Entwicklung und Evaluation eines Information Retrieval Systems

für die Unterstützung des Betriebs von PLM Systemen

Bachelorarbeit

für die Prüfung zum

Bachelor of Science

Studiengang *Angewandte Informatik*

Duale Hochschule Baden-Württemberg Mannheim

von

Lukas Retschmeier

Abgabedatum:	24. September 2018
Bearbeitungszeitraum:	12 Wochen
Matrikelnummer, Kurs:	1339518, TINF15/AI-BI
Ausbildungsfirma:	Atos Information Technology GmbH
Betreuer der Ausbildungsfirma:	Markus Reitberger, B. Sc.
	Dipl.-Phys. Norbert Hranitzky
Gutachter der Dualen Hochschule	Prof. Dr. Karl Stroetmann

EIDESSTATTLICHE ERKLÄRUNG

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

> *Entwicklung und Evaluation eines Information Retrieval Systems für die Unterstützung des Betriebs von PLM Systemen*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

München, den 22. September 2018

Lukas Retschmeier

ABSTRACT

Deutsch Dokumentationen und angesammeltes Wissen für Software liegen oftmals verteilt vor und Administratoren und Entwickler im Product Lifecycle Management (PLM) Umfeld müssen diese Informationen in verschiedenen Quellen suchen. Ziel dieser Bachelorarbeit ist es, ein Information Retrieval System zu entwickeln, das es erlaubt, den Volltext dieser Softwaredokumentationen in Echtzeit zu durchsuchen. Die Daten wurden mit Hilfe von *Elasticsearch* indexiert. Eine Evaluation hat gezeigt, dass die Scoringfunktion *Okapi BM25* die besten Ergebnisqualitäten zurückliefert. Hierbei wurden die drei Funktionen BM25, DFR ud DFI auf einen Fragekorpus von 50 Fragen ausgewertet, wobei immer die Top-20 Ergebnisse nach binärer Relevanz beurteilt worden sind.

English Documentations and accumulated knowledge for software is often distributed and administrators and developers in the Product Lifecycle Management (PLM) environment have to obtain this information laboriously from various sources. The goal of this bachelor thesis is to develop an Information Retrieval (IR) System, which allows the full text of these software documentations to be searched in real time. The data had been indexed with *Elasticsearch*. An evaluation has shown that the scoring function *BM25* returns the best quality of results. Therefore the three functions BM25, DFR und DFI were compared with the top-20 results of a corpus of 50 questions.

INHALTSVERZEICHNIS

Eidesstattliche Erklärung I

Abstract II

Abbildungsverzeichnis VIII

Listings X

Tabellenverzeichnis XII

Abkürzungsverzeichnis XIII

Mathematische Symbole XV

1 Einleitung 1

1.1 Problemstellung 3

1.2 Motivation 3

1.3 Aufbau der Arbeit 4

2 Theorie **5**

2.1 IR Systeme . . . 5

2.1.1 Was ist Information Retrieval (IR) ? . . . 6

2.1.1.1 Begriffsklärung . . . 6

2.1.1.2 Anwendungen . . . 8

2.1.2 Architektur einer Suchmaschine . . . 10

2.1.2.1 Der Prozess des Indexierens . . . 10

2.1.2.2 Der Prozess der Abfrage . . . 12

2.1.3 Indexerstellung (Index Construction) . . . 13

2.1.3.1 Tokenizing . . . 14

2.1.3.2 Entfernen von Stopwords . . . 15

2.1.3.3 Token Normalization . . . 16

2.1.3.4 Stemming . . . 17

2.1.3.5 Erstellung des Invertierten Index . . . 17

2.1.4 Scoring Funktionen . . . 20

2.1.4.1 TF/IDF Modell . . . 20

2.1.4.2 Okapi BM25 . . . 22

2.1.4.3 Divergence from Randomness (DFR) . . . 24

2.1.4.4 Divergence From Independence . . . 25

2.2 Open-Source Suchmaschinen auf Lucene Basis . . . 26

2.2.1 Apache Solr . . . 27

2.2.2 Elasticsearch (ES) . . . 28

2.2.2.1 Geschichtlicher Hintergrund . . . 28

2.2.2.2 Logisches Layout . . . 29

2.2.2.3 Physikalisches Layout . . . 30

2.2.2.4 Kommunikation über eine REST API 31
2.2.2.5 Implementation von Suchmaschinenprozessen 32
2.2.2.6 ELK Stack 34
2.2.2.7 Ändern der Scoringfunktion eines Indexes 35
2.3 Product Lifecycle Management (PLM) 35
2.3.1 Siemens Teamcenter 36
2.3.1.1 Was ist Teamcenter? 37
2.3.1.2 Dokumentationsquellen für Administratoren 37
2.4 Evaluierung eines IR Systems 38
2.4.1 Grundlagen 39
2.4.2 Evaluierung der gewichteten Ergebnisse 40
2.4.2.1 Precision@k 40
2.4.2.2 Average Precision (AP) 41
2.4.2.3 Mean Average Precision (MAP) 41
2.4.3 Bestimmung des Recalls 42

3 Anforderungsanalyse **44**
3.1 Funktionale Anforderungen (AF) 44
3.2 Nichtfunktionale Anforderungen (AN) 46
3.3 Technische Anforderungen (AT) 46

4 Methode **47**
4.1 Verwendete Werkzeuge 47
4.2 Evaluierung der Scoring Funktionen 48

5 Durchführung 49

5.1 Architektur der Lösung . . . 49

5.1.1 Data Tier . . . 50

5.1.2 Web Tier . . . 50

5.1.3 Index Tier . . . 51

5.1.4 Client Tier . . . 51

5.2 Installation des Index Servers . . . 51

5.2.1 Elasticsearch vs. Solr . . . 51

5.2.2 Installation des ELK Stacks . . . 53

5.2.3 Definition des Indexaufbaus . . . 54

5.3 Füllen der Indexdatenbank . . . 55

5.3.1 Migrieren des PLM Documentation Servers . . . 55

5.3.1.1 Erstellen eines Docker Containers für PLM-DS . . . 56

5.3.1.2 Migrieren der Daten nach ES . . . 56

5.3.2 Indexierung von GTAC Solution Center Einträgen . . . 57

5.3.2.1 Extraction der Problem Reports (PR) . . . 57

5.3.2.2 Erstellung des Index und Import der Daten (Importer) . . . 58

5.3.3 Webcrawler . . . 60

5.3.4 Verarbeitung von Binärdokumenten . . . 62

5.4 Entwicklung des Frontends . . . 62

5.4.1 Initialisieren der Verbindung zwischen ES und Front-End . . . 63

5.4.2 Darstellung der Komponenten . . . 63

5.4.2.1 Suchleiste . . . 64

5.4.2.2 Filtern von Einträgen . . . 65

5.4.2.3 Anzeigen der Ergebnisse . . . 66

5.4.3 Weiterleitung auf entsprechende Quellen 67

6 Evaluation **68**

6.1 Vorgehen 68

6.1.1 Wahl der zu untersuchenden Scoringmodelle 68

6.1.2 Wahl eines Fragenkorpus 69

6.1.3 Durchführung der Evaluation 70

6.2 Ergebnisse 71

6.2.1 Average Precision at Position k (AP@k) 71

6.2.2 MAP 72

6.2.3 Zusammenfassung 73

6.2.4 Reaktionszeit des Systems 73

7 Schluss **74**

7.1 Erfüllungsgrad der Anforderungen 74

7.2 Fazit 75

Literaturverzeichnis **76**

A Anhang **I**

A.1 Docker Compose File für Elastic PLM Stack I

A.2 Rohdaten der Evaluation II

A.2.1 Fragen und AP Werte III

A.2.2 Average Relevance und Precision at k IV

ABBILDUNGSVERZEICHNIS

2.1 Abgrenzung der Begriffe Daten, Wissen und Information (aus: [Hen08, S. 20]) 7

2.2 Entität-Erkennung mithilfe des *displaCy Named Entity Visualizer* (Eigene Abbildung) 9

2.3 *WolframAlpha* kann in natürlicher Sprache gestellte Fragen beantworten (Eigene Abbildung) 10

2.4 Der Prozess des Indexierens (aus: [CMS11, S. 15])) 11

2.5 Der Prozess der Abfrage (aus: [CMS11, S. 16]) 12

2.6 Schritte zur Erstellung eines invertierten Indexes (Eigene Abbildung) . . . 13

2.7 Auszug aus einem Buchindex (Auszug aus: [MRS08, S.544]) 18

2.8 Auszug aus einem invertierten Index der 37 Werke von *William Shakespeare* (aus: [BCC16, S. 37]) 19

2.9 Logo von Apache Lucene (von: https://lucene.apache.org/core/) 27

2.10 Logo von Apache Solr (von: http://lucene.apache.org/solr/logos-and-assets.html) 27

2.11 Logo von *Elasticsearch*
(von: https://www.elastic.co/de/products/elasticsearch) 28
2.12 Sicht einer Applikation auf Elasticsearch (eigene Abbildung) 31
2.13 Physikalischer Aufbau eines ES Clusters (eigene Abbildung) 32

5.1 Grobarchitektur der gesamten Lösung (eigene Abbildung) 52
5.2 Google Trends: Solr vs. ES (aus [Goo18]) 53
5.3 Migrieren der Daten von SOLR to ES (eigene Abbildung) 57
5.4 GTAC Solution Center (eigener Screenshot) 58
5.5 Indexierung von Binärdokumenten (eigene Abbildung) 62
5.6 Die Searchkit-UI (eigener Screenshot) 64

6.1 Average Precision at Position k (eigene Abbildung) 72

LISTINGS

2.1 Macbeth als JSON Dokument 30
2.2 Erstellung eines Indexes 33
2.3 Indexierung eines Dokumentes 33
2.4 Stellen einer Abfrage 34
2.5 Schließen eines offenen Indexes 35
2.6 Ändern der Scoringfunktion 36

5.1 Beispiel für Extraktion aus Solution Center 59
5.2 Erstellung des Indexes 60
5.3 Versenden der Bulks 61
5.4 Setzen von Crawler Parametern 61
5.5 Callbackfunktionen für Events 61
5.6 Konfigurieren des Searchkit Managers 63
5.7 Such Komponente 64
5.8 Filter Komponente 65
5.9 Ergebnisliste 66
5.10 Definition der einzelnen Hit Items 66

5.11 Weiterleitung zu den Ergebnissen 67

TABELLENVERZEICHNIS

2.1 Verschiedene Arten von Tokenizern 15

2.2 Berechnung von P@k Werten zu zwei Queries 42

6.1 Scoringfunktionen und Parameter 69

6.2 MAP/GMAP Werte 73

7.1 Erfüllung der Anforderungen 75

A.2 Average Relevance (AR@k) und Average Precision at K (AP@k) für BM25, DFI und DFR V

ABKÜRZUNGSVERZEICHNIS

ADT	Abstrakter Datentyp / Abstract Datatype
AF	Funktionale Anforderungen
AIT	Atos Information Technology GmbH
AN	Nicht-Funktionale Anforderungen
AP	Average Precision
API	Application Programming Interface
ASF	Apache Software Foundation
AT	Technische Anforderungen
AWC	Active Workspace Client
CRUD	Create - Read - Update - Delete
DHBW	Duale Hochschule Baden-Württemberg
BM25	Okapi (B)est (Matching) 25 (Scoringfunktion)
DFI	Divergence from Independence (Scoringfunktion)
DFR	Divergence from Randomness (Scoringfunktionsframework)
DS	Documentation Server
ELK	(Neuerdings: *Elastic Stack*): Technologiestack aus den Tools *Elasticsearch, Logstash, Kibana und Beats*

ES	Elasticsearch
FAQ	Frequently Asked Questions
GTAC	(Siemens) Global Technical Access Center
GMAP	Geometric Mean Average Precision
HTML	HyperText MarkupLanguage
IDF	Inverse Dokumentenhäufigkeit / Inverse Document Frequency
IR(S)	Information Retrieval (System)
ISBN	International Standard Book Number
JS	JavaScript
JSON	Java Script Object Notation
TC	(Siemens) Teamcenter
TREC	Text REtrieval Conferences
MAP	Mean Average Prevision
(N)DCG	(Normalized) Discounted Comulative Gain
PLM	Product Lifecycle Management
PR	Problem Report
P@k	Precision at k
R@k	Relevance at k
REST	Representational State Transfer
URL	Uniform Resource Locator

MATHEMATISCHE SYMBOLE

Symbol	Bedeutung
d	Dokument
df	Document Frequency / Dokumenthäufigkeit
dl	Document Length / Dokumentlänge
e_{td}	Erwartete Termhäufigkeit eines Terms t in einem Dokument d
G	Grundgesamtheit
idf	Inverse Document Frequency / Umgekehrte Dokumenthäufigkeit
L_d	Document Length / Länge des Dokuments d
L_{ave}	Average Document Length / Durchschnittliche Dokumentenlänge
N	Gesamtanzahl der Dokumente
q	Query / Abfrage
Rel	Menge der relevanten Dokumente
$relevance(d)$	Binäre Relevanz eines Dokumentes: $relevance : d \rightarrow \{0, 1\}$
Res	Menge der zurückgegebenen Dokumente
t	Term
tf	Term Frequency / Termhäufigkeit
TF	Absolute Termhäufigkeit
Ω	Undefinierter Wert
ω_{td}	Gewichtsfunktion eines Terms t in Dokument d

KAPITEL 1

EINLEITUNG

Bedeutsame Informationen aus großen Datenmengen zu beziehen ist eine schwierige Aufgabe.

Bereits 288 v. Chr. versuchte die von Ptolemäus gegründete *bibliotheca alexandrina* das Wissen der damaligen Zeit auf 900.000 Papyrusrollen festzuhalten (vgl. [Lic11]). Um in diesem Bestand überhaupt etwas zu finden, haben die Alexandriner ein System entwickelt, bei dem alle Bücher eine von zehn Kategorien zugewiesen wurde: Rhetorik, Recht, Epik, Tragik, Lyrik, Geschichte, Medizin, Mathematik, Naturwissenschaft und Miscellanea (Gemischtes). In 120 Bänden entstand ein *Index*, alphabetisch nach den Namen der Autoren in den Kategorien aufsteigend sortiert. Die einzelnen Einträge wurden mit weiteren Informationen wie einer Kurzbiografie des Autors, dem Titel des Werkes und einem Kommentar angereichert. Dadurch konnten schnell wichtige Werke zu bestimmten Themen gefunden werden.

Leider wurde die *bibliotheca alexandrina* und damit der gesamte bibliothekarische Bestand durch ein Feuer zerstört. Das Registrierungsmodell wurde jedoch sehr bald zum Standard für alle weiteren Bibliotheken bis in die Neuzeit (vgl. [Lic11]).

Im Zuge der rapiden technischen Entwicklung ab dem Ende des 20. Jahrhundert wurden neue Möglichkeiten geschaffen, Bücher durchsuchbar zu machen. Werke konnten von da an als Katalogisate in elektronische Verzeichnisse aufgenommen werden, die eine schnelle Verwaltung und Suche in den Beständen ermöglichte. Die vorher mühsame Indexierung per Hand kann nun von Algorithmen und geeigneten abstrakten Indexstrukturen übernommen werden.

Mit der Erfindung des Internets durch *Tim Berners Lee* 1989 begann eine regelrechte Informationsexplosion. Einer Schätzung von http://www.worldwidewebsize.com/ zufolge besteht alleine das indexierte Internet Anfang Juli 2018 aus 45 Milliarden Webseiten. Allein 2016 ist laut einer Prognose der *International Data Corporation* weltweit eine Datenmenge von 16 Zettabyte (1ZB = 10^{21} Bytes) an digitalen Daten generiert worden. Bis 2025 soll sich diese Zahl auf jährlich 163 ZB verzehnfachen (vgl. [RGR17, S. 3]). Um mit solchen Daten zu hantieren, wurden *Information Retrieval Systeme (IRS)* entwickelt. Diese beschäftigen sich mit der "'Verarbeitung, Visualisierung und Manipulation von Texten in natürlicher Sprache" (vgl. [BCC16, S. 2]).

Nicht jedes Wissen ist frei im Internet verfügbar und indexiert, denn Unternehmen greifen auf eigene Quellen mit Spezialwissen zurück. Das Problem ist, dass diese Dokumente häufig dezentral in verschiedenen Quellen liegen und nicht gleichzeitig durchsucht werden können.

Auch im Product Lifecycle Management (PLM) Umfeld ist diese Problematik vorhanden, denn die Dokumentationen zu dort verwendeter Software lassen sich an verschiedenen Orten finden. Für Administratoren ist es mühsam, die Quellen zu finden, die sie gerade brauchen. Daher wäre es wünschenswert, ein System zu entwickeln, das eine Lösung bereitstellt.

1.1 Problemstellung

Die Suche nach Softwareinformationen für PLM Administratoren ist zeitaufwändig. Es gibt viele verschiedene dezentrale Quellen, weswegen auf der Suche nach der richtigen Information diese einzeln durchsucht werden müssen. Das Ziel dieser Arbeit ist es, ein Information Retrieval (IR) System zu entwickeln, das eine Echtzeitsuche über mehrere Dokumentationen zu Produkten des *Product Lifecycle Management*s ermöglicht. Ein Schwerpunkt liegt bei den Dokumentationen der PLM Software *Siemens Teamcenter*. Über eine Webapplikation wird es Administratoren und Entwicklern ermöglicht, dieses System in Form einer Suchmaschine zu bedienen und die Inhalte zu erreichen.

Damit eine effektive Suche der Daten gewährleistet ist, ist die Wahl einer Bewertungsfunktion essentiell. Diese entscheidet, nach welchen Kriterien die Ergebnisse einer Abfrage gefunden und geordnet werden.

1.2 Motivation

Diese Arbeit ist im Rahmen einer Praxisphase in der PLM Abteilung des Unternehmens *Atos Information Technology GmbH* am Standort *München Perlach* entstanden und wird als *Bachelorarbeit* an der *Dualen Hochschule Baden-Württemberg Mannheim* eingereicht. Aufgrund der Dezentralität der Dokumentationen, wäre eine Lösung, die eine effektive Suche zur Verfügung stellt, wünschenswert.

1.3 Aufbau der Arbeit

Den Leser[1] erwarten in dieser Arbeit sechs größere Blöcke:

- den **Theorieteil**, der Grundlagen zu IR Systemen und Bewertungsfunktionen erklärt;
- die **Durchführung**, die erläutert, wie bei der Entwicklung der Lösung vorgegangen wurde;
- die **Anforderungsanalyse**, in der Anforderungen, die sich aus der Problemstellung ergeben, definiert werden;
- der **Methodenteil**, der die in dieser Arbeit verwendeten Methoden auflistet;
- den **Evaluierungsteil**, in dem eine optimale Bewertungsfunktion gesucht wird;
- und den **Schluss**, in dem die Anforderungen überprüft werden und ein Ausblick gegeben wird.

[1] Die Verwendung des generischen Maskulinums in dieser Arbeit, als sprachliche Vereinfachung für z.B. den Begriff des *Lesers*, dient **einzig und allein** der besseren Lesbarkeit des Textes.

KAPITEL 2

THEORIE

'Do you mean that you think you can find out the answer to it?' said the March Hare.

'Exactly so,' said Alice.

Lewis Carroll, *Alice in Wonderland*

Im weiteren Verlauf der Arbeit wird ein eigenes *Information Retrieval System* entwickelt, das eine Suche über mehrere Product Lifecycle Management (PLM) Portale ermöglicht. Dadurch wird es möglich sein, Fragen, die speziell den Betrieb von PLM Produkten betreffen, beantworten zu lassen. Ferner wird dieses System evaluiert und die Qualität der Antworten analysiert. Hierfür werden in diesem Kapitel die theoretischen Grundlagen gelegt.

2.1 Information Retrieval (IR) Systeme

Ein Information Retrieval (IR) System beschäftigt sich mit der "Darstellung, Durchsuchung und Manipulation von großen Sammlungen elektronischer Texte und anderen Daten, die

in natürlicher Sprache vorliegen" (vgl. [BCC16, S. 2]). Internet Suchmaschinen gehören zu den bekanntesten und meistgenutzten IR Systemen. Google, DuckDuckGo oder Bing erlauben es Anwendern, gezielt Informationen im Internet zu suchen, wobei die Dienste auch eine Menge weiterführender Funktionen, wie beispielsweise das Aufbereiten von Nachrichten, Aktienverläufen, Flügen und mehr bieten.

Chatbots können verwendet werden, um den First-Level-Support zu automatisieren, denn etwa 30%-50% aller Support-Fälle sind repetitiv (vgl. [Par17]). Diese haben Zugriff auf eine Datenbank mit den häufigsten Problemen und versuchen, die Frage des Nutzers zu verstehen und eine Antwort innerhalb ihrer Wissensbasis zu finden.

2.1.1 Was ist Information Retrieval (IR) ?

Zunächst wird geklärt, was der Begriff *Information Retrieval* bedeutet.

2.1.1.1 Begriffsklärung

Der Begriff *Information Retrieval* (IR) setzt sich aus zwei Teilen zusammen: *Information* und *Retrieval*, die zunächst unabhängig voneinander betrachten werden.

Information Unter Daten versteht der Informatiker die rein syntaktische Darstellung von Information (vgl. [Hen08, S. 18]). Hierbei dreht sich alles um die Frage *wie* Informationen gespeichert werden - beispielsweise, welche Formate für die Codierungen verwendet werden.

Wenn die hinterlegten *Daten* eine semantische Bedeutung erhalten, entsteht Wissen. Während einzelne Farbwerte eines Bildes nur zu den Daten zählen, ist das in den Bildern Dargestellte bereits das Wissen.

Laut Kuhlen ist *Information die Teilmenge von Wissen, die von jemandem in einer konkreten Situation zur Lösung von Problemen benötigt wird* (vgl. [Rai90]). Häufig muss für eine bestimmte Handlung oder Entscheidung Wissen aus verschiedenen Quellen zusam-

mengefasst werden, in den Lehrbüchern auch als eine "'Informationelle Handlungsabsicherung" bezeichnet (vgl. [Hen08, S. 20]). Die Abbildung 2.1 visualisiert diese Verhältnisse.

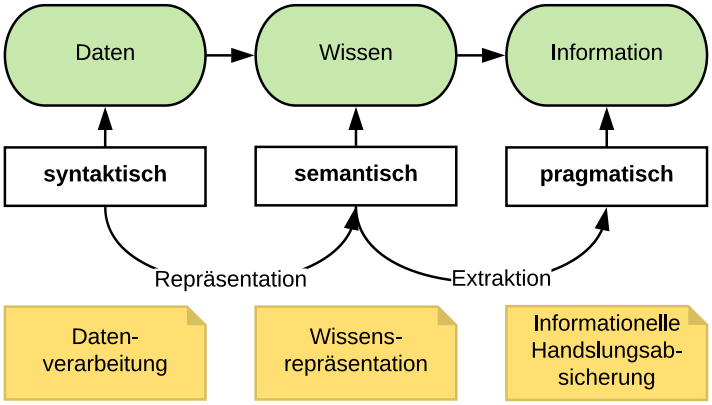

Abbildung 2.1: Abgrenzung der Begriffe Daten, Wissen und Information (aus: [Hen08, S. 20])

Retrieval Laut dem Pons Online Wörterbuch hat der englische Begriff *retrieval* im Informatikbereich mehrere Bedeutungen:

1. Daten-/Informationsabruf
2. Daten-/Informationsrückgewinnung
3. Wiederauffinden von Daten/Informationen (vgl. [Pon18])

Dadurch wird der zentrale Aspekt dieses Begriffs deutlich: Das Extrahieren von Informationen aus einer großen Datenmenge.

Wenn dies jetzt mit dem Begriff der *Information* verknüpft wird, kann zusammenfassend gesagt werden, dass *Information Retrieval* die "'Wiedergewinnung von Informationen, die von jemandem in einer konkreten Situation zur Lösung von Problemen benötigt werden", meint (vgl. [Hen08, S. 21]).

2.1.1.2 Anwendungen

Information Retrieval ist ein Oberbegriff für eine Menge verschiedener Forschungsfelder, die sich mit der Verarbeitung, Darstellung und Manipulation natürlicher Sprache auseinander setzen. Es ist oft schwierig, eine Anwendung einem konkreten Gebiet zuzuweisen, da Konzepte aus den verschiedenen Gebieten kombiniert werden.

1. Suchmaschinen sind Programme, die es ermöglichen nach und in Dokumenten zu recherchieren. Meistens ist mit diesem Begriff eine *Internet Suchmaschine* gemeint, die einen schnellen Weg bereitstellt, Internetseiten im *World Wide Web* zu finden. Diese sind für die meisten Nutzer, die Informationen im Internet suchen, zur primären Anlaufstelle geworden (vgl. [BCC16, S. 2ff]).

2. Document Filtering Systeme gehen den umgekehrten Weg eines typischen IR Prozesses. Hierbei werden vorgegebene Filter auf neu erstellte oder entdeckte Dokumente angewandt, um diese auf die Ansprüche des Nutzers anzupassen. Ein Beispiel ist das Aussortieren von Spam E-Mails, wie es von den größeren E-Mail-Kontoprovidern, wie beispielsweise Google Mail[1] standardmäßig zur Verfügung gestellt wird (vgl. [BCC16, S. 3]).

3. Information Extraction Systeme identifizieren Beziehungen zwischen *Entities* wie beispielsweise Orte, Zeiten, Personen, Zahlen oder Produkte in einem Satz oder Text. Auf diese Weise kann die Semantik eines Textfragments erschlossen werden. Abbildung 2.2 zeigt, wie dies mithilfe des Python Models spaCy[2] Entities aus einem Satz extrahiert werden können. Hierfür wurde eine Webapplikation benutzt, die von *Explosion AI* herausgegeben wird und die Entitäten von Texten grafisch

[1] https://mail.google.com (18.09.2018)
[2] https://spacy.io/ (20.09.2018)

aufbereitet.[3](vgl. [BCC16, S. 5])

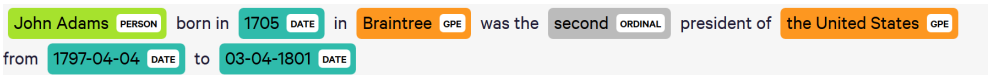

Abbildung 2.2: Entität-Erkennung mithilfe des *displaCy Named Entity Visualizer*

4. Clustering and Categorization Systeme ermöglichen es, Dokumente mit ähnlichen Eigenschaften zu gruppieren und zu kategorisieren. Musikstreaminganbieter wie *Spotify*, *Apple Music* oder *Deezer* nutzen Algorithmen um ähnliche Musik automatisch in gemeinsamen Playlisten zusammenzufassen (vgl. [BCC16, S. 4]).

5. Text Summarization Systeme fassen Texte zusammen. Beispielsweise können auf diese Weise die wichtigsten Informationen aus journalistischen Beiträgen extrahiert werden.[4] (vgl. [BCC16, S. 5]):

6. Frage-Antwort Systeme können natürliche Fragen beantworten. WolframAlpha besitzt beispielsweise ein Modul zum Erkennen der Absicht in Texten von natürlicher Sprache und kann einfachere Fragen wie "Who was the second president of the US?" zielgerichtet beantworten (siehe Abbildung 2.3 und vgl. [BCC16, S. 5]).

7. Multimedia Information Retrieval Systeme weiten die klassischen *Text Procession* Ansätze auf weitere Medientypen wie beispielsweise Bilder, Musik, Videos oder Sprache aus (vgl. [BCC16, S. 5]).

8. Expert Search Systems identifizieren Experten in einem bestimmten Gebiet. Häufig wird dies in Unternehmen angewandt, um aus einer Datenbank mit den Fähigkeiten von Mitarbeitern, die Experten der jeweiligen Gebiete zu erkennen (vgl. [BCC16, S. 5]).

[3] https://explosion.ai/demos/displacy-ent (17.08.2018)
[4] In Zusammenarbeit mit Fujitsu ist eine japanische Zeitschrift bereits diesen Schritt gegangen https://www.thesplicenewsroom.com/shinano-ai-summaries/ (23.07.18)

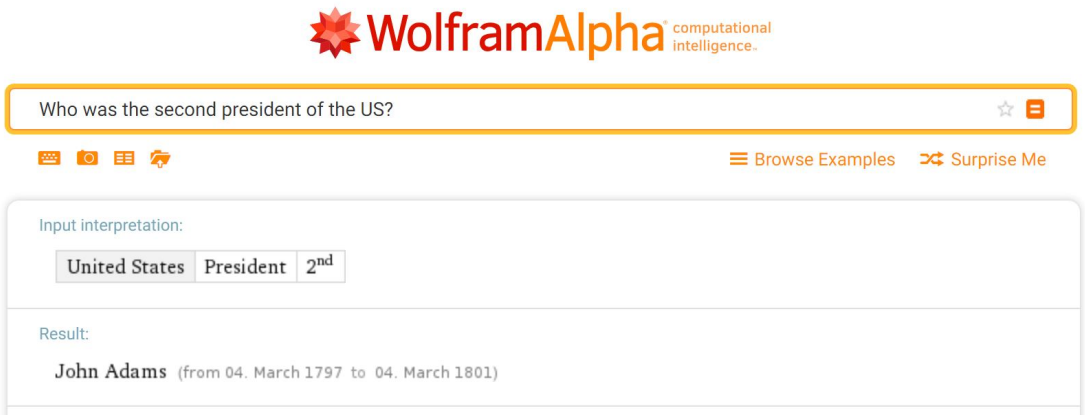

Abbildung 2.3: *WolframAlpha* kann in natürlicher Sprache gestellte Fragen beantworten

2.1.2 Architektur einer Suchmaschine

In diesem Abschnitt wird die Architektur und die einzelnen Komponenten eines speziellen IR Systems betrachtet: der *Suchmaschine*. Nach Croft et al. [CMS11, S. 14] besteht eine Suchmaschine aus zwei Hauptprozessen: Dem *Indexier-* und *Abfrage*prozess.

Der Indexierprozess strukturiert die Daten derart, dass eine effiziente Suche auf diese ermöglicht wird. Hierfür wird meist ein *Invertierter Index* erstellt. Der Abfrageprozess nutzt diese Datenstruktur, um Nutzerabfragen (*Queries*) zu beantworten und eine nach Wichtigkeit sortierte Liste mit Dokumenten zurückzuliefern.

2.1.2.1 Der Prozess des Indexierens

Der Indexierprozess, der in Abbildung 2.4 dargestellt ist, besteht aus drei Komponenten, die gemeinsam daran beteiligt sind, eine Datenbasis zu verarbeiten und schnell durchsuchbar zu machen.

Textbeschaffung (Text Acquisition) Die Hauptaufgabe der Textbeschaffungskomponente ist es, durchsuchbar zu machende Dokumente zu identifizieren und verfügbar zu

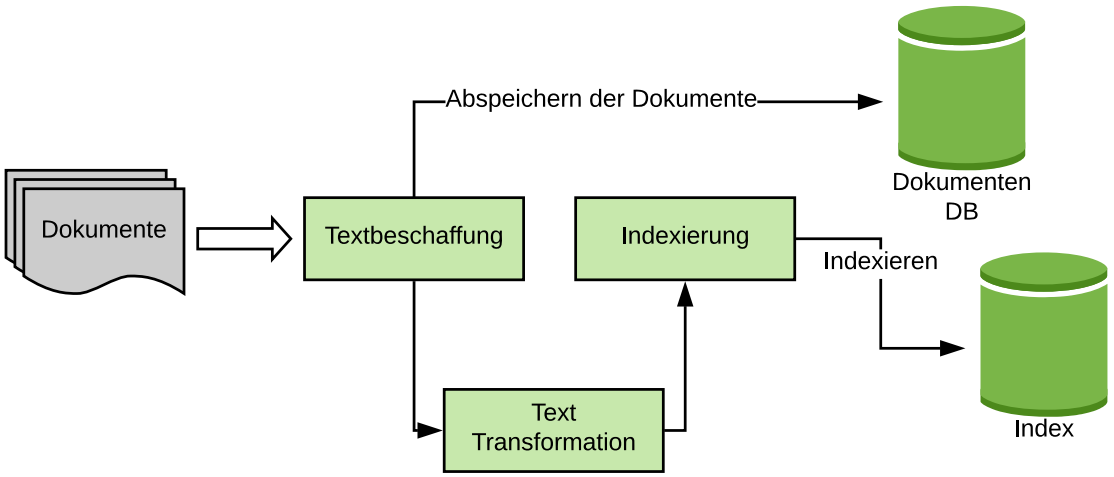

Abbildung 2.4: Der Prozess des Indexierens
(aus: [CMS11, S. 15]))

machen. Während in manchen Fällen die Daten bereits gesammelt vorliegen, müssen diese im Internet erst mit speziellen *Crawlern* gezogen werden. Ein Crawler ist hierbei ein Programm, das systematisch Quellen absucht und Informationen extrahiert. Das Unternehmen *Google* besitzt beispielsweise Cluster mit tausenden Rechnern, die nichts anderes machen, als periodisch neue oder upgedatete Internetseiten zu finden und sie zur Suche verfügbar zu machen. (vgl. [Dom13]) Weitere Aufgaben sind das Generieren von Metadaten aus den Dokumenten und das Speichern dieser in einer Datenbank.

Text Transformation Die Texttransformationskomponente erledigt die Vorarbeit der Indexerstellung. Hierbei wird der Text zuerst in *Tokens* eingeteilt, die dann mit speziellen Mitteln gefiltert werden, um eine bestmögliche Indexierung zu ermöglichen. In Kapitel 2.1.3 wird noch genauer darauf eingegangen.

Indexerstellung (Index Creation) Nachdem der Text transformiert wurde, wird er indexiert, damit eine schnelle Textsuche ermöglicht wird. Meistens wird hierfür ein *Invertierter*

Index verwendet, der im Abschnitt 2.1.3 noch genauer definiert und untersucht wird.

2.1.2.2 Der Prozess der Abfrage

Der Abfrageprozess geht nun die andere Richtung und stellt eine Schnittstelle zur Verfügung, um die indexierten Daten zu durchsuchen. Die Abbildung 2.5 stellt den Gesamtprozess dar.

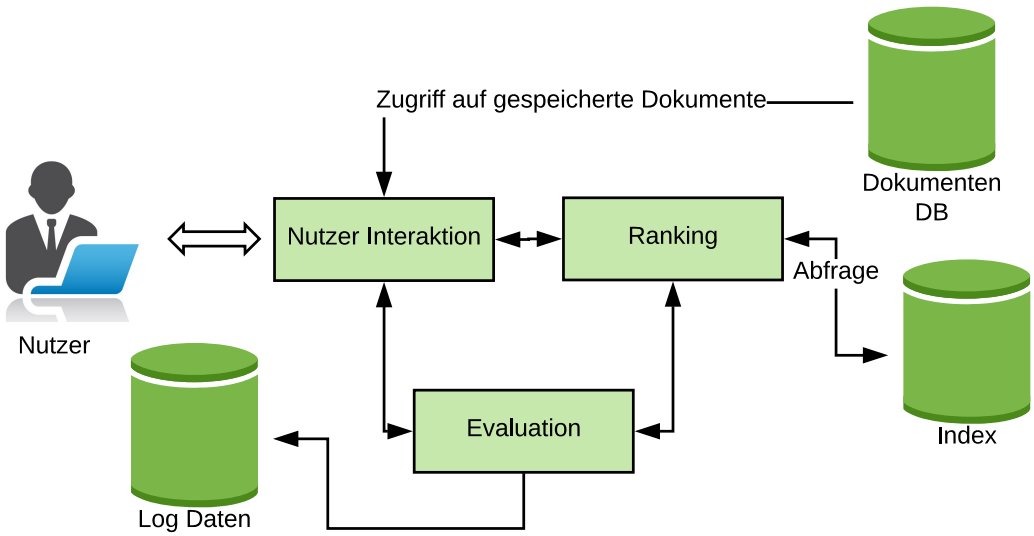

Abbildung 2.5: Der Prozess der Abfrage
(aus: [CMS11, S. 16])

Nutzerinteraktion (User Interaction) Die Nutzerinteraktion stellt die Hauptschnittstelle zur Suchmaschine zur Verfügung. Sie besteht aus Teilkomponenten zur Abfrageeingabe (*Query Input*), Abfragetransformation (Query Transformation) und der Ergebnisausgabe (Results Output).

Damit sind die Hauptaufgaben ein Entgegennehmen von *Queries*, sie in Deskriptoren (Index Terms) zu transformieren und an die Suchmaschine weiterzuleiten. Weiter gehört die Ausgabe und Aufbereitung zu den Zuständigkeiten dieser Komponente.

Ranking Das Ranking Modul ist der Kern jeder Suchmaschine. Es ist dafür zuständig, eine transformierte Nutzerabfrage zu verarbeiten und eine Rangliste mit den Ergebnissen zurückzuliefern. Spezielle Scoringfunktionen errechnen einen reellen Wert, aufgrund dessen die Wichtigkeit eines Ergebnisses definiert ist. Ferner sorgt es dafür, dass bei verteilten Indizes diese über das Netzwerk angesprochen werden. Das wird als *Query Broker* bezeichnet (vgl. [Hen08, S. 25ff]).

Evaluierung (Evaluation) Die Evaluierung sorgt dafür, dass Abfragen und Verhalten der Nutzer aufgezeichnet und ausgewertet werden. In Logfiles wird beispielsweise gespeichert, welche Ergebnisse der Nutzer anklickt oder wie lange er diese betrachtet, damit die Ergebnisqualität zukünftig weiter verbessert werden kann.

Neben einer kontinuierlichen Performanceanalyse eines Systems, werden in diesem Modul verschiedene Scoringfunktionen miteinander verglichen. Auf verschiedene Möglichkeiten der Evaluierung solcher Systeme wird in Kapitel 2.4 eingegangen.

2.1.3 Indexerstellung (Index Construction)

Der Prozess der Indexerstellung wird in diesem Abschnitt untersucht. Hierfür werden nacheinander die Schritte analysiert, die ein Dokument bis zu seiner finalen Indexierung durchläuft. Es wird davon ausgegangen, dass die Rohtexte bereits vorliegen und ein Textbeschaffungsprozess durchlaufen wurde.

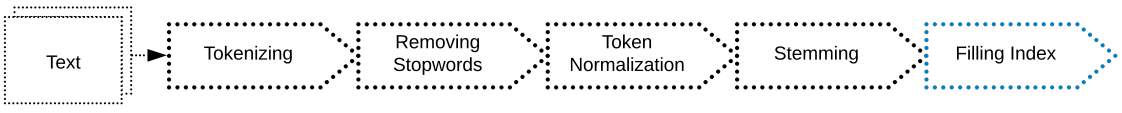

Abbildung 2.6: Schritte zur Erstellung eines invertierten Indexes

2.1.3.1 Tokenizing

Der Rohtext wird in *Token* eingeteilt. Laut Manning et al. (vgl. [MRS08, S. 22]) ist ein *Token* eine Folge von Zeichen, die als semantische Einheit für eine weitere Verarbeitung gruppiert wird. Tokens können beispielsweise Wörter, Sätze oder Silben usw. sein. Der Token*typ* ist die Klasse, die alle Tokens mit der gleichen Zeichenfolge beinhaltet.

Um sich das besser vorstellen zu können, hier ein Beispiel für ein *Tokenizing*. Der folgende Text[5]

`Et tu, Brute! Then fall, Caesar.`

wird bei einer strikten Trennung nach Leerzeichen und Satzzeichen zu:

`Et` `tu` `,` `Brute` `!` `Then` `fall` `,` `Caesar` `.`

Der naive Ansatz `1-Wort-1-Token` stößt jedoch sehr schnell an seine Grenzen. Beispielsweise stellt sich die Frage, wie Wörter mit Apostrophen zerlegt werden sollen. Soll ein englisches *aren't* in aren ' t aufgespalten werden, was intuitiv nicht richtig aussieht oder als arent, aren t oder aren't?

Solche Probleme sind von Sprache zu Sprache verschieden und müssen bei der Implementation eines IR Systems bedacht werden. Für westliche Sprachen gibt es verschiedene *Tokenizer*, die verwendet werden können. Zwei davon sind in Tabelle 2.1 aufgelistet.

Dieser Schritt kann noch mit einem *Stripping* kombiniert werden. In diesem Zuge werden zuerst einzelne Zeichen, wie Satzzeichen oder Wortketten (z.B. HTML Tags), die für den Text selbst unwichtig sind, entfernt (vgl. [ela18b]).

[5] Zitat aus: *Julius Caesar, Akt III* von W. Shakespeare
"'Auch du, Brutus? Dann falle, Caesar."

Name	Beschreibung	Ergebnis
Wort-Punkt Tokenizer	Trennt Wörter durch Satzzeichen	`Et` `tu` `,` `Brute` `!` `Then` `fall` `,` `Caesar` `.`
Whitespace Tokenizer	Trennt Wörter anhand von Leerzeichen	`Et` `tu,` `Brute!` `Then` `fall,` `Caesar.`
	...	

Tabelle 2.1: Verschiedene Arten von Tokenizern

2.1.3.2 Entfernen von Stopwords

Um die spätere Suche effizienter zu machen, werden Wörter, die in einer Sprache gehäuft vorkommen und damit nach Indexierung nicht zur Suchqualität beitragen entfernt: Die *Stoppwörter*. Diese Wörter haben oftmals eine rein grammatikalisch-syntaktische Funktion und der Sinn des Satzes kann auch ohne sie verstanden werden. Beispiele im Deutschen wären die Artikel *der*, *die*, *das* oder (Possessiv)-Pronomina wie *sein* oder *mein*. Im Englischen sind es Wörter wie *the*, *to* oder *be*.

Der Kerngedanke ist es, Speicherplatz zu sparen. Die Stoppwörter werden mit indexiert und der Index beinhaltet jeweils eine Liste mit allen Vorkommen dieser Terme. Da aber Stoppwörter eine große Häufigkeit in einer Sprache aufweisen, kann dadurch der Index sehr wachsen (vgl. [MRS08, S. 27]).

Leider sind Stoppwörter nicht so entbehrlich, wie es zuerst den Anschein hat. Viele Zitate wie "'`To be, or not to be`"[6] bestehen zu einem Großteil aus Stoppwörtern, oder bei einem "'`Flight to London`" würde nach einem entfernen des `to` nicht mehr ersichtlich, ob mit London nun die Abflug- oder Zielflughafen gemeint ist.

[6] Zitat aus: *Hamlet, Eröffnung Akt 3* von W. Shakespeare

Daher geht der Trend bei IR Systemen zurück auf kleinerere bis keine Stoppwortlisten. Die oben genannten Probleme überwiegen die Vorteile einer schnelleren Abfrageabarbeitung und einer reduzierten Indexgröße (vgl. [MRS08, vgl. S.27]). Auch in Trotman et al. wird dieses Ergebnis festgestellt:

"'*This investigation examined 9 ranking functions, 2 relevance feedback methods, 5 stemming algorithms, and 2 stop word lists. It shows that stop words are ineffective, that stemming is effective, that relevance feedback is effective, and that the combination of not stopping, stemming, and feedback typically leads to improvements on a plain ranking function. However, there is no clear evidence that any one of the ranking functions is systematically better than the others.*" [TPB14, S. 7ff]

2.1.3.3 Token Normalization

Im besten Fall stimmen die Token einer Abfrage mit denen im Index überein. Problematisch wird es jedoch, wenn beispielsweise nach Wörtern mit lexikalisch gleicher Bedeutung (Synonyme) gesucht wird. Der Nutzer erwartet beispielsweise, dass eine Suche nach `Fahrstuhl` annähernd die gleichen Ergebnisse, wie eine Suche nach `Lift` liefert.

Es gibt zwei Ansätze, dieses Problem zu lösen: *Äquivalenzklassen* und *Query Expansion* (vgl. [MRS08, S. 29]). Bei Erstem werden die Synonyme auf einen Oberbegriff abgebildet. Das heißt Wörter wie `U.S.A` oder `US` werden unter dem gemeinsamen Eintrag `USA` indexiert.

Die zweite und flexiblere Möglichkeit ist es, Beziehungen zwischen den einzelnen Tokens in speziellen Expansionslisten zu verwalten. Bei einer Abfrage wird nun auf diese Liste zurückgegriffen und nachgeschaut, welche alternativen Suchbegriffe noch in Betracht gezogen werden müssen. Statt eine externe Liste zu nutzen, können die äquivalenten Terms auch selbst indexiert werden.

2.1.3.4 Stemming

Oftmals verändern sich die Formen von Wörtern je nach grammatikalischem Einsatz. Im Lateinischen oder Deutschen werden beispielsweise Kasus, Numerus und Genus eines Substantives stark durch sich ändernde Wortendungen ausgedrückt.

Damit man nicht auf diese einzelnen Formen festgelegt ist, durchlaufen Wörter vor dem Indexieren einem *Stemming*-Prozess. Hierbei werden die Wortendungen derart abgeschnitten, dass eine Art Stammform vorliegt. Eine Kombination mit einem Wörterbuch erlaubt es zusätzlich noch, bspw. unregelmäßige Verben abzufangen und somit den Einfluss von Flexionen zu reduzieren.

Es gibt verschiedenste Stemming-Algorithmen wie den *Lovins Stemmer* oder den *Paice Stemmer*. Bekanntester dürfte wohl der *Porter Stemmer* sein. Eine genauere Analyse und Vergleich dieser Algorithmen wird aber an dieser Stelle nicht vorgenommen.

2.1.3.5 Erstellung des Invertierten Index

Der *invertierte Index* ist die zentrale Datenstruktur in fast jedem IR System. Hierbei wird eine sortierte Liste mit Einträgen erstellt, die auf die jeweiligen Stellen des Vorkommens innerhalb der Dokumente verweisen und somit eine schnelle Suche ermöglichen.

Das bekannteste Beispiel ist der Index am Ende von Büchern. Hierbei wird einer alphabetischen Liste mit den wichtigsten Wörtern im Buch die entsprechende Seitenzahl des Vorkommens zugeordnet. Abbildung 2.7 zeigt einen Ausschnitt aus dem Werk von Manning. ([MRS08, S. 544])

Definition 2.1.1 (Invertierter Index)**.** Laut Manning et al. [MRS08, S.6] besteht ein *Invertierter Index* aus zwei Komponenten: Einem *Dictionary* und den *Postings*.

Das *Dictionary* beinhaltet eine Liste aus *Schlüssel-Werte*-Paaren, wobei die *Schlüssel* die Terme und die Werte jeweils einen Verweis auf eine sogenannte *Posting-List*

top-down clustering, **395**
topic, **153**, **253**
  in XML retrieval, 211
topic classification, **253**
topic spotting, **253**
topic-specific PageRank, **471**
topical relevance, **212**
training set, **256**, 283
transactional query, **433**
transductive SVMs, **336**

XML element, **197**
XML fragment, **216**
XML Schema, **199**
XML tag, **197**
XPath, **199**

Zipf's law, **89**
zone, **110**, 337, 339, 340
zone index, 110
zone search, 197

Abbildung 2.7: Auszug aus einem Buchindex
(Auszug aus: [MRS08, S.544])

darstellen. Die Menge aus Termen wird auch als *Vocabulary* bezeichnet. Das *Dictionary* hingegen bezieht sich auf die gesamte Datenstruktur.

Diese *Posting Listen* speichern, in welchen Dokumenten und Positionen die einzelnen Terme auftreten. Die Menge aller *Posting-Listen* wird nur als *Postings* bezeichnet.

Jede konkrete Implementierung eines *Invertierten Indexes* besitzt vier Hauptfunktionen, die von Büttcher et. al. [BCC16, S. 33ff] folgendermaßen definiert werden:

- **first(t)** liefert die erste Position, an der der Term t in einer Sammlung aus Dokumenten auftaucht.

- **last(t)** liefert die letzte Position, an der der Term t in einer Sammlung aus Dokumenten auftaucht.

- **next(t, current)** liefert das nächste Auftauchen eines Terms t nach der *current* Position

- **prev(t, current)** liefert das vorherige Auftauchen eines Terms t nach der *current* Position

Wird ein Wert nicht gefunden, wird der unbestimmte Wert Ω zurückgeliefert. Im folgenden werden diese theoretischen Grundlagen anhand eines Beispiels näher erläutert.

Beispiel Hierfür folgt ein Beispiel, bei dem der Text von 37 Schauspielen des englischen Dramatiker *Shakespeare* indexiert wurde. Dabei wurden die Tokens nach einem Tokenizing (siehe 2.1.3.1) in ein *Dictionary* geschrieben und eine Verknüpfung zu den Startpositionen der Wörter hergestellt. Dies ist in Abbildung 2.8 dargestellt. Auf der Linken Seite findet sich das *Vocabulary* - die Liste der Terms - und auf der Rechten Seite die *Postings* mit den einzelnen *Posting-Listen*.

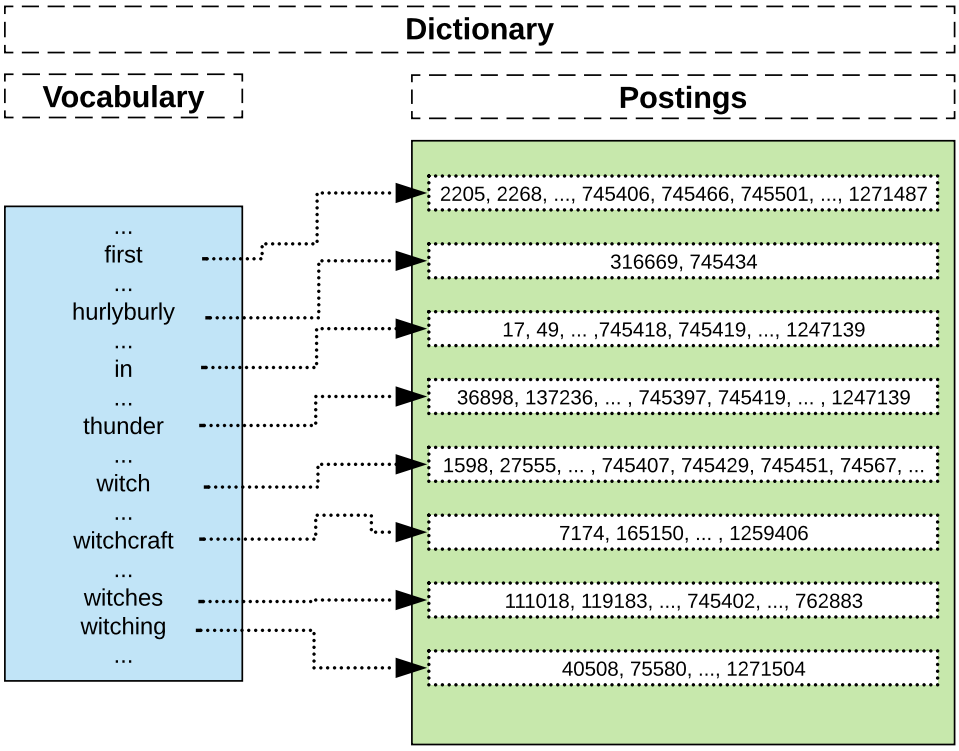

Abbildung 2.8: Auszug aus einem invertierten Index der 37 Werke von *William Shakespeare*
(aus: [BCC16, S. 37])

Im folgenden sind ein paar Beispiele dargestellt, wie die Funktionen $find(k)$, $first(k)$,

$last(k)$, $next(k,v)$, $prev(k,v)$ auf dem Index operieren.

$$\begin{aligned} i.first('first) &= 2205 \\ i.last('witches') &= 762883 \\ i.first('Gibtsnicht') &= \Omega \\ i.next('first', i.first('first')) &= 2268 \\ &[...] \end{aligned} \tag{2.1}$$

Eine weiterführende Einführung in die Implementation von *Invertierten Indizes* findet sich bei Büttcher et al. [BCC16, S. 35-66].

2.1.4 Scoring Funktionen

Diese Sektion zeigt, wie relevante Ergebnisse gefunden werden. Hierfür wird nur der in 2.5 dargestellter Ranking Prozess betrachtet.

Um die Relevanz von Dokumenten zu bestimmen, die bei einer Abfrage gefunden werden, wernde *Scoring Funktionen* verwendet. Hierfür wird eine Funktion definiert, die einem gegebenen Dokument in einer *Collection* einen reellen Wert auf eine Abfrage zuweist. Eine *Collection* beschreibt hier die Gesamtheit aller Dokumente, auf die eine *Query* q durchgeführt wird.

2.1.4.1 TF/IDF Modell

In der einfachsten Form berechnet sich der score eines Dokuments d für eine Query q folgendermaßen:

$$\text{score}_d = \sum_{t \in q} \text{tf}_{td}$$

mit dem Dokument d und der *Term Frequency* tf_{td}. Es wird gezählt, wie oft die einzelnen Terme in einem Dokument vorkommen. Je häufiger er beinhaltet ist, desto wichtiger ist das Dokument.

Eine Query q besteht aus mehreren Suchtermen. Wenn ein Term in vielen Dokumenten auftritt, dann sollte der Einfluss dieses Terms in der gesamten Suche nach unten skaliert werden. Beispielsweise tauchen die Artikel *der*, *die* und *das* in vielen Dokumenten sehr oft auf. Würde man nur durch das Auftreten dieser Wörter in einem Dokument dessen Relevanz berechnen, dann hätten diese Terme einen zu großen Einfluss auf den Gesamtscore. Damit würden seltenere Wörter (z.B. Fachbegriffe) weniger ins Gewicht fallen.

Eine Lösung ist es, die relative Häufigkeit der Terms t in der ganzen *Collection* mit einzubeziehen (vgl. [MRS08, S. 118]). Die *Collection* bezeichnet hierbei die Menge aller Dokumente des Systems:

$$\mathrm{idf}_{td} = \log_2 \frac{N}{\mathrm{df}_t} = -\log_2 \frac{\mathrm{df}_t}{N}$$

mit Dokumentanzahl N. df_t beschreibt hierbei die Anzahl der Dokumente, in denen der Term t auftritt. (document frequency).

Das wird als *Inverse Document Frequency* (idf) bezeichnet. Die Idee ist es, jedem Term ein Gewicht zu geben, dass dadurch definiert wird, in welchem Anteil an Dokumenten dieser auftritt. Ein Term, der in vielen Dokumenten vorkommt, wird so weniger ins Gewicht fallen.

Der Score eines einzelnen Terms in einer Query q kann als Produkt von IDF und TF berechnet werden und wird als $\mathrm{tf} - \mathrm{idf}_{t,d}$ bezeichnet.

$$\mathrm{tf} - \mathrm{idf}_{td} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t$$

Um nun den Score einer Abfrage q für ein Dokument d zu berechnen, werden die

einzelnen tf-idf Werte der Terme t von q summiert:

$$\text{score}(q,d) = \sum_{t \in q} \text{tf} - \text{idf}_{t,d}$$

2.1.4.2 Okapi BM25

Okapi BM25 ist eine Ranking Funktion, die seit 2016 der Standard ab Elasticsearch 5.0 (2.2.2) ist (vgl. [ela16]). BM25 erweitert die Definition von TF/IDF und erlaubt eine Parametrisierung.

Hierfür wird zuerst die $\text{idf}_{t,d}$ im Unterschied zu Punkt 2.1.4.1 folgendermaßen definiert:

$$\text{idf}_{t,d} = \log_2 \left(1 + \frac{N - \text{df}_t + 0.5}{\text{df}_t + 0.5}\right)$$

mit der Anzahl der Dokumenten N. Die *document Frequency* df_t, gibt an, in wie vielen Dokumenten der Term t vorkommt (vgl. [Con18]).

Der $\text{tf}_{t,d}$ wird ebenfalls neu definiert, um eine Parametrisierung zu ermöglichen:

$$\text{score}(q,d) := \sum_{t \in q} \text{idf}_{t,d} \cdot \frac{(k_1+1)\text{tf}_{td}}{k_1((1-b) + b \times (L_d/L_{\text{ave}})) + \text{tf}_{td}}$$

mit Tuningparameter $k_1 \in \mathbb{R}$ und $b \in [0;1]$, Dokumentenlänge L_d und durchschnittlicher Dokumentenlänge L_{ave}. Die *term frequency* tf_{td} ist hierbei die Anzahl an Vorkommnissen des Terms t im Dokument d.

Im letzten Schritt wird die Häufigkeit der einzelnen Terme in der Query q mit einbezogen. Terme, die in q häufiger auftauchen, werden stärker gewichtet, wobei eine Grenze festlegt, ab welcher Häufigkeit der Effekt gesättigt ist.

$$\text{score}(q,d) = \sum_{t \in q} \text{idf}_{t,d} \cdot \frac{(k_1+1)\text{tf}_{td}}{k_1((1-b)+b \times (L_d/L_{\text{ave}})) + \text{tf}_{td}} \cdot \frac{(k_3+1)\text{tf}_{tq}}{k3 + \text{tf}_{tq}}$$

mit Tuningparameter $k_3 \in \mathbb{R}$ und *Term Frequency* tf_{tq} in einer Query q. k_3 ist der Sättigungswert dieses Faktors und hat denselben Effekt wie der Parameter k_1. Darauf wird im folgenden Paragraphen eingegangen. *Elasticsearch* verzichtet jedoch auf diesen letzten Faktor (vgl. [Con18]).

Auswirkungen der Parameter k_1 und b auf den Score

Im folgenden wird der zweite Faktor

$$\frac{(k_1+1)\text{tf}_{td}}{k_1((1-b)+b \times (L_d/L_{\text{ave}})) + \text{tf}_{td}}$$

näher betrachtet und es wird untersucht, wie sich die Parameter k_1 und b auf die Suche auswirken.

Zunächst gibt L_d/L_{ave} an, wie lange ein Dokument relativ zur durchschnittlichen Dokumentenlänge ist. Dieses Verhältnis wird in den Nenner platziert, was zur Folge hat, dass je mehr das Dokument über der durchschnittlichen Länge liegt, desto kleiner wird der Faktor und umgekehrt. Die Idee dahinter liegt ist, dass längere Dokumente weniger gewichtet werden, denn dort ist es normal, dass ein Term häufiger auftritt. Sollte das Dokument exakt im Durchschnitt liegen, fällt dieser Faktor weg. Der Parameter b wird benutzt, den Einfluss dieses Faktors zu skalieren. Dies wird als *Field Length Normalization* bezeichnet (vgl. [ela18f]).

Um den Einfluss von k genauer zu untersuchen, wird zunächst folgende Funktion f definiert:

$$f(\mathrm{tf}_{td}) = \frac{\mathrm{tf}_{td}(k_1+1)}{\mathrm{tf}_{td} + k_1 * \mathrm{fln}}$$

mit der konstanten *Field Length Normalization* $\mathrm{fln} = (1-b) + b \times (L_d / L_{ave})$.

Mithilfe der Regel von L'Hopital kann der Grenzwert dieser Funktion berechnet werden:

$$\lim_{\mathrm{tf}_{td} \to \infty} f(\mathrm{tf}_{td}) \overset{L'Hopital}{=} \frac{k_1+1}{1} = k_1 + 1$$

Damit ist der Parameter k_1 eine waagrechte Asymptote der Funktion f. Wenn ein Wort öfters in einem Dokument vorkommt, würde der score unbegrenzt wachsen. $k_1 + 1$ stellt eine Möglichkeit dar, das Wachstum zu beschränken und einen Sättigungswert zu definieren.

Elasticsearch nutzt hierfür einen Wert von $k_1 = 1.2$. Dies kann folgendermaßen interpretiert werden: Wenn ein Wort ein oder zweimal in einem Dokument vorkommt, dann macht dies einen Unterschied. Kommt das Wort noch öfters vor, hat das keinen weiteren Einfluss auf den Gesamtwert.

2.1.4.3 Divergence from Randomness (DFR)

DFR ist kein Modell im eigentlichen Sinn, sondern ein Framework, aus dem Modelle abgeleitet werden. Es wurde von Amati und van Rijsbergen 2002 vorgestellt (vgl. [Net16]).

"'Die Idee ist, Begriffsgewichte zu berechnen, indem die Abweichung zwischen einer durch einen Zufallsprozess erzeugten Begriffsverteilung (innerhalb der Sammlung) und der tatsächlichen Begriffsverteilung (innerhalb des Dokuments) gemessen wird." [Net16, trans. S.2]

Der Score eines Dokumentes berechnet sich wieder aus der *term frequency*, die mit einem Gewicht multipliziert wird (vgl. [Net16]):

$$\text{score}(q,d) = \sum_{t \in q} \text{tf}_{tq} \times w_{t,d}$$

Hierbei stellt tf_{tq} die Termhäufigkeit eines Terms t in einem Dokument d dar. Das Gewicht w ist das Produkt aus zwei Wahrscheinlichkeitsfunktionen. Eine genauere Beschreibung dieses Modells findet sich bei *Neto* [Net16].

2.1.4.4 Divergence From Independence

Die parameterlose Gewichtsfunktion *Divergence from Independence* (DFI) wurde 2012 im Zuge eines TREC[7] Web Wettbewerbs entwickelt und veröffentlicht.

Zunächst wird die die *erwartete Häufigkeit* e_{td} eines Terms t im Dokument d definiert:

$$e_{td} = \frac{\text{TF}_t \times L_d}{N}$$

Umso länger ein Dokument ist, desto häufiger wird ein bestimmter Term auch erwartet. TF wird als *Collection Frequency* bezeichnet, gibt also an, wie oft ein Term in der gesamten *Collection* C vorkommt ($\text{TF}_t := \sum_{d \in C} \text{tf}_{td}$). Der Score s eines Dokumentes d wird folgendermaßen berechnet (vgl. [Din12, S. 5]):

$$\text{score}(q,d) = \sum_{t \in q} \text{tf}_{td} \times w_{td} \times \Lambda_{td}$$

mit der Gewichtsfunktion

$$w_{td} = log_2 \left(\frac{\text{tf}_{td} - e_{td}}{\sqrt{e_{td}}} + 1 \right)$$

für $\text{tf}_{td} - e_{td} > \log_2 (0+1) = 0$, ansonsten 0. Diese Definition des Gewichtes wird auch

[7] Text REtrieval Conference: Konferenzen zur Förderung und Unterstützung der Forschung im Bereich *Information Retrieval*

standardization genannt (vgl. [Din12, S. 2]). Das Gewicht wird logarithmisch erhöht, je öfter der Term in einem Dokument vorkommt. Das bedeutet, dass der Unterschied, ob der Term ein oder zweimal über der erwarteten Häufigkeit auftritt, größer ist, als beispielsweise zwischen dem zehnten und elften Auftreten über der erwarteten Häufigkeit. Vorausgesetzt ist jedoch, dass der Term in einem Dokument mindestens seiner *erwarteten Häufigkeit* entspricht, denn sonst ist das Gewicht null und das ganze Dokument fällt als Ergebnis weg.

Um Dokumente, in denen zum einen Terme relativ zur Dokumentenlänge selten oder Begriffe unabhängig von der Dokumentenlänge häufig vorkommen, zu fördern, wird folgende Funktion Λ definiert:

$$\Lambda_{td} = \alpha_{td}^{3/4} \times \beta_{td}^{1/4} = \left(\frac{L_d - \mathrm{tf}_{td}}{L_d}\right)^{3/4} \times \left(\frac{2}{3} \times \frac{\mathrm{tf}_{td} + 1}{\mathrm{tf}_{td}}\right)^{1/4}$$

Die Konstante $2/3$ wurde als Ergebnis von *Try-and-Fail* Experimenten bestimmt (vgl. [Din12, S. 4]). Die Exponenten $3/4$ und $1/4$ haben sich als gute Werte herausgestellt (vgl. [Din12, S. 4]). Eine weitere Beschreibung und Analyse dieser Methodik ist in [Din12] zu finden.

2.2 Open-Source Suchmaschinen auf Lucene Basis

Einige Open-Source Projekte haben es sich zur Aufgabe gemacht, Suchmaschinenplattformen zu entwickeln. Zwei Softwarebibliotheken, die auf die Suchbibliothek *Apache Lucene* setzen werden nun vorgestellt: *Elasticsearch* und *Solr*.

Apache Lucene ist eigenen Angaben zufolge eine "'High-Performance, full-featured text search engine library written in Java" [Apa16] unter der *Apache License 2.0*.

Damit ist *Lucene* eine Softwarebibliothek, die sich auf das Indexieren und Durchsuchen

Abbildung 2.9: Logo von Apache Lucene

von Texten konzentriert und die Basis für viele Suchmaschinenbibliotheken.

Lucene wurde 1997 auf dem Open-Source Portal *Sourceforge* von *Doug Cutting*[8] veröffentlicht. 2001 wurde das Projekt in die *Jakarta* Familie der *Apache Software Foundation* (ASF) aufgenommen und hat seitdem stark an Popularität gewonnen. 2005 wurde das *Apache Solr* Projekt mit *Lucene* verschmolzen (vgl. [MHG10]).

Der Suchmaschinenkern wird heute von Unternehmen wie *Apple*, *Google*, *IBM* und Co. genutzt. [Hei11, vgl.]. Aktuell im Juli 2018 liegt *Lucene* in der Version 6.6.5 vor.

2.2.1 Apache Solr

Abbildung 2.10: Logo von Apache Solr

Apache Solr[9] ist eine auf *Lucene* aufbauende Suchplattform und ein Server-Wrapper, der eine Representational State Transfer (REST)[10] Schnittstelle zur Verfügung stellt. Außerdem erweitert es *Lucene* um Funktionen für Monitoring und Administration, Optimierungen für hohen Netzwerk Traffic und ein Plugin System (vgl. [Apa17]).

Abbildung 2.11: Logo von *Elasticsearch*

2.2.2 Elasticsearch (ES)

Elasticsearch (ES) ist eine weitere Suchmaschine, die sich durch seine Schnelligkeit, Verteilt- und Skalierbarkeit auszeichnet. Ebenso stellt ES eine Serverschnittstelle zur Verfügung, mit deren Hilfe *Lucene* angesprochen wird. Im September 2018 liegt ES in der Hauptversion 6.4 vor. Die Kommunikation von außen erfolgt über eine REST Schnittstelle, die universell eingesetzt werden kann.

2.2.2.1 Geschichtlicher Hintergrund

Anfang der 2000er Jahre wurde *Elasticsearch* von *Shay Banon* als Suchmaschine für seine Frau, die als Köchin eine stetig wachsende Sammlung an Rezepten besaß, entworfen und unter einer Open-Source Lizenz veröffentlicht. Aufgrund vieler positiver Reaktionen wurde kurz darauf das Unternehmen *Elasticsearch Inc.* gegründet. 2015 wurde das Unternehmen in *Elastic* umbenannt und *Packetbeat* und die Toolfamilie *Beats* in das Unternehmen eingegliedert.

2012 wurden die damals noch unabhängigen Open-Source Projekte Logstash - Ein Tool zur Sammlung von *Logdateien* - , dem Visualisierungswerkzeug *Kibana* und *Elasticsearch* miteinander verschmolzen. Seitdem erfuhr die Software eine wachsende Beliebtheit. Im

[8] Trivia: *Lucene* ist der zweite Vorname seiner Frau
[9] Ausgesprochen "'Solar"
[10] https://de.wikipedia.org/wiki/Representational_State_Transfer (19.09.2018)

Februar 2018 wurde die Software 270 Millionen mal heruntergeladen (vgl. [ela18e]).

2.2.2.2 Logisches Layout

Dieser Abschnitt untersucht, wie ES auf Applikationsebene aufgebaut ist. ES setzt sich aus *Dokumenten*, *(Mapping) Typen* und *Indizes* zusammen.

Dokumente Ein Dokument ist die "'kleinste Einheit von Daten, die [man] indexieren und durchsuchen kann." (vgl. [GHR14, S. 26]). Ein Dokument liegt im JSON (Java Script Object Notation) Format vor und besteht aus einer Sammlung von *Key-Value*-Paaren, wobei der *Key* der Name eines Feldes und der *Value* der tatsächliche Wert ist.

Wichtige Eigenschaften von ES Dokumenten sind:

- *Eigenständigkeit*: Ein Dokument beinhaltet immer Feldnamen **und** zugehörigen Wert.
- *Hierarchisch*: Dokumente können aus Unterdokumenten bestehen. Beispielsweise werden geographische Koordinaten in Längen-und Breitengrad aufgeteilt.
- *Flexibilität/Schemafreiheit*: Dokumente sind nicht auf ein festes Schema festgelegt. Dokumente müssen nicht zwingend die in *Mappings* definierten Feldern beinhalten und sind daher nicht an eine vorgegebene Struktur gebunden (vgl. [MHG10, S. 28]).

Code 2.1 zeigt ein Beispiel für ein Dokument: Die Tragödie *Macbeth* ([Sha86, S.1]) von W. Shakespeare besitzt vier Felder: *title*, *author*, *type* und *content* und stellt ein einzelnes Dokument dar.

Mapping Types Der *Mapping Type* definiert, *wie* Dokumente in ES indexiert werden. Meistens wird dieser nur als *Type* bezeichnet. Er definiert die Namen und Eigenschaften (z. B. Datentyp) der Felder eines Dokuments. Meistens werden Typen gleichzeitig zur Indexerstellung mit angegeben.

Code 2.1: Macbeth als JSON Dokument

```
{
        "title" : "The Tragedy of Macbeth",
        "author" : "William Shakespeare",
        "type" : "play",
        "content" : "ACT I SCENE I. A desert place [...]"
        ...
}
```

Indizes ES betrachtet den Index als einen "'Container für Mapping Typen" (vgl [GHR14]). Ein Index ist laut Tong vergleichbar mit einer Datenbank in einem relationalem Datenbanksystem:

"'*An index is like a 'database' in a relational database. It has a mapping which defines multiple types. An index is a logical namespace which maps to one or more primary shards and can have zero or more replica shards.*" [Ton18]

Indizes können große Datenmengen beinhalten, die nicht auf einmal gespeichert werden können. *Shards* bieten die Möglichkeit, einen Index in mehrere Teilindizes aufzuspalten, die jeweils einen voll funktionsfähigen und unabhängigen Index darstellen und auf jeden Knoten in einem Cluster gehostet werden können (vgl. [ela18a])

Die Abbildung 2.12 stellt diese Zusammenhänge dar. Ein ES Server besteht aus mehreren Indizes, wie hier im Beispiel *Literatur* und *Musik*, die dann die Dokumente von verschiedenen Typen beinhalten. Ein Typ wird hierbei durch ein oder mehrere Felder definiert. Beispielsweise besitzt ein Buch die Felder *Autor* und *ISBN*, wohingegen bei einer Webseite anstatt der *ISBN* eine *URL* gespeichert wird.

2.2.2.3 Physikalisches Layout

Weiter wird nun der Aufbau von ES auf Administrationsebene betrachtet. Dies wird in Gheorghe et al. [GHR14, S. 30] beschrieben.

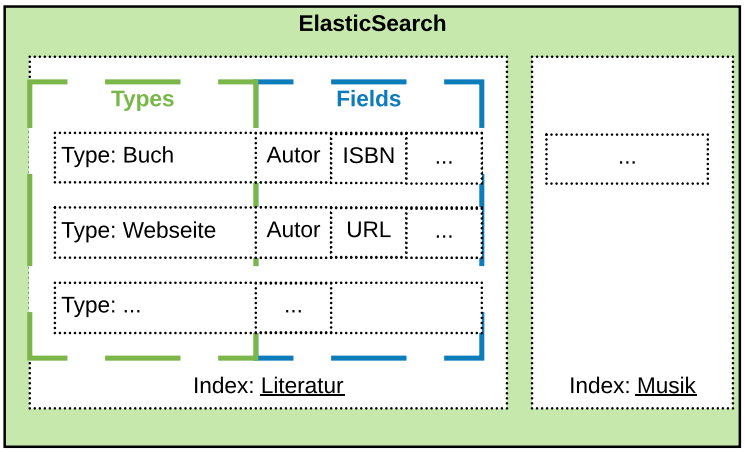

Abbildung 2.12: Sicht einer Applikation auf Elasticsearch

Nodes & Clustering Ein *Node* ist eine ES Instanz. Mehrere Nodes können zu einem *Cluster* zusammengefasst und somit die Daten auf mehrere Knotenpunkte verteilt werden.

Shards & Replikas Ein einzelner Index kann viel Speicherplatz verbrauchen. Daher ist es sinnvoll, diesen auf mehrere Teile, sogenannte *Shards*, aufzuteilen. Um Konsistenz zu gewährleisten, wird für jeden Shard zusätzlich mindestens ein *Replika* angelegt, der eine exakte Kopie darstellt und auf einem anderen Node abgespeichert wird.

Abbildung 2.13 zeigt, wie Indizes auf verschiedenen Nodes gespeichert und repliziert werden.

2.2.2.4 Kommunikation über eine REST API

Die Kommunikation mit dem ES Server erfolgt über eine automatisch angelegte REST Schnittstelle. Dies hat den Vorteil, dass offizielle Bibliotheken für alle gängigen Sprachen wie *Java*, *C#*, *Python*, *JavaScript* usw. existieren.
Diese Schnittstelle ermöglicht

- das Abrufen von Statistiken und Status des Clusters oder der Nodes

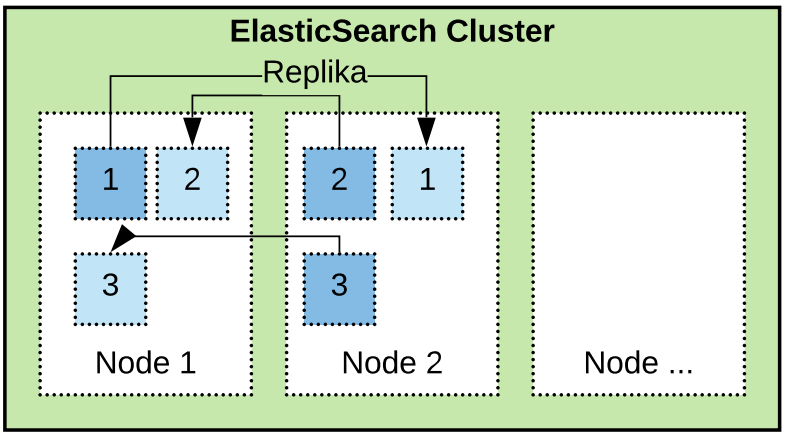

Abbildung 2.13: Physikalischer Aufbau eines ES Clusters

- das Verwalten des Clusters, der Nodes, Indexdaten oder Metadaten
- das Absetzen von Create-Read-Update-Delete (CRUD) Befehlen auf Indizes

2.2.2.5 Implementation von Suchmaschinenprozessen

Nun wird beschrieben, wie die Suchmaschinenprozesse, wie sie in 2.1.2 dargestellt wurden, umgesetzt sind.

Indexierung von Dokumenten

Um Daten indexieren zu können, wird zunächst ein Index angelegt. Sollte ein Index nicht explizit manuell definiert werden, wird dieser von ES automatisch angelegt (vgl. [ela18c]). Die Erstellung wird durch die *Index API* ermöglicht. Codebeispiel 2.2 zeigt, wie der in Abbildung 2.12 dargestellte Literaturindexaufbau mithilfe eines *HTTP POST-Requests* und der Nutzung des Kommandozeilentools *cURL* erstellt werden kann.

Elasticsearch erzeugt nun automatisch einen API Endpunkt für den neuen Index unter *localhost:9200/literature*[11], mit dem Dokumente indexiert werden können. Hierfür wird das

[11] von ES Hostadresse abhängig

Code 2.2: Erstellung eines Indexes

```
curl -X PUT "localhost:9200/literature"
        -H 'Content-Type: application/json'
        -d'
{
        "settings" : {
                "number_of_shards" : 1
        },
        "mappings" : {
                "Buch" : {
                        "properties" : {
                                "author" : { "type" : "text" },
                                "title" : { "type" : "text" },
                                "isbn" : { "type" : "text" },
                                [...]
                        }
                },
                "Website": { [...] }
        }
}'
```

Code 2.3: Indexierung eines Dokumentes

```
curl -X PUT "localhost:9200/literature"
        -H 'Content-Type: application/json'
        -d'
{
        "author" : "William Shakespeare",
        "title" : "The Tragedy of Macbeth",
        [...]
}'
```

zu indexierende Dokument als JSON an den ES Server übergeben. Dies wird in Codebeispiel 2.3 dargestellt. Hierfür wird in Zeile fünf das Buch mit dem Titel *The Tragedy of Macbeth* und dem Autor *William Shakespeare* indexiert.

Abfragen

Jeder Index besitzt einen weiteren API Endpunkt unter */indexname/_search*, über den Abfragen gestellt werden. Wenn kein spezifischer Index mit angegeben wird, werden automatisch alle durchsucht. In seiner minimalsten Form wird im HTTP Body nur ein *query*

Feld mit der Abfrage/den Suchbegriffen übergeben. Die in 2.2 dargestellt Abfrage liefert alle Dokumente in unserem Literatur-Index, die das Wort *Macbeth* im Titel beinhalten.

Code 2.4: Stellen einer Abfrage

```
curl -X GET "localhost:9200/literature"
        -H 'Content-Type: application/json'
        -d'
{
        "query": {
                "simple_query_string" : {
                        "fields" : ["title"],
                        "query" : "Macbeth"
                }
        }
}'
```

Anmerkung: Natürlich ist die API weitaus komplexer als hier in der Kürze dargestellt. Für weitere Informationen sei ein Blick in die offizielle Dokumentation Elasticsearch Reference[12] empfohlen.

2.2.2.6 ELK Stack

ELK ist die Abkürzung für *Elasticsearch*, *Logstash*, *Beats* und *Kibana*. Diese Tools werden oft miteinander kombiniert und stellen den *Elastic Stack* dar. (vgl.[ela18d])

- *Elasticsearch*: Kernsuchmaschine und Datenbank
- *Kibana*: Grafisches Datenanalysetool für ES Instanzen. Stellt zusätzlich Entwicklerwerkzeuge zur Verfügung
- *Logstash*: Tool zur Auswertung von Logdateien
- *Beats*: "'Shipper", die Daten von vorgegebenen Quellen importieren.

[12] https://www.elastic.co/guide/en/elasticsearch/reference

2.2.2.7 Ändern der Scoringfunktion eines Indexes

ES erlaubt es, verschiedene Scoringfunktionen, zum Beispiel die in Abschnitt 2.1.4 Vorgestellten, zu verwenden (vgl. [ela18h]).

Es gibt bereits fertige Implementationen für die Modelle *BM25*, *DFR*, *DFI*, *Information Based Model* (IB) oder *LM Dirichlet*. Zusätzlich ist es möglich, Skripte zu schreiben, die eigene Bewertungsfunktionen implementieren (vgl. [ela18h]). Um die Scoringfunktion eines bestehenden Indexes zu ändern, muss er zuerst geschlossen werden. Hierfür wird einfach der *_close* Endpunkt der Index API aufgerufen. Dies wird in Codebeispiel 2.5 dargestellt.

Code 2.5: Schließen eines offenen Indexes

```
1 curl -X POST "localhost:9200/literature/_close"
2         -H 'Content-Type: application/json'
```

Jetzt lässt sich das Modell ändern. Hierfür muss ein *HTTP PUT* Request an den Index geschickt werden, der die Indexeinstellungen ändert (siehe Codebeispiel 2.6).

1. In Zeile neun wird das standard Scoringmodell nach *BM25* geändert. Dazu werden die neuen Indexeinstellungen an die *Index API* gesendet.

2. Zusätzlich können Parameter gesetzt werden. In Zeile zehn wird beispielsweise der Parameter b auf 0.75 gesetzt. Diese sind oft optional (vgl. [ela18h]).

3. Nach erfolgreicher Änderung wird der Index in Zeile 16 wieder geöffnet.

2.3 Product Lifecycle Management (PLM)

Prof. Dr. Schuh, Leiter des Lehrstuhls für Produktionssystematik an der RWTH Aachen definiert den Begriff *PLM* folgendermaßen:

Code 2.6: Ändern der Scoringfunktion

```
curl -X PUT "localhost:9200/literature/"
        -H 'Content-Type:application/json'
        -d '
{
        "settings": {
                "index": {
                        "similarity": {
                                "default": {
                                        "type": "bm25",
                                        "b": "0.75"
                                }
                        }
                }
        }
}'
curl -X POST "localhost:9200/literature/_open"
        -H 'Content-Type:application/json'
```

Definition 2.3.1 (Product Lifecycle Management [Sch15]). *"'Beim Product Lifecycle Management (PLM) handelt es sich um einen Ansatz zur ganzheitlichen, unternehmensweiten Verwaltung und Steuerung aller Produktdaten und Prozesse. Der PLM-Ansatz betrifft den gesamten Produktlebenszyklus [...]. In diesem Zusammenhang ist eines der Ziele des PLM-Gedankens den Produktentstehungsprozess durch ein zentrales und konsistentes Datenmanagement zu unterstützen und die Entwicklungsproduktivität zu erhöhen."*

Damit wird mittels Prozessen der gesamte Produktionslebenszyklus von der *Planung und Entwicklung* über *Montage* und *Betrieb* bis hin in den After-Sales-Bereich mit *Service & Wartung* eines Produktes abgedeckt. Auch die Prozesse der *Demontage* - dem Ende des Lebenszyklus -gehören zu PLM.

2.3.1 Siemens Teamcenter

Diese Sektion beschäftigt sich mit *Siemens Teamcenter (TC)*, für dessen verteilte Dokumentationen im weiteren Verlauf der Arbeit ein IR System entwickelt wird.

2.3.1.1 Was ist Teamcenter?

Nachdem Siemens im Mai 2007 *UGS PLM Solutions (UGS)* übernommen hatte, wurde *Siemens PLM Software* in den *Siemens Industry Sector* eingegliedert. Diese Division ist für eine Weiterentwicklung der PLM Software *Teamcenter* zuständig. Die ursprüngliche Idee in den 1980er war die Entwicklung eines reinen *Computer Aided Design* (CAD) Systems, das sich jedoch schnell in die Richtung einer ganzheitlichen PLM Suite entwickelt hat (vgl. [CIM10, S. 2]). Der Softwarekern wurde in den weiteren Jahren kontinuierlich um weitere Komponenten erweitert, damit ein vollständiges PLM Umfeld abgedeckt werden kann.

Um *Teamcenter* weiter auf dem aktuellen technischen Stand zu halten, wurde in den 1990er Jahren die Software in mehreren Versionen von Grund auf überarbeitet und modernisiert. Das aktuelle Teamcenter 12 besteht aus einer Vielzahl von Technologien und Programmiersprachen. Das war die Ursache für das im folgenden Kapitel beschriebene Problem, denn die Softwaredokumentationen zu diesen Komponenten sind auf viele verschiedene Anlaufstellen verteilt.

2.3.1.2 Dokumentationsquellen für Administratoren

Administratoren müssen bei der Suche nach Informationen verschiedenste dezentrale Quellen durchsuchen. Sie haben oftmals die folgenden Quellen gleichzeitig offen, um das benötigte Ergebnis zu finden. Dies erweist sich häufig als sehr zeitintensiv.

PLM Documentation Server (PLM-DS) Der *PLM Documentation Server* ist eine eigenständige Software, die die Siemens PLM Produkte dokumentiert. Dieser besteht aus einer Webserverkomponente und einem *Solr* Index Server. Damit können die Dokumentationen über den Browser aufgerufen werden. Wenn für ein bestimmtes Produkt eine Dokumentation benötigt wird, kann diese in diesen Softwarestack installiert werden. Dort

finden sich genaue Anleitungen für Administratoren, die den Betrieb der Software unterstützen.

Siemens Global Technical Access Center (GTAC): Solution Center Das *Global Technical Access Center* liefert Unterstützung und Support für *Siemens PLM Produkte*. Das *Solution Center* ist die Anlaufstelle für Probleme mit Teamcenter aller Art. Hier finden sich neben *FAQ*s oder *Updatenotes* auch *Problem Reports*. Probleme anderer Administratoren werden so mit möglichen Lösungsschritten dokumentiert und veröffentlicht.

Trotz der umfangreichen Datenbasis des Portals, ist die Nutzung aufgrund einer langsamen Suche zeitintensiv. Häufig dauert es eigenen Erfahrungen nach mehrere Sekunden (Stand Juli 2018), bis eine Suche komplett abgearbeitet wurde.

Online Webseiten Da für die Administration auch z.B. spezielle Linux Befehle notwendig sind, werden diese Informationen im Internet gesucht.

Unstrukturierte Dokumente aus Projekten Dokumentationen zu Lösungen, die speziell für Kundenprojekte entwickelt wurden, werden in eigenen Projektbereichen verwaltet. Oftmals entstehen in Projekten Lösungen die auch an anderen Stellen verwendet werden könnten. In diesem Zuge entstehen Projektdokumentationen, wie zum Beispiel Designdokumente.

2.4 Evaluierung eines IR Systems

Bevor ein IR System ausgerollt wird, muss untersucht werden, wie gut dieses seinen Zweck erfüllt. Hierfür existieren spezielle Metriken, die vor allem dazu benutzt werden, die Qualität der zurückgelieferten Ergebnisse mit anderen Systemen zu vergleichen.

2.4.1 Grundlagen

Im Folgnden werden zwei Metriken vorstellt, die das Untersuchen der Ergebnisqualität eines IR Systems ermöglichen: **Precision** und **Recall**.

Im Folgenden sei G die Grundgesamtheit an Dokumenten, auf der ein IR System operiert. Sei Res eine Menge von Ergebnissen, die vom IR System auf eine Query zurückgeliefert wird und Rel die Menge aller relevanter Dokumente $Rel, Res \subseteq G$. Die folgenden Definitionen sind im Werk von Manning et al. [MRS08, S. 155] zu finden.

Definition 2.4.1 (Precision)**.** Durch die *Precision* wird das Verhältnis von relevanten zu irrelevanten Dokumenten in der Antwort bestimmt. Diese ist definiert durch

$$\text{Precision} = \frac{|\text{Res} \cap \text{Rel}|}{|\text{Res}|}$$

Definition 2.4.2 (Recall)**.** Der Recall drückt aus, welcher Anteil von allen relevanten Dokumenten geliefert wurde. Er ist definiert durch

$$\text{Recall} = \frac{|\text{Res} \cap \text{Rel}|}{|\text{Rel}|}$$

Eine Suchmaschine könnte einfach alle Dokumente zurückliefern, um einen maximalen Recall von 1.0 zu erhalten. Dabei minimiert sich jedoch der Precision-Wert, weil sich der Anteil an relevanten Dokumenten reduziert. Somit beeinflussen sich diese beiden Maße gegenseitig.

Beispiel Ein IR System besteht aus den zehn Dokumenten $G = \{d_0, d_1, ..., d_9\}$. Auf eine Abfrage hin sind vier relevant: $\text{Rel} = \{d_0, d_4, d_5, d_9\}$. Das System liefert nun fünf Ergebnisse: $\text{Res} = \{d_1, d_2, d_4, d_5, d_8\}$.

Somit ergibt sich eine Precision $P = \frac{|\text{Res} \cap \text{Rel}|}{|\text{Res}|} = \frac{2}{5} = 40\%$ und ein Recall von

$$R = \frac{|\text{Res} \cap \text{Rel}|}{|\text{Rel}|} = \frac{2}{4} = 50\%$$

2.4.2 Evaluierung der gewichteten Ergebnisse

Die Reihenfolge der Ergebnisse ist bei Suchmaschinen essentiell, da der Nutzer die Ergebnisliste von oben nach unten auf brauchbare Dokumente hin absucht. Somit erfahren höher eingeordnete Ergebnisse eine größere Aufmerksamkeit, was in den Metriken, die die Qualität der Antworten bestimmen, berücksichtigt werden muss.

Suchmaschinen generieren aus verschiedensten Parametern einen Score, der die Wichtigkeit eines zurückgegebenen Ergebnisses einer reellen Zahl zuweist. Somit können alle von der Suchmaschine gefundenen Einträge sortiert werden. Im Folgenden werden die vom IR System gelieferten Ergebnisse als Liste $\text{Res} := [a_0, a_1, ..., a_n]$ mit $a_n \in G$. $a_0 > a_1 > ... > a_n$ betrachtet, die nach der vom System berechneten Relevanz geordnet sind.[13]

2.4.2.1 Precision@k

Die erste Möglichkeit ist, nur die Präzision der Top k Ergebnisse zu betrachten. Dieser Ansatz wird als *Precision at k (P@k)* bezeichnet (vgl. [MRS08, S. 161]):

$$P@k = \frac{|\text{Res}[1..k]| \cap \text{Rel}}{|k|}$$

Beispiel Definiert seien zwei Abfragen q_1 und q_2, welche die folgenden Ergebnisse zurückliefern. Diese sind in den Listen so kodiert, dass eine '1' ein *relevantes* und eine '0' ein *nicht relevantes* Dokument beschreibt: $q_1 = [1,0,0,1,1]$ und $q_2 = [1,1,0,1,1]$. Damit ergeben sich die in Tabelle 2.2 abgebildeten P@K Werte.

[13] Hierbei bezeichnet '>' einen binären Operator als 'ist wichtiger als'

2.4.2.2 Average Precision (AP)

Damit das k nicht willkürlich auf einen Wert gesetzt werden muss (z.B. $k = 10$), bildet man meist das arithmetische Mittel über die einzelnen P@k Werte, was als *Average Precision* bezeichnet wird (vgl. [BCC16, S. 408]):

$$\mathrm{AP}_n = \frac{1}{|\mathrm{Rel}|} \sum_{i=1}^{n} P@k(i) \cdot \mathrm{relevance}(i)$$

mit der Anzahl zurückgelieferter Ergebnisse n auf eine Query q und der Anzahl der relevanten Dokumente $|\mathrm{Rel}|$. $\mathrm{relevance}(i)$ beurteilt die binäre Relevanz eines Dokumentes, womit ist die Wertemenge der Funktion $W = \{0,1\}$ beträgt.
Im Beispiel 2.2 werden die P@k Werte gemittelt, um den AP zu bestimmen. Es wird davon ausgegangen, dass vier relevante Dokumente existieren und damit die Abfrage q_1 ein relevantes Dokument nicht findet. Dafür wird $P@k = 0$ gesetzt und es ergibt sich für q_1:

$$\mathrm{AP}_{q_1} = \frac{1}{5} * (\frac{1}{1} + 0 + 0 + \frac{2}{4} + \frac{3}{5}) = 0.42$$

2.4.2.3 Mean Average Precision (MAP)

Um einen statistisch aussagekräftigen Durchschnittswert des APs für ein System zu erhalten, werden mehrere Abfragen gestellt und die einzelnen APs gemittelt. Damit eine Dopplung des Begriffs *Average* vermieden wird, wird dies als *Mean Average Precision* bezeichnet. Diese Metrik ist in den TREC (Text REtrieval Conferences) Konferenzen zu einem Standard geworden (vgl. [MRS08, S. 160]). Es ergibt sich somit:

$$\mathrm{MAP}(Q) := \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{i=1}^{m_j} P@k(R_{jk}) = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \mathrm{AP}_{\mathrm{q_i}}$$

mit der Menge aller Abfragen Q mit den *Information Needs* $q_i \in Q \quad \{d_1,..,d_{m_j}\}$. R_{jk} ist die Menge der rangierten zurückgelieferten Ergebnisse von dem Top-Result bis zum Dokument d_k. In unserem Beispiel 2.2 ergibt sich damit folgender Wert:

$$\text{MAP}(Q) := \frac{1}{|Q|}\sum_{i=1}^{|Q|}\text{AP}_{\text{qi}} = (AP_{q_1} + AP_{q_2})/2 = (0.42 + 0.71)/2 = \mathbf{0.565}$$

$$\text{mit } \text{AP}_{q_2} = \left(1 + 1 + \frac{3}{4} + \frac{4}{5}\right) \times \frac{1}{5} = 0.71$$

Position k	q_1	q_2	$q_1 - P@k$	$q_2 - P@k$
1	1	1	1/1	1/1
2	0	1	1/2	2/2
3	0	0	1/3	2/3
4	1	1	2/4	3/4
5	1	1	3/5	4/5

Tabelle 2.2: Berechnung von P@k Werten zu zwei Queries

2.4.3 Bestimmung des Recalls

Oftmals ist es unmöglich, die genaue Anzahl der relevanten Dokumente aus der Grundgesamtheit zu einer Anfrage zu bestimmen. Heutige Suchmaschinen greifen auf Millionen von Dokumenten zurück, bei denen schwer zu sagen ist, wie viele hiervon wichtig sind. Damit wird die Bestimmung des Recalls sehr schwierig.

Es gibt aber einige Möglichkeiten, diesen hochzurechnen und Systeme vergleichbar zu machen. Diese werden von *Henrich* (vgl. [Hen08, S.70]) beschrieben. Im Folgenden wird die Technik des *Poolings* vorgestellt.

Pooling Durch *Pooling* können verschiedene IR Systeme miteinander verglichen werden, wenn sie auf die gleiche Datenbasis zugreifen. Diese können beispielsweise unterschiedliche Scoringfunktionen oder Parameter sein. Dabei wird die Ergebnismenge aller Systeme auf eine bestimmte Abfrage q vereinigt. Es wird die Annahme getroffen, dass jedes wichtige Dokument von mindestens einem System gefunden wird. Diese Vereinigung stellt die Menge aller gesuchten Dokumente dar, aufgrund derer der Recall berechnet wird.

KAPITEL 3

ANFORDERUNGSANALYSE

Zunächst wird definiert, welche Anforderungen sich aus der Problemstellung (siehe 1.1) ergeben.

- *Funktionale Anforderungen*: Bestimmte Funktionalitäten, die die Anwendung erfüllen muss.
- *Nichtfunktionale Anforderungen*: Beschreiben, nach welchen Qualitätsmaßstäben funktionale Anforderungen erfüllt sein müssen.
- *Technische Anforderungen*: Legt Techniken für Realisierung fest.

3.1 Funktionale Anforderungen (AF)

1. Datenhaltung
    a) Die Quelldaten aus folgenden Quellen müssen für unsere Applikation extrahiert und indexiert werden:

i. **AF-1-1-1** Siemens PLM Documentation Server

ii. **AF-1-1-2** GTAC Solution Center

iii. **AF-1-1-3** Spezielle Webseiten im PLM Kontext

iv. **AF-1-1-4** Weitere Dokumenttypen

b) **AF-1-2-1** Die Daten sollen sich in fest definierbaren Intervallen automatisch aktualisieren.

2. Die Oberfläche der Anwendung muss über folgende Funktionen verfügen:

a) Die Suchergebnisse müssen in folgender Art dargestellt werden:

i. **AF-2-1-1** Alle Ergebnisse müssen übersichtlich dargestellt werden.

ii. **AF-2-1-2** Bei einem Mausklick auf ein Ergebnis muss der Nutzer auf die entsprechende Quelle weitergeleitet werden.

iii. **AF-2-1-3** Die Ergebnisse müssen aufbereitet in einer abgekürzten Form dargestellt werden.

iv. **AF-2-1-4** Durch *Pagination* werden dem Nutzer immer nur ein paar Ergebnisse pro Seite angezeigt, wie das von gängigen Suchmaschinen bekannt ist.

b) Die Suche selbst muss folgende Anforderungen erfüllen:

i. **AF-2-2-1** Darstellung einer *Suchzeile*, in die der Nutzer seine Abfragen eingeben kann. Diese muss zu jedem Zeitpunkt auf der Seite erreichbar sein, um eine schnelle Bedienung zu gewährleisten.

ii. **AF-2-2-2** Die Einträge müssen während der Eingabe aktualisiert werden.

iii. **AF-2-2-3** Die Suche muss über eine erweiterte Suchfunktion verfügen, die diese beispielsweise um logische Operatoren erweitert.

iv. **AF-2-2-4** Ergebnisse müssen nach Sprache, Dateiformat und Quelle gefiltert werden können.

v. **AF-2-2-5** Es muss ein Algorithmus ausgewählt werden, der die Reihenfolge der Suchergebnisse optimiert.

vi. **AF-2-2-6** Die Suche muss über alle in **AF-1-1** definierten Quellen erfolgen.

3.2 Nichtfunktionale Anforderungen (AN)

AN-1 Die Suche muss schnell sein. Hierbei muss bei einem Datensatz von 30.000 Einträgen die Abfragezeit und Antwortzeit unter 200ms gehalten werden.

AN-2 Die Suchergebnisse sollen eine angemessene Qualität erreichen.

3.3 Technische Anforderungen (AT)

AT-1 Da die meisten Arbeitssysteme der Anwendergruppe unter *Windows 7* laufen, muss eine Unterstützung dieses Betriebssystems gewährleistet sein.

KAPITEL 4

METHODE

Zunächst wir beschrieben, wie die Problemstellung aus Punkt 1.1 gelöst wurde. Hierfür wird ein IR System entwickelt, das in Echtzeit durchsucht werden kann.

Bei der Entwicklung des Systems wurde schrittweise vorgegangen: Zuerst wurde ein ES Indexserver aufgesetzt und die Daten aus den verschiedenen Quellen von Teamcenter Dokumentationen gesammelt. Daraufhin wurden diese indexiert und in einer Datenbank gespeichert. Ein webbasiertes Front-End dient als Interaktionsschnittstelle mit dem Nutzer, worüber die Daten durchsucht werden können.

Das Ranking der Ergebnisse wird mit *Scoringfunktionen* kalkuliert. Um hierfür diejenige zu finden, welche die Besten liefert, wird eine Evaluation durchgeführt, die die Qualität der verschiedenen Funktionen in unserem System miteinander vergleicht.

4.1 Verwendete Werkzeuge

Die Wahl der Werkzeuge der verschiedenen Komponenten ist auf folgende Technologien und Programmiersprachen gefallen:

1. Web Frontend: **React.js** mit **searchkit**
2. Index Server: **Elasticsearch** & **Kibana**
3. Datenmigration aus TC Dokumentationen (siehe 2.3.1.2)
    a) *PLM Documentation Server*: **Python Script**
    b) *GTAC Solution Center*
        i. Crawling: **Python Script**
        ii. Importieren in Elasticsearch: **JavaScript**
    c) Webcrawling: *JavaScript* **Crawler + Importer**

4.2 Evaluierung der Scoring Funktionen

Um geeignete Scoringfunktionen und Parameter in ES zu finden, wurde eine Evaluation durchgeführt.

Hierbei wurden 50 Fragen definiert, die den täglichen Informationsbedarf eines PLM Administrators abbilden und auf drei verschiedene Scoringmodelle nach AP und MAP Werten evaluiert: *Okapi BM25*, *Divergence from Randomness* und *Divergence from Independence* (siehe 2.1.4). Da die Menge der relevanten Dokumente je Abfrage unbekannt ist, wird auf *Pooling* zurückgegriffen.

KAPITEL 5

DURCHFÜHRUNG

Die folgende Durchführung erläutert, wie bei der Entwicklung der Lösung vorgegangen wurde.

5.1 Architektur der Lösung

Die Lösung besteht aus vier Kernkomponenten: Der Datenbasis, dem *Web-*, *Index-, Data-* und *Client Tier*, siehe Abbildung 5.1. Eine 4-Tier Architektur eignet sich, weil dadurch die Entwicklung der einzelnen Komponenten unabhängig voneinander erfolgen kann. Zudem erfolgt eine klare Abgrenzung der einzelnen Funktionen innerhalb des Systems, welches dadurch wart- und erweiterbarer wird. Ein zentraler Indexserver hat zudem den Vorteil, dass neue Datensätze sofort für alle Clients zur Verfügung stehen. Dieser Sachverhalt wird nachfolgend diskutiert:

5.1.1 Data Tier

Im Data Tier liegen die Quelldaten vor, die für eine schnelle Suche indexiert werden. Diese Quellen wurden in 2.3.1.2 beschrieben: Der *PLM Documentation Server*, das *GTAC Solution Center*, Webseiten und unstrukturierte Dokumente, wie PDFs oder Excel Tabellen.

5.1.2 Web Tier

Für den Webtier wurden einige Programme entwickelt, die die Quelldaten aus dem Data Tier extrahieren und in einen ES Server importieren können.

SOLR-TO-ES ist ein Pythonprogramm, das einen SOLR Index nach Elasticsearch migriert. Damit werden die Daten des *PLM Documentation Servers* in unsere ES Umgebung importiert.

GTAC-Crawler ist ein Python-Crawler, der die Einträge des GTAC Solution Centers im xml Format exportiert. Ein JavaScript Programm liest diese aus und kopiert sie in den ES Server.

Webcrawler ist ein auf JavaScript basierender Crawler, der Webseiten bis zu einer bestimmten Tiefe durchsucht, indem er Querverweise (Hyperlinks) verfolgt. Dadurch ist es möglich, Webdokumentationen oder Webseiten schnell zu extrahieren und indexieren. Eine zweite Komponente, die auch vom **GTAC-Crawler** verwendet wird, lädt diese Daten dann in den ES Server.

Weitere Dokumente[1] Text aus Binärdateien wie Bilder, Bücher, usw. können mithilfe von *Apache Tika* extrahiert und ebenfalls indexiert werden.

[1] Nur konzeptionell

5.1.3 Index Tier

Der Index Tier besteht aus einem *Elasticsearch* Indexserver, in dem die Daten indexiert werden. Hier werden Abfragen sowohl entgegengenommen, als auch verarbeitet und stellen damit den Kern der Suchmaschine dar.

5.1.4 Client Tier

Der Client Tier stellt eine Webseite - realisiert mit *react.js* - für den Nutzer zur Verfügung. Hier werden in Echtzeit Abfragen an den ES Server gestellt und die Ergebnisse aufbereitet.

Der Vorteil an einer webbasierten Lösung liegt darin, dass die Software zentral aufgesetzt werden kann und jeder internetfähige Browser im Firmennetz darauf Zugriff hat.

5.2 Installation des Index Servers

5.2.1 Elasticsearch vs. Solr

Elasticsearch hat in den letzten Jahren an Bedeutung zugenommen und laut *Google Trends* *Apache Solr* bereits 2015 an weltweitem Interesse überholt. Laut Grafik 5.2 ist das Interesse an *Elasticsearch* (Blaue Linie) die letzten Jahre kontinuierlich gestiegen, während *Solr* (Rot) auf einem gleichbleibend niedrigen Wert stagniert. Das drückt sich auch in den Zahlen auf Github aus, auf dem beide Open-Source Projekte verwaltet und veröffentlicht werden: 32.000 mal[2] wurde mit einem *Star* ein besonderes Interesse an *Elasticsearch* bekundet, wohingegen das Interesse an das längere existierende *Apache Solr* nur 1.800 *Stars*[3] hervorgebracht hat - ein Zeichen dafür, dass sich das allgemeine Interesse nach *Elasticsearch* verlagert hat.

[2] https://github.com/elastic/elasticsearch, Stand August 2018
[3] https://github.com/apache/lucene-solr, Stand August 2018

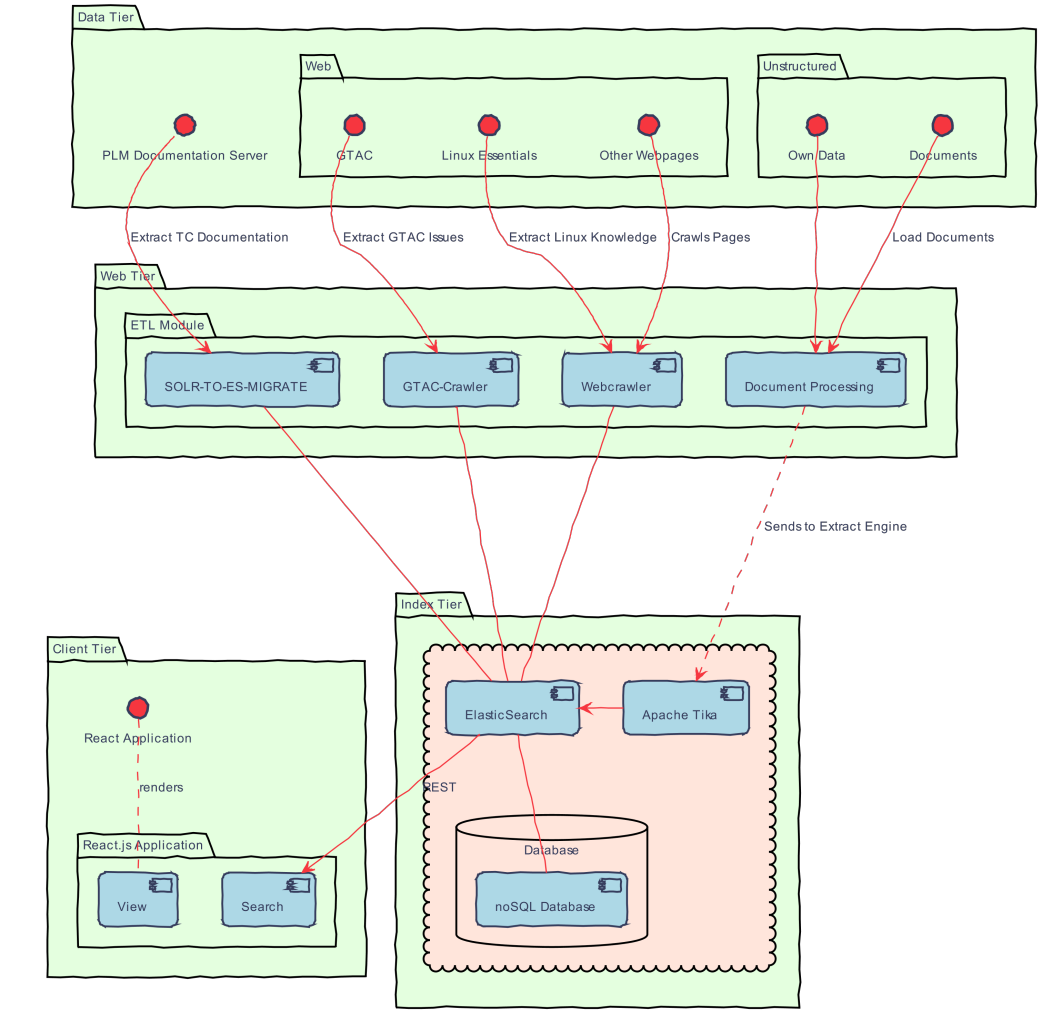

Abbildung 5.1: Grobarchitektur der gesamten Lösung

Der Funktionsumfang beider Plattformen ist ausgewogen und ähnlich. Während *Elasticsearch* durch bessere Analysemöglichkeiten und die Möglichkeit in der Cloud besser zu skalieren, besticht, trumpft *Apache Solr* mit einem auf die Jahre gewachsenen größeren Ökosystem und Dokumentationsschatz.

Der Grund, warum viele auf *Elasticsearch* setzen, ist seine Leichtgewichtigkeit. Die gezippte Installationsgröße von *Elasticsearch* beträgt nur 80MB wohingegen *Solr* mit 160MB das Doppelte benötigt. ES läuft ohne Konfiguration out-of-the-Box, wohingegen Solr zwin-

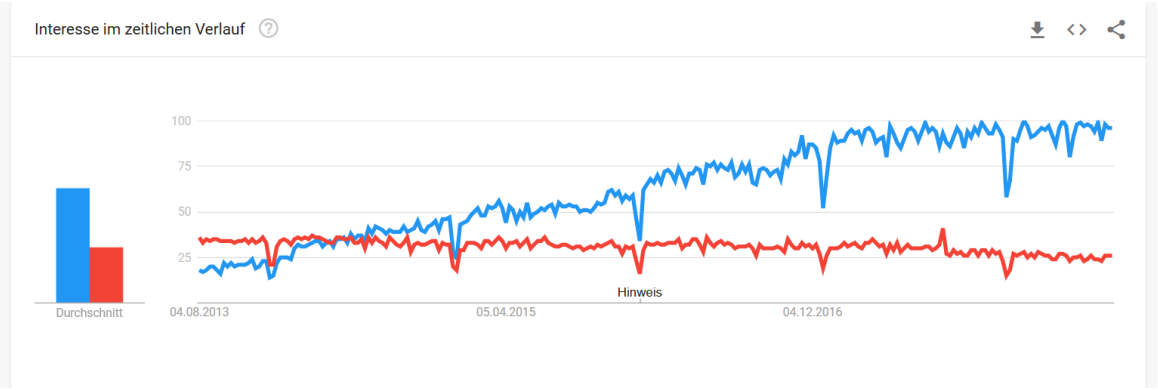

Abbildung 5.2: Google Trends: Vergleich von Solr und Elasticsearch: Solr & Elasticsearch [Goo18]

gend auf Konfiguration angewiesen ist. Daher kann *Elasticsearch* auch ohne Vorwissen schnell produktiv verwenden werden, was vor allem für jüngere Entwickler interessant ist. Jedem, der sich für die genaueren Unterschiede interessiert, sei dieser[4] Artikel empfohlen.

5.2.2 Installation des ELK Stacks

Docker[5] ist eine *Open-Source* (vgl. [Tur14, S.7]) Platform zur Isolation von Anwendungen in Containern mithilfe von Betriebssystemvirtualisierung (vgl. [Tur14, S.7]). Dadurch ergibt sich eine leichtgewichtige und schnelle Virtualisierungstechnik, die auf *Kernel*ebene agiert (vgl. [Tur14, S. 9]). Programme können fertig konfiguriert ohne viel Aufwand in ein neues System eingespielt werden. Mit *Docker Compose* kann ein ganzes Containersystem in einer Textdatei definiert werden (vgl [RS17, S.148ff]). Das bringt den Vorteil, dass ein ganzer ELK[6] Technologiestack auf einmal erstellen werden kann. Dieser ist außerdem plattformunabhängig (vgl. [RS17, S. 9]).

Ein solches Composefile wird im folgenden diskutiert:

[4] https://logz.io/blog/solr-vs-elasticsearch/ (17.08.2018)
[5] https://www.docker.com/ (20.08.2018)
[6] Elastic, Logstash, Beats und Kibana

```
1  version: '2'
2  services:
3          elasticsearch:
4                  build:
5                  - context: elasticsearch/
6                  ports:
7                  - "9200:9200"
8                  environment:
9                  - ES_JAVA_OPTS: "-Xmx256m -Xms256m"
10                 networks:
11                 - elk
12         logstash: [...]
13         kibana:
14                 depends_on:
15                 - elasticsearch
16         plmdocsrv: [...]
17 networks:
18         elk:
19                 driver: bridge
```

- In Zeile 3, 12, 13 und 16 wird jeweils ein Container für unseren ELK Stack und den *PLM Documentation Server* erstellt: *elasticsearch*, *kibana*, *logstash* und *plmddocsrv*.
- Hierfür werden die benötigten Ports freigegeben. Für elasticsearch ist das beispielsweise der Port 9200 (Z. 7).
- Alle Container werden einem virtuellen Netzwerk *elk* (Z. 19) zugewiesen, in dem sie miteinander kommunizieren können.

Mithilfe eines einzelnen *docker-compose up* Befehls wird der ganze Stack auf einmal gestartet. Die vollständige *Docker Compose* Datei befindet sich im Anhang A.1.

5.2.3 Definition des Indexaufbaus

Die Idee ist, dass für jeden Quelltyp ein eigener Index erzeugt wird: Einen für die GTAC Daten, einen für den PLM-DS usw. Dadurch bleiben die Daten der verschiedenen Quellen getrennt. Seit *Elasticsearch 6.0* ist es nicht mehr möglich, mehrere Typen auf einen Index

zu definieren (vgl. [ela18g]).

Damit alle verschiedenen Daten durchsucht und aggregiert werden können, wird ein Interface benötigt, eine Struktur, die jedes gespeicherte Dokument aufweisen muss. Zusätzlich fordert die Frontend Bibliothek searchkit[7], die im weiteren Verlauf der Arbeit verwenden wird, eine derartige Datenkonsistenz. Die Feldername wurden vom *Documentation Server* übernommen. Siemens hat diese selbst für den SOLR Server definiert. Diese lauten:

- *title_exact*: Der Titel eines Dokumentes
- *deliverableId*: Eine eindeutige ID
- *format*: Das Format, indem das Dokument vorliegt
- *lang*: Die Sprache eines Dokumentes
- *rawDataExact*: Der Inhalt eines Dokumentes

5.3 Füllen der Indexdatenbank

Nun wird die *Elasticsearch* Datenbank mit unseren Daten gefüllt und indexiert. Hierfür wird auf die einzelnen Data-Tier Komponenten eingegangen, die in 5.1 vorgestellt worden sind.

5.3.1 Migrieren des PLM Documentation Servers

Der PLM-DS ist eine Webapplikation zur Darstellung der Dokumentation von Siemens PLM Produkten. Dieser besitzt einen eigenen *Solr* Server, damit die Einträge durchsucht

[7] http://searchkit.co/ (19.09.2018)

werden können. Die Idee ist es, die Daten aus diesem *Solr Server* in einen ES Index unserer Anwendung zu migrieren.

Um die Daten später aufbereitet anzuzeigen, wird zusätzlich ein PLM-DS bereitgestellt. Der Nutzer kann über unsere Oberfläche seine Ergebnisse suchen und wird dann auf die entsprechende Seite im PLM-DS weitergeleitet. Das hat den Vorteil, dass die Oberfläche nicht neu geschrieben werden muss und der Nutzer sich am Ende in einer gewohnten Umgebung wiederfindet.

5.3.1.1 Erstellen eines Docker Containers für PLM-DS

Damit der *PLM Documentation Server* nicht für jedes Deployment manuell installiert werden muss, wird ein *Docker Container* erzeugt. Dieser hat den Vorteil, dass ein eigenständiger PLM-DS derart in unseren ELK Stack wird, dass dieser automatisch mit dem Indexserver gemeinsam gestartet wird.

5.3.1.2 Migrieren der Daten nach ES

Es existiert ein Python Tool, das eine Migration der Daten von *Solr* nach ES unterstützt: **SOLR-TO-ES**[8]. Im Grunde macht das Programm nichts anderes, als einmal über alle Einträge des *Solr* zu iterieren und diese dann über die HTTP Anfragen an den Elasticsearch zu senden und indexieren zu lassen.

Nach der Installation können die Daten mit folgendem Befehl migriert werden: Die

```
1 $ solr-to-es SOLRHOST:8283/solr/prod ESHOST:9200 solrprod xml
```

Abbildung 5.3 zeigt, dass alle Daten übernommen wurden.

[8] https://github.com/o19s/solr-to-es, Stand Juli 2018

Abbildung 5.3: Migrieren der Daten von SOLR to ES

5.3.2 Indexierung von GTAC Solution Center Einträgen

Das GTAC Solution Center ist eine weitere Wissensquelle für Administratoren. Hier finden sie vor allem *Problem Reports (PR)*, die mit einer Lösung veröffentlicht werden. Abbildung 5.4 zeigt einen Screenshot dieses Webportals.

5.3.2.1 Extraction der Problem Reports (PR)

Da Siemens keine Schnittstelle für diese Daten zur Verfügung stellt, müssen sie manuell einbezogen werden. Hierfür wird eine Suche an das System gestellt, die alle Einträge zurückliefert (Wildcard '*'), iteriert über diese in HTML vorliegenden Einträge und schreibt sie in eine xml Datei. Das wird von einem *Pythoncrawler* erledigt. Damit die Datenmenge nicht zu groß wird, wird sie in 470 Dateien mit jeweils 1000 Einträgen aufgesplittet, womit sich insgesamt 47.000 PRs (im August 2018) ergeben. Damit auch neuste Einträge geladen werden können, ist der Crawler so parametrisiert, dass er alle Einträge in einer gewissen Zeitspanne laden kann. Dadurch wird es möglich immer nur die Deltas seit dem letzten Update zu laden. Codebeispiel 5.1 zeigt, wie eine solche Datei aufgebaut ist. Ein PR besitzt somit neben der eigentlichen Problemstellung und dem Titel auch Metdaten wie das Erstellungsdatum, den Dokumenttypus und eine eindeutige ID.

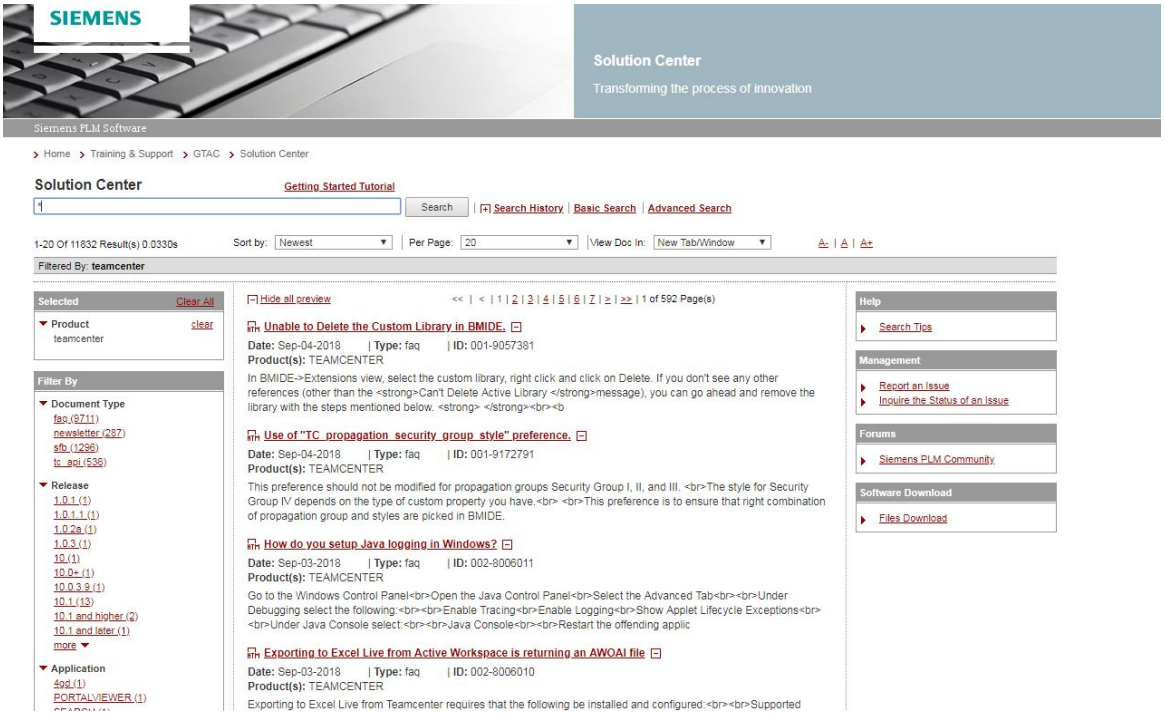

Abbildung 5.4: GTAC Solution Center (Screenshot)

5.3.2.2 Erstellung des Index und Import der Daten (Importer)

Nun müssen die Extrakte in den ES Server importiert werden. Hierfür wurde ein JavaScript (JS) Programm geschrieben, das die Daten einliest, parst und in den Index Server schreibt. Mithilfe von JS können XML Daten ausgelesen und verarbeitet werden. Hierfür wird *Elasticsearch.js* benutzt, die offizielle JS API für Elasticsearch. Codebeispiel 5.2 zeigt einen kleinen Ausschnitt, der im Folgenden analysiert wird.

1. In Zeile eins bis fünf wird das ES Modul geladen und ein Objekt mit einer Verbindung zum Index Server instantiiert.

2. In Zeile sieben wird eine Funktion *client.indices.create* aufgerufen, die einen neuen Index erstellt. Hierfür wird der Index innerhalb eines JSON Objektes definiert. Beispielsweise definiert Zeile 13 das Feld *title_exact* als *analysierten String*, um es für

Code 5.1: Beispiel für Extraktion aus Solution Center

```
1  <topic href="view.php?q=*&amp;dt=&amp;rows=100&amp;sort=desc&amp;␣
       ↪ ds=&amp;de=&amp;pd=teamcenter&amp;ds=01-01-2010&amp;␣
       ↪ p=15&amp;file_type=text&amp;i=pr-8368529&amp;k=4&amp;o=1400">
2          <title>TCRS-*-approve-first-step cause TCServer crash</title>
3          <date>2018-03-28</date>
4          <type>problem report</type>
5          <ident>8368529</ident>
6          <problem>
7          Product(s): TEAMCENTER Software Versions/Configuration:
8          ================================
9          TC: 10.1.7, 11.4
10         Description of Problem:
11         ======================
12         Using the handlers TCRS-trigger-approve-first-step and
               ↪ TCRS-auto-approve-first-step TC server crash
13         [...]
14         </problem>
15 </topic>
```

eine Volltextsuche bereit zu machen.

3. Fehler werden in einem eigenen Callback in Zeile 22 abgefangen und nach oben gereicht.

Jetzt können unsere Daten in den ES Server geschrieben werden. Dafür gibt es zwei Möglichkeiten: Die Datensätze einzeln zum Server zu schicken oder gebündelt als sogenannte *bulk*s. Das Problem an dem ersten Ansatz ist, dass zum einen unnötig viel Datenverkehr entsteht, weil für jedes Problem ein eigener *HTTP Request* erstellt wird, zum anderen der Server schlicht überlastet ist, denn das Indexieren der Daten benötigt Zeit. Dieser hat eine begrenzte Queue, die bestimmt, wie viele Requests zwischengespeichert werden. Sollten nun mehr Anfragen kommen, als der ES bewältigen kann, wird eine Fehlermeldung zurückgegeben und der Request abgelehnt.

Eleganter ist die Möglichkeit, *bulk*s zu nutzen. Dabei werden Hunderte zu indexierende Einträge in einzelnen Anfragen mitgeschickt, was deutlich performanter ist. Das Code-

Code 5.2: Erstellung des Indexes

```
1  var ElasticSearch = require('elasticsearch');
2  var client = new ElasticSearch.Client({
3          host: localhost,
4          apiVersion: 1.6
5  })
6  function createIndex(name, callback) {
7  client.indices.create({
8          index: name,
9                  body: {
10                         "mappings": {
11                                 "aType": {
12                                         "properties": {
13                                                 "title_exact": {
14                                                         "type": "string",
15                                                         "index": "analyzed"
16                                                 }, // title
17                 [...] // Declaration of other fields
18                                                 }
19                                 }
20                         }
21                 }
22         }, function (err, resp, respcode) {
23                 callback(err, resp, respcode);
24         })
25 }
```

beispiel 5.3 zeigt, wie die geladenen Daten an den ES Server geschickt werden. Hierfür wird die Funktion *bulk* verwendet. Sollten Fehler auftreten, wird der Vorgang bis zu einer gewissen Anzahl Neuversuche rekursiv wiederholt (Z. 8).

5.3.3 Webcrawler

Um beliebige Webseiten zu indexieren wurde ein Crawler geschrieben, der Webseiten systematisch absuchen kann. Hierfür folgt er den Querverweisen in einem HTML Dokument bis zu einer bestimmten Suchtiefe. Auf diese Weise können automatisch ganze Dokumentationen auf einmal geladen werden.

Code 5.3: Versenden der Bulks

```
1  // body: variable with GTAC data
2  function uploadBulk(body, numRetries) {
3          if (numRetries > 0) {
4                  client.bulk({
5                          "body": body
6                  }, function (err, resp) {
7                          if (resp.errors) { // If an Error occured, we will resend the bulk
8                                  uploadBulk(body, numRetries--);
9                          }
10                 });
11         }
12 }
```

Für die Implementation wird das npm Modul simplecrawler[9] verwendet. Im folgenden Codeauszug 5.4 wird ein neues Crawler-Objekt erzeugt und einige Parameter gesetzt, wie beispielsweise die Suchtiefe oder den Zeitabstand, in dem neue Seiten geladen werden.

Code 5.4: Setzen von Crawler Parametern

```
1  var crawler = new crawler("http://beyondplm.com/");
2  crawler.interval = 250;
3  crawler.maxConcurrency = 3;
4  crawler.maxDepth = 4;
5  crawler.scanSubdomains = true;
6  crawler.respectRobotsTxt = true;
```

Über Events kann der Crawler nun verwaltet werden. Beispielsweise wird an das *fetch-complete* Event eine Callbackfunktion (Codeauszug 5.5) übergeben, die nach dem Laden der Seite aufgerufen wird.

Code 5.5: Callbackfunktionen für Events

```
1  crawler.on('fetchcomplete', (queueItem, responseBody, responseObject) => {
2          // Do something with the data
3  });
```

Der Upload der Daten selbst erfolgt mit dem in 5.3.2.2 vorgestellten Importer.

[9] https://www.npmjs.com/package/simplecrawler

5.3.4 Verarbeitung von Binärdokumenten

In diesem Abschnitt wird ein Konzept, wie weitere Dokumenttypen durchsucht werden können, vorgestellt. Leider konnte dieses nicht mehr implementiert werden.

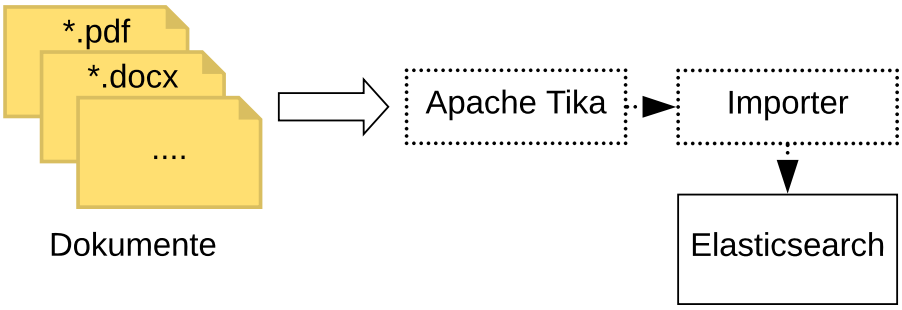

Abbildung 5.5: Indexierung von Binärdokumenten

Ein möglicher Entwurf ist in Abbildung 5.5 abgebildet. Die Idee ist es, zunächst mögliche Texte aus den Dokumenten zu extrahieren und dann zu indexieren. Hierfür bietet sich das Werkzeug Apache Tika[10] an, das aus einer Vielzahl von verschiedenen Dateiformaten Texte und Metadaten auslesen kann. Ein Importer nimmt die Daten und transferiert sie dann in einen ES Index.

5.4 Entwicklung des Frontends

Nun wird beschrieben, wie ein Front-End (siehe Abbildung 5.6) für *Elasticsearch* entwickelt wurde. React.js[11] ist eine Open-Source JavaScript Bibliothek von *Facebook* um User Interfaces zu erstellen. (vgl. [Fac18]) Der Grund für diese Wahl ist das im Folgenden verwendete *Searchkit*[12], das auf *react.js* basiert.

[10] https://tika.apache.org/
[11] https://reactjs.org/ (28.09.2018)
[12] http://searchkit.co/ (28.09.2018)

Searchkit stellt eine Menge von Komponenten zur Verfügung, mit denen ein ES Server durchsucht werden kann: Neben einer Suchleiste können auch Filterelemente benutzt und die Darstellung der Ergebnisse (Hits) modifiziert werden. Neben der reinen grafischen Oberfläche kümmert sich *Searchkit* zusätzlich um die Verbindung zu dem Indexserver, das Formulieren von Anfragen oder das Verarbeiten der Antworten.

Searchkit kann mit dem node.js[13] Paketinstaller *npm* über die Kommandozeile installiert werden.

```
1 $ npm install searchkit --save
```

5.4.1 Initialisieren der Verbindung zwischen ES und Front-End

Nachdem jetzt die notwendigen Vorbereitungen erledigt wurden, kann eine Verbindung zu einem ES initialisiert werden. Hierfür wird ein sogenanntes *SearchkitManager* Objekt, das fortan als zentrale Schnittstelle zum Server fungiert. Zunächst muss jedoch die *searchkit* Bibliothek eingebunden werden. Codebeispiel 5.6 zeigt dies.

Code 5.6: Konfigurieren des Searchkit Managers

```
1 import {SearchkitManager} from searchkit; // Einbinden der Bibliothek
2 var searchkit = new SearchkitManager("localhost:9200/");
```

5.4.2 Darstellung der Komponenten

Die einzelnen UI Komponenten können nun verwendet werden, um ein ansprechendes User Interface zu gestalten. Abbildung 5.6 zeigt die finale Webapplikation in einem Webbrowser. Die im folgenden erläuterten Elemente sind in diesem Bildschirmfoto farbig hervorgehoben.

[13] https://nodejs.org/en/ (19.09.2018)

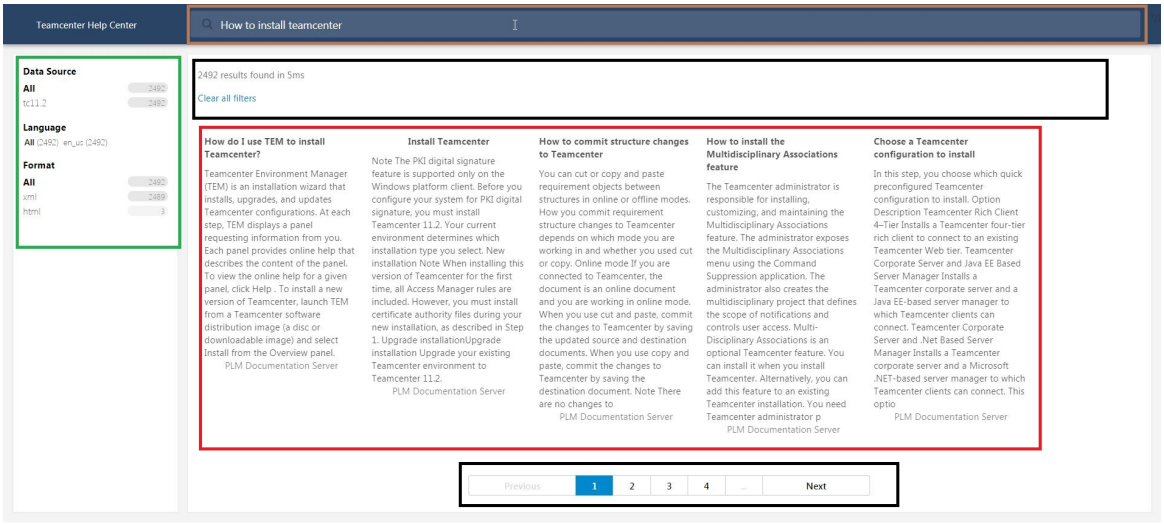

Abbildung 5.6: Die Searchkit-UI

5.4.2.1 Suchleiste

Zunächst wird eine Suchleiste, wie man es von Google und Co. kennt, am oberen Bildschirmrand implementiert. Hierfür wird eine *Searchbox* Komponente benutzt und die Felder, die in eine Suche mit einbezogen werden sollen, definiert. Es wird der *Standard Analyzer* verwendet, der keine Stopwörter filtert.

Code 5.7: Such Komponente

```
1 <div className="sk-layout">
2         <div className="sk-layout__top-bar sk-top-bar">
3                 <div className="sk-top-bar__content">
4                         <div className="my-logo">Teamcenter Help Center</div>
5                                 <SearchBox searchOnChange={true} queryFields={{"title_exact",
                                      ↪ "lang", "format", "rawDataExact"}}
                                      ↪ queryOptions={["analyzer":"standard"]}>
6                                  </Searchbox>
7                         </div>
8                         [...]
```

1. In Zeilen eins bis drei werden zunächst einige *Searchkit* Komponenten eingebunden, die das Layout der Seite definieren.

2. In Zeile 5 wird die Suchleiste eingebunden. Hierfür werden in den HTML Klassen die Felder definiert, die im Index durchsucht werden sollen.

5.4.2.2 Filtern von Einträgen

In 5.2.3 wurde eine Schnittstelle definiert, die jeder Index implementieren muss. Der Grund hierfür war, dass nun über alle Indizes aggregieren werden soll. Dies ist aber nur möglich, wenn die Felder, über die aggregiert wird einheitlich sind.

Im Speziellen werden die Felder *format*, *deliverableId* und *lang* benutzt, damit der Nutzer seine Ergebnisse vertikal durchsuchen kann. "'*Eine Vertikale Suche bezeichnet [eine] Suchanfrage, die sich nur auf ein bestimmtes Themengebiet oder ein spezielles Segment einer gesamten Suche bezieht.*" [Ryt18] Searchkit stellt mit seinen *Facets* Komponenten zur Verfügung, die eine weitere Einschränkung der Suchergebnisse ermöglichen.

Codebeispiel 5.8 zeigt, wie die *MenuFilter* Filterlemente implementiert werden. Dazu muss der Feldname, wie er im ES Index benannt wurde, in dem *field* Attribut angegeben werden:

Code 5.8: Filter Komponente

```
1 <div className="sk-layout__filters">
2         <MenuFilter field="deliverableId" title="Data Source" id="source"
              ↪ listComponent={ItemHistogramList} bucketsTransform={this.bucketTransform}>
              ↪ </MenuFilter>
3         <MenuFilter field="lang" title="Language" id="lang"
              ↪ listComponent={TagCloud}></MenuFilter>
4         <MenuFilter field="format" title="Format" id="format"
              ↪ listComponent={ItemHistogramList}> </MenuFilter>
5 </div>
```

1. Zunächst wird in Zeile eins ein Abschnitt mit Filtern eingeleitet.

2. Die folgenden Zeilen erzeugen die Filter Komponente. In den Attributen werden dann die entsprechenden Parameter gesetzt, die definieren, *welche* Felder durchsucht und

wie diese angezeigt werden. In Zeile drei wird zum Beispiel ein Filter erzeugt, der nach der Sprache des Dokumentes aggregiert.

5.4.2.3 Anzeigen der Ergebnisse

Eine *Hits* Komponente erlaubt, die Ergebnisse in einer Liste darzustellen, wie es in 5.6 in rot hervorgehoben ist. Codebeispiel 5.9 zeigt, wie ein Abschnitt festgelegt wird, in dem die Suchergebnisse (Hits) gerendert werden.

Code 5.9: Ergebnisliste

```
1 <Hits hitsPerPage={this.state.hitsPerPage}
2         highlightFields={["title_exact", "text_exact", "rawDataExact"]}
3         mod="sk-hits-grid" itemComponent={HitItem}>
4 </Hits>
```

Die Darstellung der einzelnen Items lässt sich verändern, indem eine *HitItem* Komponente definiert wird. An dieses wird jeweils ein *props* Objekt gegeben, das den Inhalt des Suchergebnisses beinhaltet.

Dem Attribut *itemComponent* wird in Zeile drei eine React-Komponente übergeben, die für jeden *Hit* gerendert wird. Diese wird in Codebeispiel 5.10 gezeigt. Hierbei wird jedes Item mit einer *onClick* Funktion belegt, die aufgerufen wird, wenn das Element von dem Nutzer angeklickt wird. In diesem Fall wird die Funktion *openNewTab* aufgerufen, die dann auf die hinterlegte *URL* zu jedem Item weiterleitet.

Code 5.10: Definition der einzelnen Hit Items

```
1 const HitItem = (props) => (
2 [...]
3 className={[props.bemBlocks.item().mix(props.bemBlocks.container("item")), "item"].join(" ")}
      ↪ onClick={() => this.openNewTab(props.result)}>
4 [...]
```

Code 5.11: Weiterleitung zu den Ergebnissen

```
1 openNewTab = (e) => {
2         [...]} // Do some parsing
3                 window.open("" + address, "_blank"); // Opens up a new tab
4         }
```

5.4.3 Weiterleitung auf entsprechende Quellen

Es wurden zwei Möglichkeiten untersucht, das ganze Ergebnis im Browser anzuzeigen.

Die erste Möglichkeit liegt darin, den Inhalt der externen Quelle direkt auf der Suchseite mithilfe eines *iframes* einzubetten und somit auf der gleichen Seite darzustellen. Das ist aber schnell unübersichtlich geworden.

Zusätzlich kam das Feedback der zukünftigen Nutzergruppe, dass sich dies "'unnatürlich" anfühle, denn von Suchmaschinen wie *Google* ist man es gewöhnt, dass der Nutzer entweder komplett auf die Seite weitergeleitet oder diese in einem neuen Fenster geöffnet wird. In beiden Fällen verlasse der User aber die ursprüngliche Suchmaschine selbst. Bei einem Klick auf das Ergebnis einen neuen Browser Tab aufzumachen und die Seite dort zu laden hat sich daher als bessere Variante herausgestellt. Ergebnisse können so länger offen gelassen oder als Lesezeichen gespeichert werden.

Dafür wurde eine Funktion (siehe 5.11) implementiert, die zuerst die Adressen der einzelner Ergebnisse zusammensetzt und dann mithilfe der *window.open* Funktion ein neues Tab öffnet.

KAPITEL 6

EVALUATION DER QUALITÄT DER SUCHERGEBNISSE

Nachdem im vorangehenden Kapitel die Applikation selbst konzipiert wurde, soll nun evaluiert werden, welcher Scoringalgorithmus für diese der Beste ist. Hierfür werden die drei verschiedenen Modelle *Okapi BM25*, *Divergence from Randomness (DFR)* und *Divergence from Independence (DFI)* miteinander verglichen. Diese Modelle sind in Abschnitt 2.1.4 genauer erläutert.

6.1 Vorgehen

6.1.1 Wahl der zu untersuchenden Scoringmodelle

Es werden drei verschiedene Scoringfunktionen miteinander verglichen, die bereits in ES implementiert sind.

Seit 2016 ist *Okapi BM25* die standardmäßige Scoringfunktion ab ES 5.0, die mit anderen Funktionen verglichen wird (vgl. [ela16]).

Manning et al. [MRS08, S.233] meint, dass sich in Studien für k_1 eine Wert zwischen 1.2 und 2 und für $b = 0.75$ als sinnvoll herausgestellt hat.

Eine zweite Funktion wird aus dem *Divergence from Randomness (DFR)* Framework erzeugt. Als Basismodell wird das Bose-Einstein-Modell, einen Laplace Aftereffekt und eine Längennormalisierung auf Basis gleichmäßiger Verteilung der Termhäufigkeit ausgewählt.

Als letztes werden diese mit dem *Divergence from Independence (DFI)* Modell mit *Standardization* wie dies von Dincer [Din12, S. 1] vorgeschlagen wird, verglichen. Auf Seite 2 wird beschrieben, dass sich für (Internet) Suchmaschinen, die sich durch kurze Abfragelängen auszeichnen, das Verfahren mit *Standardization* am besten eignet, um hohe Recall und Precision Werte zu erhalten.

Tabelle 6.1 zeigt zusammengefasst, welche Modelle und Parameter verwendet wurden.

Modell	**Parameter**
BM25	$k_1 = 1.2$ $b = 0.75$
DFR	*Basis Modell*: Bose-Einstein *Aftereffekt*: Laplace *Normalisierung*: h_1 (Gleichmäßige Verteilung der Termhäufigkeit)
DFI	Metrik: standardisiert

Tabelle 6.1: Scoringfunktionen und Parameter

6.1.2 Wahl eines Fragenkorpus

Laut Manning et al. [MRS08, S. 152] sind mindestens 50 Queries notwendig, um aussagekräftige Zahlen über die Antwortqualität eines IR Systems zu erhalten. Streng genommen spricht der Manning et. al von einem *Information Need*, noch nicht von einer konkreten Abfrage, sondern von Informationsbedürfnissen, die dann in Form einer Query an das System gestellt werden. Da aber für jeden *Information Need* eine (Ab)frage (Query) erstellt

wird und dieser in ihr leicht erkennbar ist, werden diese zwei Begriffe fortan synonym verwendet.

Um möglichst realistische Fragen zu erhalten, wurde die eigentliche Nutzergruppe der PLM Administratoren nach ihrem Informationsbedarf befragt. Aus diesen Antworten und eigenen Erfahrungen wurden 50 Fragen definiert, bei denen darauf geachtet wurde, eine möglichst hohe Diversität der Wissensfelder im Teamcenter Umfeld zu erreichen.

Beispiele für solche Fragen sind "'*How to install teamcenter*", "'*log file locations*" oder "'*Cleaning database*". Eine detaillierte Liste aller Fragen kann im Anhang unter A.2 eingesehen werden.

6.1.3 Durchführung der Evaluation

Die 50 Abfragen wurden nun an das System gestellt und jeweils die 20 besten Antworten notiert. Dieser Vorgang wurde für alle drei Modelle durchgeführt, was später 1000 Relevanzbeurteilungen pro Bewertungsmodell entspricht.

Judging der Ergebnisse Die Ergebnisse wurden vom Autor selbst bewertet. Aufgrund der begrenzten Ressourcen konnten keine weiteren Bewerter mit einbezogen werden, wie dies in ausführlicheren Tests der Regelfall ist. Benutzt wurde hierbei ein binäres System, das einem Dokument entweder das Prädikat *Relevant* (1) oder *Nicht Relevant* (0) zuordnet.

Auf eine feinere Graduierung wurde zum einen aus zeitlichen Gründen, zum anderen, weil sich diese Dokumente schwer einstufen lassen, verzichtet. Es handelt sich um sachliche Dokumentationen, die entweder eine Lösung zu dem Informationsbedürfnis geben, oder nicht.

Pooling Da die genaue Anzahl der relevanten Dokumente unbekannt ist, muss auf die *Pooling*-Technik (siehe 2.4.3) zurückgegriffen werden. Hierbei wurden die 20 Ergebnisse

je Abfrage zuerst nach Relevanz beurteilt und dann mit den relevanten Dokumenten aus den anderen Algorithmen vereinigt. Die Mächtigkeit der Vereinigungsmenge entspricht einer Schätzung für die Anzahl der relevanten Dokumente mit der Annahme, dass jedes relevante Dokument von mindestens *einem* System gefunden wird.

6.2 Ergebnisse

Im folgenden werden die Ergebnisse ausgewertet. Hierfür wurde die Metrik des *MAP* und die zugehörigen AP@K-Werte berechnet. Diese Metrik hat sich in der TREC Community als Standard etabliert. (vgl. Manning et al. [MRS08, S. 159])

6.2.1 Average Precision at Position k (AP@k)

Abbildung 6.1 zeigt die Average Precision Werte auf den einzelnen Positionen.

Es wird deutlich, dass die *Average Precision* an der Position k von DFI den anderen Funktionen unterlegen ist, denn die DFI Kurve liegt immer 0.05 bis 0.1 Punkte unterhalb der anderen. Ab Position 15 nähert sich diese den anderen Funktionen von unten an. Dies kann so interpretiert werden, dass die relevanten Dokumente von DFI später gefunden werden, wohingegen bei den anderen Funktionen die meisten sich bereits weiter vorne eingereiht haben und hinten unwichtigere sind.

BM25 und DFR liefern an der ersten Stelle eine durchschnittliche Precision von etwa 0.85 bis 0.89. Dies bedeutet, dass das erste Dokument fast immer relevant ist. DFI hingegen liefert dort nur einen Wert von 0.7, was im Vergleich deutlich schlechter ist.

Es kann festgestellt werden, dass ist die *Average Precision* von BM25 und DFR ähnlich ist. DFR liefert zwar auf den ersten drei Rängen bessere Präzisionen, liegt aber von Rang vier bis 13 gleichauf mit BM25 und weist am Ende schlechtere durchschnittliche

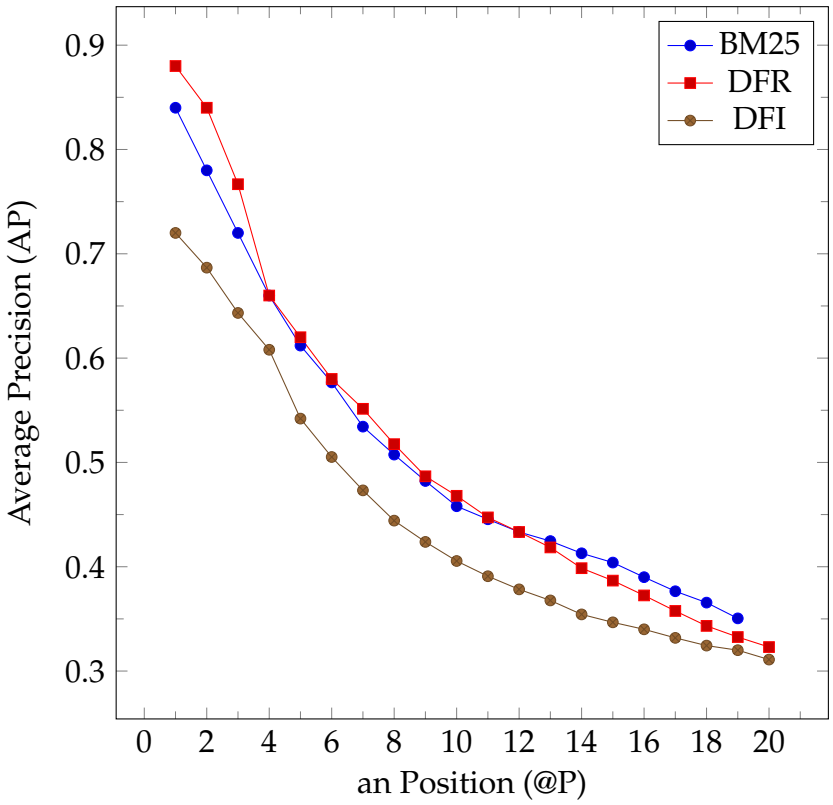

Abbildung 6.1: Average Precision at Position k (AP@k)

Präzisionen auf.

Zusammenfassend lässt sich sagen, dass BM25 und DFR in etwa gleich gute Ergebnisse liefern, wohingegen DFI tendenziell deutlich schlechtere Ergebnisse liefert.

6.2.2 MAP

Auch die *Mean Average Precision (MAP)* zeigt, dass die Qualität von DFI schlechter als von DFR und BM25 ist. Mit einem Wert von 0.5647 liegt dieser signifikant unter den Werten von 0.6739 bei BM25 und 0.6625 bei DFR. Interessanterweise hat BM25, obwohl diese Funktion an den ersten Stellen schlechtere P@K Werte geliefert hat, einen besseren Wert als DFR.

Die Werte finden sich in Tabelle 6.2.

	BM25	DFR	DFI
MAP	0.6739	0.6625	0.5647

Tabelle 6.2: MAP/GMAP Werte

6.2.3 Zusammenfassung

Zusammenfassend lässt sich sagen, dass DFI deutlich schlechtere Ergebnisse liefert, als BM25 und DFR. Die letzten beiden weisen eine ähnliche Antwortqualität auf: DFR liefert zwar auf den ersten Positionen bessere Präzisionen, insgesamt aber einen schlechteren MAP-Wert als BM25.

Daher wird für das PLM IR System die BM25 Funktion verwendet.

6.2.4 Reaktionszeit des Systems

Neben der eigentlichen Qualität der Antworten gibt es noch einen weiteren Parameter, der die Güte eines IR Systems beschreibt: den **Time Lag**, der durchschnittlichen Antwortzeit des Systems auf eine Abfrage.

In einem Testsystem (Ubuntu VM, 11GB RAM, Intel i7-2720QM mit 2.20GHz) lagen die Antwortzeiten bei 30k indexierten Dokumenten im Durchschnitt bei etwa 33.58ms (siehe Anhang A.2.1).

Hierfür wurde die Reaktionszeit mit BM25 Scoring auf unsere 50 Fragen gemittelt. Die Varianz betrug hierbei 16.13 bei einem maximalen Wert von 70ms und einem schnellsten Wert von 16ms. Damit erhält der Nutzer bereits während der Eingabe der Frage ohne merkbare Verzögerungen die Ergebnisse in Echtzeit.

KAPITEL 7

SCHLUSS

7.1 Erfüllungsgrad der Anforderungen

Nachdem die Entwicklung und Evaluation soweit abgeschlossen ist, wird untersucht inwieweit die Anforderungen aus Kapitel 3 erfüllt wurden.

Bis auf zwei Punkte konnten alle erfüllt werden. Tabelle 7.1 zeigt dies.

AF-1-1-4 Eine automatische Aktualisierung der Inhalte wurde aus zeitlichen Gründen nicht mehr implementiert. Jedoch besitzen alle Tools die Möglichkeit, die neusten Daten aus den Quellen manuell nachzuladen. Da sich die Dokumentationen nicht oft ändern ist das vorerst ausreichend.

AF 1-2-1 Auch eine Pipeline, die Binärdaten verarbeiten kann, konnte nicht mehr implementiert werden. Jedoch wurde ein Konzept entwickelt, wie diese Funktionalität konkret implementiert werden kann. Dieses wurde im Abschnitt 5.3.4 diskutiert.

Tabelle 7.1: Erfüllung der Anforderungen

Anforderung	Kurzbeschreibung	Erfüllt
AF-1-1-1	Siemens PLM Documentation Server	Ja
AF-1-1-2	GTAC Solution Center	Ja
AF-1-1-3	Webseiten	Ja
AF-1-1-4	Weitere Dokumente	Nur Konzeptionell
AF-1-2-1	Automatische Aktualisierung	Nein
AF-2-1-1	Übersichtliche Darstellung	Ja
AF-2-1-2	Weiterleitung bei Interaktion	Ja
AF-2-1-3	Aufbereitete Darstellung	Ja
AF-2-1-4	Pagination	Ja
AF-2-2-1	Suchzeile	Ja
AF-2-2-2	Echtzeitaktualisierung der Ergebnisse	Ja
AF-2-2-3	Erweiterte Suchfunktion	Ja
AF-2-2-4	Filtermöglichkeit	Ja
AF-2-2-5	Optimierung der Suchergebnisse	Ja
AF-2-2-6	Suche auf alle Quellen	Ja
AN-1	Durchschnittliche Suchzeit <200ms	ja
AN-2	Optimale Suchqualität	Ja
AT-1	Windows 7 Unterstützung	Ja

7.2 Fazit

In dieser Arbeit wurde ein IR System für PLM Administratoren entwickelt und festgestellt, dass BM25 die beste Ergebnisqualität für dieses liefert. Damit wurden die Dokumentationen im PLM Umfeld zentralisiert. Hätte man vor einigen Jahrhunderten diese Daten noch gemeinsam in einer Bibliothek ablegen und per Hand indexieren müssen, hat uns heute diese Arbeit ein Indexserver abgenommen.

Es muss sich nun keine Sorgen mehr gemacht werden, dass ein kleines Feuer unser gesamtes Wissen vernichtet, wie dies in der *Großen Bibliothek von Alexandria* der Fall war, denn die *Shards* und *Replikas* können auf mehrere Server verteilt sein. Somit sind alle Daten sind dezentral auf verschiedene Server gespeichert und gesichert.

LITERATURVERZEICHNIS

[Apa16] Apache Software Foundation: *Apache Lucene Core*. https://lucene.apache.org/core/index.html, 2016. – (26/07/18)

[Apa17] Apache Software Foundation: *Solr Features*. http://lucene.apache.org/solr/features.html, 2017. – (18/09/18)

[BCC16] Büttcher, Stefan ; Clarke, Charles L. A. ; Cormack, Gordon V.: *Information Retrieval - Implementing and Evaluating Search Engines*. Cambridge : MIT Press, 2016

[CIM10] CIMData, Inc.: *Die einheitliche Umgebung von Teamcenter - Whitepaper*. Michigan : CIMData, 2010

[CMS11] Croft, Bruce ; Metzler, Donald ; Strohman, Trevor: *Search Engines - Information Retrieval in Practice*. Amsterdam : Pearson Education, 2011

[Con18] Connelly, Shane: *Practical BM25 - Part 2:The BM25 Algorithm and its Variables*. https://www.elastic.co/blog/practical-bm25-part-2-the-bm25-algorithm-and-its-variables, 2018. – (20/09/18)

[Din12] Dincer, Bekir T.: *IRRA at TREC 2012: Divergence from Independence (DFI)*. https://trec.nist.gov/pubs/trec21/papers/irra.web.nb.pdf, 2012. – (10/08/18)

[Dom13] Domenech, Juan: *Where is Google Bot?* http://blog.domenech.org/2013/09/where-is-googlebot-crawler-located.html, 2013. – (17/09/18)

[ela16] ELASTIC: *Practical BM25 - Part 1: How Shards Affect Relevance Scoring in Elasticsearch*. https://www.elastic.co/blog/practical-bm25-part-1-how-shards-affect-relevance-scoring-in-elasticsearch, 2016. – (05/08/18)

[ela18a] ELASTIC: *Basic Concepts*. https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html#_basic_concepts, 2018. – (18/09/18)

[ela18b] ELASTIC: *Docs - Character Filters*. https://www.elastic.co/guide/en/elasticsearch/reference/6.x/analysis-charfilters.html, 2018. – (18/09/18)

[ela18c] ELASTIC: *Index API - Automatic Index Creation*. https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-index_.html, 2018. – (10/09/18)

[ela18d] ELASTIC: *Der Open Source Elastic Stack*. https://www.elastic.co/de/products, 2018. – (18/09/18)

[ela18e] ELASTIC: *Our Story*. https://www.elastic.co/about/history-of-elasticsearch, 2018. – (04/09/18)

[ela18f] ELASTIC: *Pluggable Similarity Algorithms*. https://www.elastic.co/guide/en/elasticsearch/guide/current/pluggable-similarites.html, 2018. – (18/09/18)

[ela18g] ELASTIC: *Removal of Mapping Types*. https://www.elastic.co/guide/en/elasticsearch/reference/6.x/removal-of-types.html, 2018. – (18/09/18)

[ela18h] ELASTIC: *Similarity module*. https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-similarity.html, 2018. – (20/09/18)

[Fac18] FACEBOOK INC.: *React*. https://reactjs.org/, 2018. – (18/09/18)

[GHR14] GHEORGHE, Radu ; HINMAN, Matthew L. ; RUSO, Roy: *Elasticsearch in Action*. Manning Publications, 2014

[Goo18] Google LLC: *Google Trends: ElasticSearch vs. Solr*. https://trends.google.de/trends/explore?geo=DE&q=ElasticSearch,SOLR, 2018. – (30/07/18)

[Hei11] Heise Online: *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, 2011. – (26/07/18)

[Hen08] Henrich, Andreas: *Information Retrieval 1 - Grundlagen, Modelle und Anwendungen*. 1.2. Bamberg : Lehrstuhl Medieninformatik der Otto-Friedrich-Universität Bamberg, 2008

[Lic11] Lichtinger, Peter: *Die Bibliothek von Alexandria*. http://imperiumromanum.com/kultur/bildung/bibliothek_alexandria_01.htm, 2011. – (15/08/18)

[MHG10] McCandless, Michael ; Hatcher, Erik ; Gospodnetic, Otis: *Lucene in Action*. Stamford : Manning Publications Co., 2010

[MRS08] Manning, Christopher D. ; Raghavan, Prabhakar ; Schütze, Hinrich: *Introduction to Information Retrieval*. Cambridge, England : Cambridge University Press, 2008

[Net16] Neto, Ribeiro: *DFR - Divergence From Randomness*. 2016. – (21/09/18)

[Par17] Parlo: *The 3 essentials of AI Bots for IT Help Desk*. https://chatbotsmagazine.com/the-3-essentials-of-ai-bots-for-it-help-desk-9bce2ffa4446, 2017. – (20/09/18)

[Pon18] Pons-Verlag: *Information Retrieval*. https://de.pons.com/%C3%BCbersetzung/englisch-deutsch/retrieval, 2018. – (23/07/18)

[Rai90] Rainer, Kuhlen: *Zum Stand pragmatischer Forschung in der Informationswissenschaft*. Konstanz : Universitätsverlag Konstanz, S. 13-18, 1990

[RGR17] Reinsel, David ; Gantz, John ; Ryding, John: *The Evolution of Data to Life-Critical - Don't Focus on the Data That's Big*. Framingham : IDC, 2017

[RS17] Russ, McKendrick ; Scott, Gallagher: *Mastering Docker - Master this widely used containerization tool*. 2017. – (20/09/18)

[Ryt18] Ryte: *Vertical Search*. https://de.ryte.com/wiki/Vertical_Search, 2018. – (21/09/18)

[Sch15] Schuh, Guenther: *PLM (Product Lifefycle Management)*. http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/informationssysteme/Sektorspezifische-Anwendungssysteme/Product-Life-Cycle-Management/index.html, 2015. – (27/07/18)

[Sha86] Shakespeare, William: *Macbeth - Englisch/Deutsch*. Reclam, 1986

[Ton18] Tong, Zachary: *What is an Elasticsearch Index*. https://www.elastic.co/de/blog/what-is-an-elasticsearch-index, 2018. – (18/09/18)

[TPB14] Trotman, Andrew ; Puurula, Antti ; Burgess, Blake: *Improvements to BM25 and Language Models Examined*. http://www.cs.otago.ac.nz/homepages/andrew/papers/2014-2.pdf, 2014. – (18/09/18)

[Tur14] Turnbull, James: *The Docker Book*. North Corolina, 2014

ANHANG A

ANHANG

A.1 Docker Compose File für Elastic PLM Stack

```
version: '2'
services:
        elasticsearch:
                build:
                context: elasticsearch/
                volumes:
                - ./elasticsearch/config/elasticsearch.yml:
                        /usr/share/elasticsearch/config/elasticsearch.yml
                ports:
                - "9200:9200"
                - "9300:9300"
                environment:
                ES_JAVA_OPTS: "-Xmx256m -Xms256m"
                networks:
                - elk
        plmdocsrv:
                image: atos/tcdocs11_2
                ports:
                - "8282:8282"
                - "8383:8383"
                networks:
                - elk
```

```
23          logstash:
24                  build:
25                  context: logstash/
26                  volumes:
27                  - ./logstash/config/logstash.yml:/usr/share/logstash/config/logstash.yml
28                  - ./logstash/pipeline:/usr/share/logstash/pipeline
29                  ports:
30                  - "5000:5000"
31                  environment:
32                  LS_JAVA_OPTS: "-Xmx256m -Xms256m"
33                  networks:
34                  - elk
35                  depends_on:
36                  - elasticsearch
37          kibana:
38                  build:
39                  context: kibana/
40                  volumes:
41                  - ./kibana/config/:/usr/share/kibana/config:ro
42                  ports:
43                  - "5601:5601"
44                  networks:
45                  - elk
46                  depends_on:
47                  - elasticsearch
48  networks:
49          elk:
50          driver: bridge
```

A.2 Rohdaten der Evaluation

Die folgende Tabelle listet alle Fragen und AP Werte auf, die im Verlauf der Evaluation berechnet wurden. Hierbei wurden die Scoringalgorithmen *Okapi BM25*, *Divergence From Randomness (Bose-Einstein Basic Model, Laplace Aftereffect und H1 length normalization* und *Divergence from Independence (Standardized)* auf dem Datensatz des PLM Dokumentation Servers verglichen.

A.2.1 Fragen und AP Werte

Questions	BM25	DFR	DFI	Lag [ms]
How to install teamcenter	0,3440	0,3786	0,4646	32
What is the fcc	0,8218	0,8734	0,2038	28
Teamcener Architecture	0,1976	0,4572	0,7354	26
Markup layers	0,6521	0,5363	0,4395	26
Upgrade teamenter	0,8033	0,4173	0,7377	34
Manage teamcenter data model	0,6883	0,6878	0,4365	80
Hide item types when user create new item	1,0000	1,0000	0,5000	31
Performing maintance with tem	0,5500	0,5500	0,5134	26
Cleaning database	1,0000	0,9762	0,1083	35
Using teamcenter help	0,6423	0,7616	0,5446	56
Summary stylesheet	1,0000	1,0000	1,0000	30
Performing maintance with tem	0,5500	0,5500	0,5134	26
What is FMS	0,6999	0,4646	0,2200	23
Restart fcc process	0,9594	0,7705	0,8295	21
Customize teamcenter	1,0000	1,0000	1,0000	20
Change menus	0,7707	0,7469	0,7044	24
Create a form for richt client	0,8076	0,4205	0,1389	48
Rich client installation	0,6546	0,6762	0,7388	44
Improve startup performance	0,7269	0,6193	0,3417	33
Set teamcenter privileges	0,3820	0,3446	0,0294	24
Woring with thin client	0,4076	0,5204	0,7270	18
Manage teamcenter workflow	0,6911	0,4885	0,4072	62
Use product structure	0,5190	0,4494	0,3071	22
Send teamcenter email	0,7500	0,5556	0,6667	22
What are programs	0,8333	0,8167	0,7500	18
Creating reports	0,5507	0,5593	0,6462	41
Create Visual report	0,6531	0,5000	0,4760	32
Use tem to upgrade	0,7576	0,7381	0,7778	18
Using Business Modeler DIE	0,6161	0,5355	0,5252	20
Change Language Settings	0,7455	0,8713	0,9581	26

Table A.1 Fortsetzung

Questions	BM25	DFR	DFI	Lag [ms]
Configure database	0,8714	0,6533	0,8100	16
Use Active Workspace	0,9667	0,9667	0,9429	18
Starting eclipse	0,4444	0,4693	0,3889	25
What is a transient volume	0,3937	0,4972	0,5042	58
Log file location	0,6938	0,5464	0,3607	32
Teamcenter System requirements	0,6160	0,7736	0,7092	25
Improve system performance	0,7624	0,6488	0,8557	21
Functionalities of Thin Client	0,1902	0,1726	0,4263	26
Thin client differ rich client	0,5909	0,5576	0,4881	47
Create workflow process template	0,6234	0,9427	0,4920	32
What is a teamcenter workflow	0,3744	0,2659	0,4216	35
How does teamcenter email work	0,3667	0,4167	0,2500	77
What is the dispatcher client	1,0000	1,0000	0,6325	37
Create volumes	0,6262	0,7048	0,8064	20
Classify product data	0,4439	0,7946	0,0602	62
Preferences for programs	0,7500	0,8333	0,5000	40
Teamcenter features	1,0000	0,9500	1,0000	45
Remove components	0,8578	0,9722	0,6432	20
Uninstall Teamcenter	0,7441	0,8418	0,8138	28
Manage teamcenter data model	0,6042	0,8542	0,6875	78
Mean Average Precision (MAP)	**0,6739**	**0,6625**	**0,5647**	$\varnothing = 33.58$
Geometric Mean Average Precision (GMAP)	**0,6339**	**0,6231**	**0,4794**	$\sigma = 16.13$

A.2.2 Average Relevance und Precision at k

Hier finden sich die Daten, auf denen die Grafik 6.1 und basiert.

Pos. k	**BM25**-AP@k	**DFR**-AP@	**DFI**-AP@k
1	0.8400	0.8800	0.7200
2	0.7800	0.8400	0.6867
3	0.7200	0.7667	0.6433
4	0.6600	0.6600	0.6080
5	0.6120	0.6200	0.5420
6	0.5767	0.5800	0.5052
7	0.5343	0.5514	0.4732
8	0.5075	0.5175	0.4442
9	0.4822	0.4867	0.4238
10	0.4580	0.4680	0.4055
11	0.4455	0.4473	0.3909
12	0.4333	0.4333	0.3783
13	0.4246	0.4185	0.3677
14	0.4129	0.3986	0.3543
15	0.4040	0.3867	0.3467
16	0.3900	0.3725	0.3400
17	0.3765	0.3576	0.3318
18	0.3656	0.3433	0.3244
19	0.3505	0.3326	0.3200
20	0.3390	0.3230	0.3110

Tabelle A.2: Average Relevance (AR@k) und Average Precision at K (AP@k) für BM25, DFI und DFR