeffentlichungsdatum, '') Veroeffentlichungsdatum,
31      ISNULL(bd.Aenderungsdatum, '') Aenderungsdatum
32  FROM BundesnormenDokument bd
33  INNER JOIN BundesnormenSuchworteView bs ON bd.ID =
        bs.BundesnormenDokumentID";
34
35  // Create index with custom analyzer
36  CustomAnalyzer analyzer = new CustomAnalyzer();
37  FSDirectory dirI = FSDirectory.open(new File(indexDir));
38  IndexWriterConfig config = new IndexWriterConfig(Version.LUCENE_40,
        analyzer);
39  IndexWriter writer = new IndexWriter(dirI, config);
40
41  // Execute SQL query and index content
42  Statement statement = conn.createStatement();
43  ResultSet rs = statement.executeQuery(sqlBundesnormen);
44
45  while (rs.next()) {
46      // Create Lucene document and the relevant fields
47      Document doc = new Document();
48      doc.add(new Field("Suchworte", rs.getString("Suchworte"),
            Field.Store.YES, Field.Index.ANALYZED,
            Field.TermVector.WITH_OFFSETS));
```

```
49    // add other fields according to the SQL statement
50    writer.addDocument(doc);
51    writer.close();
52 }
```

B.4 Range Query

Listing 15: Numeric Range Query

```
1 int minimum = 1;
2 int maximum = 3;
3 NumericRangeQuery nq =
      NumericRangeQuery.newIntRange("Paragraphnummer", minimum,
      maximum, true, true);
```

Listing 16: Term Range Query

```
1 String min = "a";
2 String max = "d";
3 BytesRef bytes1 = new BytesRef(min);
4 BytesRef bytes2 = new BytesRef(max);
5 Query rq = new TermRangeQuery("Paragraphbuchstabe", bytes1, bytes2,
      true, true);
```

Listing 17: Field Type

```
1 // Field for integer values
2 doc.add(new IntField("Paragraphnummer",
      rs.getInt("Paragraphnummer"), Field.Store.YES));
3
4 // Field for string values
5 doc.add(new Field("Paragraphbuchstabe",
      rs.getString("Paragraphbuchstabe"), Field.Store.YES,
      Field.Index.NOT_ANALYZED, Field.TermVector.WITH_OFFSETS));
6
7 // Field for string values defining the structure of the date
8 doc.add(new Field("Veroeffentlichungsdatum",
      DateTools.dateToString(rs.getDate("Veroeffentlichungsdatum"),
      DateTools.Resolution.DAY), Field.Store.YES,
      Field.Index.ANALYZED, Field.TermVector.YES));
```

B.5 Update

Listing 18: Lucene Update

```
// Open directory
FSDirectory dir = FSDirectory.open(new File(indexDir));
IndexWriter writer = new IndexWriter(dir, new
    IndexWriterConfig(Version.LUCENE_40, analyzer));
Document doc = new Document();

// Update specific document in Lucene index
doc.add(new Field("Suchworte", "new text", Field.Store.YES,
    Field.Index.ANALYZED));
writer.updateDocument(new Term("ID", "123"), doc);
writer.close();
```

B.6 Auto Corrections

Listing 19: Lucene Auto Corrections

```
public class SpellCheckIndexer {

   private static IndexWriter writer;
   private static GermanAnalyzer analyzer = new
       GermanAnalyzer(Version.LUCENE_40);

   public static void main (String[] args) throws IOException{

      String spellCheckDir = "E:\\RISIndex\\Justiz\\WordIndex";
      String indexDir = "E:\\RISIndex\\Justiz";
      String indexField = "Suchworte";

      // Create Lucene index for dictionary
      FSDirectory dir = FSDirectory.open(new File(indexDir));
      IndexWriterConfig config = new
          IndexWriterConfig(Version.LUCENE_40, analyzer);
      config.setOpenMode(OpenMode.CREATE);
      writer = new IndexWriter(dir, config);

      // Create dictionary on field Suchworte
      FSDirectory spellDir = FSDirectory.open(new
          File(spellCheckDir));
```

```
20      SpellChecker spell = new SpellChecker(spellDir);
21      IndexReader r = DirectoryReader.open(FSDirectory.open(new
            File(indexDir)));
22      try {
23         spell.indexDictionary(new LuceneDictionary(r, indexField),
               config, false);
24      } finally {
25         r.close();
26      }
27   }
28}
```

B.7 Faceting

Listing 20: Category

```
1 List categories = new ArrayList();
2 categories.add(new CategoryPath("Gericht",
      rs.getString("Gericht")));
3 CategoryDocumentBuilder categoryDocBuilder = new
      CategoryDocumentBuilder(taxoWriter);
4 categoryDocBuilder.setCategoryPaths(categories);
5 categoryDocBuilder.build(doc);
```

Listing 21: Faceting

```
1  public static void faceting (String indexDir, String taxoDir,
       String q)
2        throws Exception{
3     IndexReader indexReader =
          DirectoryReader.open(FSDirectory.open(new File(indexDir)));
4     IndexSearcher searcher = new IndexSearcher(indexReader);
5     TaxonomyReader taxoReader = new
          DirectoryTaxonomyReader(FSDirectory.open(new File(taxoDir)));
6
7     Query parser = new QueryParser(Version.LUCENE_40, "Suchworte",
          analyzer).parse(q);
8     TopScoreDocCollector tdc = TopScoreDocCollector.create(10, true);
9
10    FacetSearchParams facetSearchParams = new FacetSearchParams();
11    facetSearchParams.addFacetRequest(new CountFacetRequest(
12          new CategoryPath("Gericht"), 10));
```

```
13    FacetsCollector facetsCollector = new FacetsCollector(
14          facetSearchParams, indexReader, taxoReader);
15
16    searcher.search(parser, MultiCollector.wrap(tdc,
          facetsCollector));
17    List<FacetResult> res = facetsCollector.getFacetResults();
18 }
```

B.8 Synonyms

Listing 22: Synonyms

```
1  public class Synonyms {
2
3     public static void main(String[] args) {
4
5        String base1 = "fast";
6        String syn1 = "rapid";
7
8        // Define synonyms
9        SynonymMap.Builder sb = new SynonymMap.Builder(true);
10       sb.add(new CharsRef(base1), new CharsRef(syn1), true);
11
12       // Create synonyms
13       SynonymMap synonym = null;
14       try {
15          synonym = synonym = sb.build();
16       } catch (IOException e) {
17          e.printStackTrace();
18       }
19
20       // Apply synonym filter on token stream
21        Tokenizer tokenizer = new
             WhitespaceTokenizer(Version.LUCENE_40, new
             StringReader("INPUT TEXT"));
22        SynonymFilter filter = new SynonymFilter(tokenizer, synonym,
             true);
23    }
24 }
```

B.9 Segmentation

Listing 23: Segmentation

```
public class MultiFieldFilter extends TokenFilter{

    private final CharTermAttribute termAtt;
    private final PositionIncrementAttribute posAtt;
    private final OffsetAttribute offAtt;
    private int position;

  protected MultiFieldFilter(TokenStream input) {
     super(input);
      this.termAtt = addAttribute(CharTermAttribute.class);
      this.posAtt = addAttribute(PositionIncrementAttribute.class);
      this.offAtt = addAttribute(OffsetAttribute.class);
  }

  @Override
  public boolean incrementToken() throws IOException {
      if (!input.incrementToken()) return false;

      // Position of the current token
      position = posAtt.getPositionIncrement();

      // Increase the distance between tokens when new line is
         detected
      if(termAtt.toString().equals("\n")){
           posAtt.setPositionIncrement(position+10);
      }
        return true;
  }
}
```

Listing 24: Span Query

```
SpanQuery[] span = new SpanQuery[]{
  new SpanTermQuery(new Term("ID", term1)),
  new SpanTermQuery(new Term("ID", term2))};

SpanNearQuery sq = new SpanNearQuery(span, 2, true);
```

B.10 Canonisation

Listing 25: Canonisation

```
1  public class CanonicalFilter extends TokenFilter{
2
3      private final CharTermAttribute termAtt;
4      private final PositionIncrementAttribute posAtt;
5      private final OffsetAttribute offAtt;
6      private int position;
7      private State savedState;
8      private LinkedList<String> tokens = new LinkedList<String>();
9
10   protected CanonicalFilter(TokenStream input) {
11      super(input);
12       this.termAtt = addAttribute(CharTermAttribute.class);
13       this.posAtt = addAttribute(PositionIncrementAttribute.class);
14       this.offAtt = addAttribute(OffsetAttribute.class);
15   }
16
17   @Override
18   public boolean incrementToken() throws IOException {
19       // Add token from the list to the current position
20       if (!tokens.isEmpty() && termAtt.toString().equals("§7
             ABGB")) {
21           restoreState(savedState);
22           termAtt.setEmpty().append(tokens.remove());
23           return true;
24       }
25
26       // Save the current position of the token and add token to a
             list
27       if(input.incrementToken()){
28         if(termAtt.toString().equals("ABGB 7§")){
29            tokens.add(termAtt.toString());
30            savedState = captureState();
31         }
32         return true;
33       }
34       return false;
35   }
36 }
```

B.11 Custom Analyzer

Listing 26: Custom Analyzer

```
1
2  public class CustomAnalyzer extends Analyzer{
3
4     protected TokenStreamComponents createComponents(String
          fieldName, Reader reader) {
5
6        StandardTokenizer tokenizer = new
             StandardTokenizer(Version.LUCENE_40, reader);
7        TokenFilter filter = new StandardFilter(Version.LUCENE_40,
             tokenizer);
8        filter = new LowerCaseFilter(Version.LUCENE_40, filter);
9
10       // Remove german stop words
11       filter = new StopFilter(Version.LUCENE_40, filter,
             GermanAnalyzer.getDefaultStopSet());
12
13       // Normalizatiion of tokens
14       filter = new GermanNormalizationFilter(filter);
15       // stemming
16       filter = new GermanLightStemFilter (filter);
17
18       // Custom filters
19       filter = new CanonicalFilter(filter);
20       filter = new MultiFieldFilter(filter);
21
22       return new TokenStreamComponents(tokenizer, filter);
23    }
24 }
```

B.12 Entity Extraction

Listing 27: Multiple Analyzers

```
1 Map<String,Analyzer> analyzerPerField = new HashMap<String,Analyzer
      >();
2 analyzerPerField.put("SuchworteFacet", new EntityExtractionAnalyzer
      ());
3 PerFieldAnalyzerWrapper aWrapper = new PerFieldAnalyzerWrapper(new
      GermanAnalyzer(Version.LUCENE_40), analyzerPerField);
```

Listing 28: Entity Extraction Analyzer

```
public class EntityExtractionAnalyzer extends Analyzer{
   @Override
   protected TokenStreamComponents createComponents(String field,
       Reader reader) {
      Pattern p = Pattern.compile("(0[1-9]|[12][0-9]|3[01])[-
          /.](0[1-9]|1[012])[- /.](19|20)\\d\\d");
      PatternTokenizer tokenizer = null;
      try {
         tokenizer = new PatternTokenizer(reader, p, 0);
      } catch (IOException e) {
         e.printStackTrace();
      }
      return new TokenStreamComponents(tokenizer);
   }
}
```

C Mindbreeze Content

C.1 Query Expansion

Listing 29: Query Transformer

```
1  public class MyTransformer
2    extends QueryExprTransformationServiceProtos.
         QueryExprTransformationService
3  {
4    private static final Logger logger = Logger.getLogger(MyTransformer
         .class);
5
6    public void transform(RpcController rpcController,
         QueryExprTransformationServiceProtos.
         QueryExprTransformationRequest request, RpcCallback<
         QueryExprTransformationServiceProtos.
         QueryExprTransformationResponse> done)
7    {
8      try
9      {
10       logger.info("input:" + request.getQueryExpr());
11       QueryExprProtos.QueryExpr transformedQueryExpr =
             transformQueryExpr(request.getQueryExpr());
12       logger.info("transformed:" + transformedQueryExpr);
13       QueryExprTransformationServiceProtos.
             QueryExprTransformationResponse.Builder responseBuilder =
             QueryExprTransformationServiceProtos.
             QueryExprTransformationResponse.newBuilder();
14       responseBuilder.setQueryExpr(transformedQueryExpr);
15       QueryExprTransformationServiceProtos.
             QueryExprTransformationResponse response = responseBuilder.
             build();
16       done.run(response);
17     }
18     catch (Exception e)
19     {
20       throw new RuntimeException(e);
21     }
22   }
23
24   QueryExprProtos.QueryExpr transformQueryExpr(QueryExprProtos.
         QueryExpr expr)
```

```
25  {
26    if ((expr.getKind() == QueryExprProtos.QueryExpr.Kind.
          EXPR_LABELED) && (expr.hasNamedExpr()))
27    {
28      if ("Paragraph".equalsIgnoreCase(expr.getNamedExpr().getLabel()
            )) {
29        if ((expr.getNamedExpr().hasExpr()) && (expr.getNamedExpr().
            getExpr().getKind() == QueryExprProtos.QueryExpr.Kind.
            EXPR_QUOTED_TERM))
30        {
31          QueryExprProtos.QueryExpr.Labeled labeled = expr.
              getNamedExpr();
32          QueryExprProtos.QueryExpr paragraph_ziel = QueryExprHelper.
              labeled("Paragraph", QueryExprHelper.term(labeled.
              getExpr().getQuotedTerm().replace(" ", "")));
33
34          return QueryExprProtos.QueryExpr.newBuilder(expr).
              setNamedExpr(QueryExprProtos.QueryExpr.Labeled.
              newBuilder(labeled).setLabel("Paragraph").setExpr((
              QueryExprProtos.QueryExpr)transformOnlyFieldsOfMessage(
              paragraph_ziel))).build();
35        }
36      }
37      if (("title".equalsIgnoreCase(expr.getNamedExpr().getLabel()))
            &&
38        (expr.getNamedExpr().hasExpr()) && (expr.getNamedExpr().
            getExpr().getKind() == QueryExprProtos.QueryExpr.Kind.
            EXPR_QUOTED_TERM))
39      {
40        QueryExprProtos.QueryExpr.Labeled labeled = expr.getNamedExpr
              ();
41        QueryExprProtos.QueryExpr ziel = QueryExprHelper.labeled("
            title", QueryExprHelper.regexPattern(labeled.getExpr().
            getQuotedTerm().replace("_", ".").replace("*", ".*")));
42
43        return QueryExprProtos.QueryExpr.newBuilder(expr).
            setNamedExpr(QueryExprProtos.QueryExpr.Labeled.newBuilder(
            labeled).setLabel("title").setExpr((QueryExprProtos.
            QueryExpr)transformOnlyFieldsOfMessage(ziel))).build();
44      }
45    }
46    return (QueryExprProtos.QueryExpr)transformOnlyFieldsOfMessage(
          expr);
```

```
47  }
48
49  private Message transformMessageAndFields(Message inputMessage)
50  {
51    if (QueryExprProtos.QueryExpr.getDescriptor().equals(inputMessage
        .getDescriptorForType())) {
52      return transformQueryExpr((QueryExprProtos.QueryExpr)
          inputMessage);
53    }
54    return transformOnlyFieldsOfMessage(inputMessage);
55  }
56
57  private Message transformOnlyFieldsOfMessage(Message inputMessage)
58  {
59    Message.Builder builder = inputMessage.newBuilderForType().
        mergeFrom(inputMessage);
60    for (Map.Entry<Descriptors.FieldDescriptor, Object> entry :
        inputMessage.getAllFields().entrySet()) {
61      if (((Descriptors.FieldDescriptor)entry.getKey()).getType() ==
          Descriptors.FieldDescriptor.Type.MESSAGE) {
62        if (((Descriptors.FieldDescriptor)entry.getKey()).isRepeated
            ())
63        {
64          Collection<?> collection = (Collection)entry.getValue();
65          List<Object> transformed = new ArrayList();
66          for (Object msg : collection) {
67            transformed.add(transformMessageAndFields((Message)msg));
68          }
69          builder.setField((Descriptors.FieldDescriptor)entry.getKey
              (), transformed);
70        }
71        else if (QueryExprProtos.QueryExpr.getDescriptor().equals(((
            Descriptors.FieldDescriptor)entry.getKey()).getMessageType
            ()))
72        {
73          QueryExprProtos.QueryExpr expr = (QueryExprProtos.QueryExpr
              )entry.getValue();
74          builder.setField((Descriptors.FieldDescriptor)entry.getKey
              (), transformQueryExpr(expr));
75        }
76        else
77        {
```

```
78          builder.setField((Descriptors.FieldDescriptor)entry.getKey
                ((), transformOnlyFieldsOfMessage((Message)entry.getValue
                ()));
79        }
80      }
81    }
82    return builder.buildPartial();
83  }
84 }
```

D RIS Content

D.1 Database Bundesnormen

Name	Type	Description
BundesnormenDokument	Table	This table contains all metadata for a document
BundesnormenDokumentDaten	Table	Contains the unstructured documents in their respective format
BundesnormenDokumentNative	Table	A document can consist of several parts. Each of those parts has an entry in this table
BundesnormenHistory	Table	Shows the modification history
BundesnormenHistoryDocument	Table	Contains the original data
BundesnormenIndexItem	Table	Contains all index data in order to guarantee a fast search
BundesnormenNutzdaten	Table	Contains the text of the documents
VersionInfo	Table	This table contains the current version of the application, which can be used in update scripts and queries
BundesnormenSuchworteView	View	This view consists of a combined text that includes all metadata fields and the all attributes from the BundesnormenNutzdaten table
BundesnormenTitelView	View	Contains all parts of the title necessary for a full text query

Table 22: Database Bundesnormen

D.2 Queries

Requirement	Query
Right wildcard	Suchworte:Radfahrstreifen*
Multiple wildcard	Suchworte:Arbeitnehmer*schutz*
Near Search	Suchworte:Fahrbahnen nahe Radfahrstreifen
Combined Search 01	Suchworte:Fahrstreife* nahe Radfahrstreifen*
Combined Search 02	"Suchworte:((Abwasser und Anlage) oder Verordnung) nahe Emission Titel: Milch"
Special Characters	Suchworte:§3
Numbers	Gliederungszahl:001*
Metadata and Full-text Search	"Suchworte:Buddhismus Titel:Lehrplan und Religion"
Different Spelling 01	Paragraph:1a
Different Spelling 02	Paragraph:1 a
Word Delimiter	"Geschäftszahl:OEA14/72 Suchworte:§3 "
Boolean: AND	Suchworte:Bund und Gericht
Boolean: OR	Suchworte:Bund oder Gericht
Boolean: NOT	Suchworte:Bund nicht Gericht
Range Query	"Titel:Asylgesetz Paragraph:32a bis 33b"
Group of Words	Suchworte:BGBl I
Group of Words including Stop Words	Suchworte:"der bürger"
Segmentation	Norm:ZPO Abs1
Canonisation 01	Norm:ABGB §7
Canonisation 02	Norm:§7 ABGB
Global Search	Suchworte:gerichtshof

Table 23: RIS Queries