FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Performance-Steigerung in Semantik-basierten Abfrage-Systemen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Simon Wagner
Matrikelnummer 0456124

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber

Wien, 19.04.2014

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Simon Wagner
Stockhofstraße 33a, 4020 Linz

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

(Ort, Datum)	(Unterschrift Verfasser/in)

Danksagung

Ich möchte mich bei all jenen bedanken, die mir direkt oder indirekt bei der Entstehung dieser Diplomarbeit beigestanden haben. Besonderer Dank gebührt vor allem meinem Betreuer, Herrn Prof. Dipl.-Ing. Dr.techn. Andreas Rauber, der mir in allen Phasen meiner Arbeit mit Rat und Tat unterstützt hat.

Ich bedanke mich ebenfalls bei Prof. Dipl.-Ing. Dr. Josef Küng, der mir als Mentor zur Seite gestanden ist.

Diese Arbeit widme ich meinen Eltern und meiner Großmutter, die mir dieses Studium durch ihre finanzielle und persönliche Unterstützung ermöglicht haben. Danke!

Kurzfassung

Ein Frage-Antwort System benutzt Algorithmen um semantisch ähnliche Fragen zu finden. Diese benötigen durch ihre hohe algorithmische Komplexität mehrere Sekunden für eine Berechnung. Das Matching soll jedoch in einer Anwendung eingesetzt werden, in der dem Benutzer oder der Benutzerin die Resultate in Echtzeit präsentiert werden.

Damit diese Anforderung erfüllt werden kann schlägt diese Arbeit eine Vorberechnung von Ähnlichkeiten vor, deren Ergebnisse in einer geeigneten Datenstruktur abgelegt und anschließend dazu benutzt werden um in wenigen Millisekunden Antworten generieren zu können. Die Matching-Algorithmen verwenden Schlüsselwörter für ihre Berechnungen. Folglich entspricht die Menge aller Kombinationen dieser Keywords allen theoretisch denkbaren Eingaben. Da nicht all diese Möglichkeiten berechnet werden können, ist ein zentraler Punkt, eine sinnvolle Einschränkung zu treffen. Der Ansatz, der im Zug dieser Arbeit vorgestellt wird, verbindet nur Schlüsselwörter miteinander, bei denen die Schnittmenge der Matches nicht leer ist.

Alle berechneten Ähnlichkeiten werden in einer Lucene-Indexstruktur gespeichert. Um dem Benutzer oder der Benutzerin die ähnlichsten Fragen präsentieren zu können, auch wenn noch kein Schlüsselwort komplett eingetippt wurde oder Tippfehler in der Eingabe vorhanden sind, werden die Vorberechnungen zusätzlich mit einer syntaktischen Suche kombiniert.

Abstract

A given Query Answering System uses algorithms in order to find semantically similar questions. Due to the high algorithmic complexity the calculations require a couple of seconds. However, the matching should be used in an application which should be able to present the results in real time.

To meet these requirements this thesis proposes some sort of pre-calculation of similarities. The results are stored in a suitable data structure and as a result they are used to generate answers in a few milliseconds. For their calculations the matching algorithms are using keywords. Consequently, the set of all combinations of keywords would contain all theoretically possible inputs. Due to the fact that it is impossible to calculate all those some useful restrictions are needed. The approach presented in this work only combines keywords with an intersecting set of matches.

All calculated similarities are stored in a Lucene index structure. In order to present the most similar questions to the user, even if the input contains no keyword or contains typographical errors, these pre-computations are combined with the results of a syntactic search.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

KAPITEL 1

Einleitung

Aufgrund der stetig wachsenden Menge an Daten wird ist es immer wichtiger etwas wiederzufinden. In den letzten 15 Jahren haben sich Suchmaschinen zu einem nicht mehr wegzudenkenden Werkzeug für viele Bereiche des täglichen Lebens entwickelt. Wissen galt seit jeher als wichtig, Suchmaschinen haben laut [1] allerdings diese Tatsache über den Haufen geworfen. Mit Mobiltelefonen, Tablets und Laptops, die immer in Griffreichweite sind, ist es heute oft interessanter zu wissen wie und wo man etwas findet, als die Fakten im Gehirn zu speichern. Mit Google oder Bing kann heute fast jede Webseite zu jedem beliebigen Thema schnell und bequem gefunden werden. Betrachtet man allerdings klassische Frage-Antwort Seiten, so ist die dort vorhandene Suche oft sehr eingeschränkt verwendbar und basiert in vielen Fällen auf einem einfachen Textvergleich.

Mit dem S-Mate[1] Projekt wird ein neuartiger Ansatz verfolgt, in dem die Suche nach ähnlichen Fragen auch mit Hilfe semantischer Algorithmen durchgeführt wird. Die Zielvorgabe, eine möglichst hohe Qualität bei den Ähnlichkeitsberechnungen erreichen zu wollen, kann nur auf Kosten der Performance erreicht werden. Im Zuge des Projekts werden unterschiedliche vorhandene Methoden eingesetzt und neue entwickelt um an möglichst optimale Ergebnisse zu kommen. Diese Analyse wird keine (bzw. kaum) Rücksicht auf die Geschwindigkeit und Skalierbarkeit der Lösung nehmen, sondern das Hauptaugenmerk darauf legen, unabhängig von der Eingabe immer eine zufriedenstellende Rückgabe liefern zu können.

In dieser Diplomarbeit sollen Methoden gefunden und untersucht werden, die das Finden von ähnlichen Fragen in Echtzeit ermöglichen. Dabei werden die existierenden semantischen Algorithmen als Blackbox betrachtet. Bei der Implementierung war Lucene ein wichtiges Framework, daher findet sich in Kapitel 2 eine kurze Einführung. In Kapitel 3 wird das S-Mate Projekt kurz vorgestellt und die wichtigsten Punkte erwähnt, die zu dieser Diplomarbeit geführt haben. Der Hauptteil der Arbeit konzentriert sich auf die Implementierung der Verbesserung der Suche und findet sich in Kapitel 4. Dort werden auch die Ergebnisse analysiert und mögliche

[1] Scio-Matching Tool Engineering

zukünftige Verbesserungen präsentiert. Schlussendlich bietet Kapitel 5 eine kurze Zusammenfassung dieser Arbeit und ein Resümee über die gewonnenen Erkenntnisse.

KAPITEL 2

Lucene

Da im Zuge der Diplomarbeit diese Software zum Einsatz kommt, werden in diesem Kapitel die wichtigsten Elemente des Frameworks kurz angesprochen. Die meisten Informationen dazu wurden [12] entnommen, da dieses eines der wenigen Bücher darstellt, das sich vom Anfang bis zum Ende allen Aspekten dieses Frameworks widmet.

Bei Lucene handelt es sich um eine eigenständige Java Bibliothek im Information Retrieval Bereich, die sich durch Performance und Skalierbarkeit auszeichnet. Durch die flexible API kommt sie in verschiedenen Bereichen und dutzenden Anwendungen zum Einsatz[1].

Der naivste Ansatz für die Textsuche vergleicht einen Suchstring der Länge M sequentiell an jeder Stelle mit einem Text mit der Länge N. Die Laufzeit wäre im schlechtesten Fall, wenn sich das Gesuchte am Ende befindet oder gar nicht enthalten ist, $O(N * M)$. Als Verbesserung beinhaltet ein von Knuth, Morris und Pratt vorgestellter Algorithmus [3] einen möglichst weiten Shift bei einem Missmatch. Aus diesem Grund kann die Laufzeit auf $O(N + M)$ reduziert werden. Unabhängig vom gewählten Ansatz hängt die Zeit, die für eine Suche im schlechtesten Fall benötigt wird, linear von der Größe des Textes ab.

Ohne Vorberechnung ist es natürlich nicht möglich unter $O(N)$ zu gelangen. Der Ansatz, den auch Lucene verwendet, ist eine Umwandlung der zu Grunde liegenden Texte in einen sogenannten Index um eine performantere Suche zu ermöglichen.

2.1 Dokumente

Das Dokument in Lucene entspricht in etwa einem Datensatz in einer Datenbanktabelle und besteht wiederum aus mehreren Feldern. Es gibt allerdings kein fixes Schema, das innerhalb eines Indexes eingehalten werden muss. Das bedeutet, dass jedes Dokument anders aufgebaut sein könnte, um beispielsweise verschiedene Entitäten zu repräsentieren. Felder haben außer einem Inhalt (Text, numerische bzw. binäre Daten) einen Namen und Informationen für die Verarbeitung. Mit letzteren kann festgelegt werden ob ein Feld indexiert wird, was wiederum die

[1] http://wiki.apache.org/lucene-java/PoweredBy

Voraussetzung für eine spätere Suche ist. Es gibt außerdem eine Option, die Termvektoren für komplexere Suchen abzulegen. Um Speicherplatz zu sparen müssen die eigentlichen Rohdaten nicht abgelegt werden, wenn sie später nicht mehr benötigt werden.

2.2 Index Operationen

Für den Index stehen die üblichen drei Basisoperationen zur Verfügung. Während mit der Hilfe von `addDocument` neue Dokumente hinzugefügt werden, wird `deleteDocument` zum Entfernen vorhandener Einträge verwendet. Um Änderung in abgelegten Dokumenten vornehmen zu können existiert mit `updateDocument` diese dritte Funktionalität, die allerdings nur als Kombination der ersten beiden implementiert ist.

2.3 Analyse

Beim Einfügen eines Dokuments in einen Index wird der Inhalt jedes Feldes in Token zerlegt, auf die anschließend verschiedene Operationen angewendet werden.

Die von den Analyzern erzeugten Tokens werden sowohl für die Erzeugung des Indexes, als auch für die Suche benötigt. Lucene bietet eine Reihe von fertigen Implementierungen an, die direkt verwendet werden können. Natürlich besteht des weiteren die Möglichkeit eigene Analyzer zu bauen, die bestehende und/oder neue Filter verwenden.

Das Ergebnis solcher Analyzer wird im folgenden Beispiel anhand eines Satzes gezeigt. Der `WhitespaceAnalyzer` entfernt alle Sonderzeichen und teilt den String an den Leerzeichen auf. Etwas komplexer ist die Funktionalität des `SimpleAnlyzer`, der einen String an Zeichen trennt, die keine Buchstaben darstellen, und jedes Token in Kleinbuchstaben umwandelt. Das dritte Beispiel verwendet zwar auch eine vorhandene Implementierung, zeigt allerdings welche Möglichkeiten bei der Entwicklung solcher Lösungen bestehen. Der `StopAnalyzer` erweitert den `SimpleAnlyzer` um eine Funktionalität, die häufig vorkommende Wörter, sogenannte Stoppwörter, aus dem Tokenstream entfernt. Unter den Analyzern im Lucene-Core steckt im `StandardAnalyzer` die meiste Logik. Es werden nicht nur Stoppwörter entfernt, sondern auch komplexere Strings wie Firmennamen, URLs und eMail Adressen als komplette Tokens erkannt.

```
Beispielsatz: "Wie viel Obst und Gemüse soll ich www.essen.com?"
  WhitespaceAnalyzer:
    [Wie] [viel] [Obst] [und] [Gemüse] [soll] [ich] [www.essen.com?]
  SimpleAnlyzer:
    [wie] [viel] [obst] [und] [gemüse] [soll] [ich] [www] [essen] [com]
  StopAnalyzer:
    [obst] [gemüse] [www] [essen] [com]
  StandardAnalyzer:
    [obst] [gemüse] [www.essen.com]
```

2.4 Suche und Scoring

Die Suche in Lucene entspricht auf Code-Ebene einem Query Objekt. Das einfachste Beispiel ist die `TermQuery`, die einen Term enthält, der für ein bestimmtes Feld erfüllt sein soll. Eine Suche nach dem Term „obst“ in einer Indexstruktur, die das Feld „frage“ beinhaltet, besteht aus den zwei Zeilen die in Listing 2.1 zu sehen sind.

Listing 2.1: TermQuery Beispiel

```
1 Term t = new Term("frage", "obst");
2 Query query = new TermQuery(t);
```

Sobald mehr als ein Suchterm in einer Abfrage behandelt werden soll wird beispielsweise die `BooleanQuery` verwendet, der in weiterer Folge verschiedene Subqueries hinzugefügt werden. Soll zum Beispiel eine Frage gefunden werden, in der „obst“ vorkommt und „fleisch“ nicht, so kann dies wie in Listing 2.2 gezeigt aufgebaut sein.

Listing 2.2: Boolean Query Beispiel

```
1 Term t1 = new Term("frage", "obst");
2 Term t2 = new Term("frage", "fleisch");
3 BooleanQuery query = new BooleanQuery();
4 query.add(new TermQuery(t1), Occur.MUST);
5 query.add(new TermQuery(t2), Occur.MUST_NOT);
```

Jedes Dokument (d), welches einen Match bei einer Suche mit einer bestimmten Query (q) erzeugt, bekommt eine Bewertung mit Hilfe einer Score Funktion.

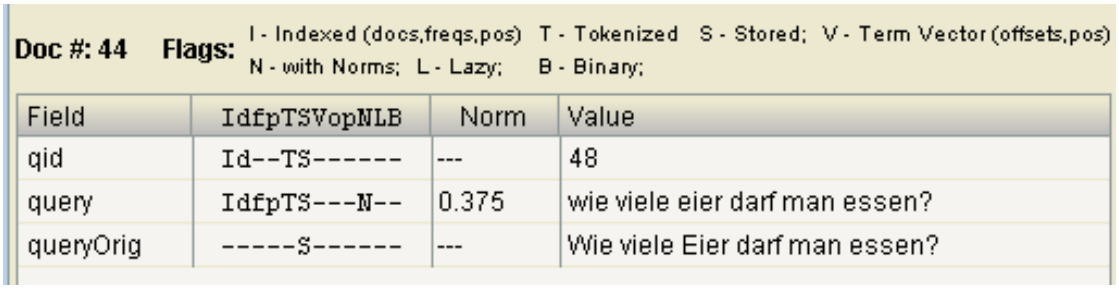
$score(q,d) = \sum_{t\,in\,q}(tf(t\,in\,d) * idf(t)^2 * t.boost() * norm(t,d)) * coord(q,d) * norm(q)$

Diese besteht aus einer Summe, die sich aus den Subscores der einzelnen Terms zusammensetzt, und wird anschließend mit einem Koordinierungsfaktor und einem Normalisierungswert multipliziert. Der Wert für jeden Term (t) setzt sich aus der Frequenz im Dokument ($tf(t\,in\,q)$), der Häufigkeit des Vorkommens ($idf(t)$), einem Boostwert und einem Normalisierungsfaktor zusammen.

2.5 Luke (Lucene Index Toolbox)

Es handelt sich hier um eine Software, die verschiedene Interaktionen mit Lucene ermöglicht. Da Luke in Kapitel 4 dazu verwendet wird, Dokumente eines Indexes visuell darzustellen, wird diese Funktionalität hier kurz erklärt.

Auf Abbildung 2.1 sieht man die einzelnen Felder eines Dokuments und mit welchen Flags diese abgelegt sind. In diesem Fall sind die ersten beiden Werte indexiert, was bedeutet, dass nach ihnen gesucht werden kann. Da der Marker **S** gesetzt ist, wird der zum Feld gehörende String abgelegt und kann später abgerufen werden. In diesem Fall ist nur für das Feld „query“ ein Normalisierungswert (Norm) gespeichert. Es ist auch ersichtlich, ob ein Term-Vektor zum Feld existiert. Auch für Lazy-Load und binäre Felder gibt es in der Dokumentenansicht eine Flag.



Doc #: 44 **Flags:** I - Indexed (docs,freqs,pos) T - Tokenized S - Stored; V - Term Vector (offsets,pos) N - with Norms; L - Lazy; B - Binary;

Field	IdfpTSVopNLB	Norm	Value
qid	Id--TS------	---	48
query	IdfpTS---N--	0.375	wie viele eier darf man essen?
queryOrig	-----S------	---	Wie viele Eier darf man essen?

Abbildung 2.1: Query Index Eintrag

KAPITEL 3

S-Mate

3.1 Einleitung

In diesem Kapitel wird das Projekt kurz vorgestellt, welches den Hintergrund der Diplomarbeit darstellt. Bei „S-Mate“ handelt es sich um ein Frage-Antwort System, das komplexe und aufwändige Algorithmen für Ähnlichkeitsberechnungen verwendet.

3.2 Ziele

Die Idee hinter dem S-Mate[1] Projekt ist es, zu Fragen die am besten passende Antwort eines Experten oder einer Expertin zu finden bzw. die Erstellung dieser in Auftrag zu geben. Die Beantwortung soll allerdings nicht, wie es manche traditionelle Frage-Antwort Portale praktizieren, in Textform erfolgen, sondern als Videobotschaft. Aus diesem Grund ist die Verknüpfung vorhandener Fragen, Antworten, Benutzer, Benutzerinnen, Experten und Expertinnen bei einer gleichzeitig hohen Qualität und guten Skalierbarkeit das zentrale Thema dieses Projekts. Um diese Ziele zu erreichen soll neben verschiedenen semantischen Technologien auch künstliche Intelligenz eingesetzt werden. Das erste Ziel dieses Projekts dreht sich um die Qualität der Matchings, die in jedem Fall 90% betragen soll.

Eine weitere Zielvorgabe ist die automatische Verknüpfung von Inhalten. Wenn dem Benutzer oder der Benutzerin eine Antwort geboten wird, die objektiv gesehen optimal auf die Frage passt, bedeutet dies allerdings nicht, dass der Benutzer oder die Benutzerin den Inhalt auch versteht, da ihm vielleicht entsprechendes Vorwissen fehlt. Sogenannte „Fast Tracks“ sollen daher automatisch generiert werden und dem User die Lösung mit einer minimalen Reihenfolge an verknüpften Fachbotschaften präsentieren.

Das dritte Hauptziel ist die Wiederverwendbarkeit aller vorhandenen Informationen. Insbesondere geht es hier darum, dass bei neu gestellten Fragen festgestellt werden soll, ob bereits eine passende Antwort im System existiert. Der Schlüssel zur Bewältigung diese Aufgabe liegt

[1] Scio-Matching Tool Engineering

darin, die Daten entsprechend zu strukturieren um semantische Technologien optimal nutzen zu können. Der für diese Arbeit wichtigste Aspekt dieses Zieles ist die Vorgabe, ähnliche Fragen finden zu können. Eine neue Frage kann daher unter Umständen mit einer bereits vorhandenen Fachbotschaft beantwortet werden, die entweder über die Ähnlichkeit mit einer bereits beantworteten Frage oder durch einen guten semantischen Match mit einer existierenden Antwort gefunden werden kann.

3.3 Abgrenzung zu existierenden Produkten

Das Produkt, welches in diesem Projekt entstehen soll, hat einerseits Eigenschaften einer klassischen Frage-Antwort-Plattform, andererseits sollen auch Elemente eines Recommender Systems zu finden sein. Es gibt Seiten, die sich darauf spezialisiert haben, dass verschiedene Experten und Expertinnen Fragen textuell beantworten. Die Alternative dazu sind sogenannte „Community Q&A“ Produkte, bei denen Benutzer und Benutzerinnen Fragen anderer Personen beantworten. Beide Formen haben oft das Problem, dass es schwer ist, sich als Benutzer oder Benutzerin ein Bild über die Qualität der Antwort bzw. der dahinter stehenden Experten und Expertinnen zu machen, da eine Antwort in den meisten Fällen immer nur für exakt eine Frage erzeugt wird. Es gibt zwar Ansätze, auf gewissen Webseiten Fragen manuell zusammenzufassen, allerdings existieren keine automatischen Mechanismen, wie sie für das S-Mate Projekt angedacht sind.

Als zweiter Punkt dieses Unterkapitels soll nun ganz kurz auf Recommender Systems eingegangen werden, da hier Technologien zum Einsatz kommen, die auch für das Projekt interessant sind. Portale, die solche Ansätze verwenden, setzen oft einen k-nearest-neighborhood-Ansatz bzw. Collaboratives Filtering ein um Benutzern und Benutzerinnen Informationen anzeigen zu können, die mit ihren Interessen korrelieren. Das geplante Projekt wird vor allem in Bereichen wie zum Beispiel „Fast Track“ von den Aspekten der Recommender Systems profitieren.

3.4 Risiken und Probleme

Es gibt einige zentrale Probleme, die während der Umsetzung auftreten können und eine Lösung verlangen. Die vor dem Projektbeginn befürchteten Risiken waren:

- Gewünschte Qualität des Matchings wird nicht erreicht
- Die Antwortzeit für Matchingergebnisse ist zu hoch
- Die Lösung skaliert inhaltlich nicht
- Automatische Erweiterung für semantische Hintergrundinformationen arbeitet schlechter als benötigt
- Das Konzept der „Fast Tracks“ funktioniert nicht wie gewünscht.

Da das Problem, welches sich aus dem Risiko „Die Antwortzeit für Matchingergebnisse ist zu hoch“ ergeben hat, zu dieser Diplomarbeit geführt hat, ist es wichtig kurz darauf einzugehen. Insbesondere geht es hier um das „Finden ähnlicher Fragen“, das während der Projektrealisierung zu erheblichen Problemen geführt hat.

3.5 Übersicht

Um die Vorstellung des Projekts, das zur Diplomarbeit angeregt hat, zu beenden soll in diesem Teil das Konzept für die technische Umsetzung grob vorgestellt werden. Eine Übersicht aller beteiligten Komponenten ist in Abbildung 3.1 zu sehen. Die mittlere Schicht ist für die Erstellung, Verarbeitung und Verwaltung der Medien zuständig und enthält mit der Core Engine das zentrale Element für alle Matching Funktionalitäten.

In der ganz oben liegenden Anwendungsschicht sind Komponenten angesiedelt, die für die Interaktion mit Benutzern und Benutzerinnen sowie Experten und Expertinnen zuständig sind. Hier ist auch die Visualisierung der Suche, die Präsentation der Antworten und Empfehlungen und die Kommunikation zwischen Benutzern und Benutzerinnen angesiedelt.

Zu guter Letzt sorgt die Infrastruktur-Schicht für die Speicherung und Verwaltung aller Informationen, während das Domain Model eine semantische Sicht auf die Daten bietet. Natürlich sind hier auch Elemente angesiedelt die sich um den Anmeldevorgang von Benutzern und Benutzerinnen kümmern.

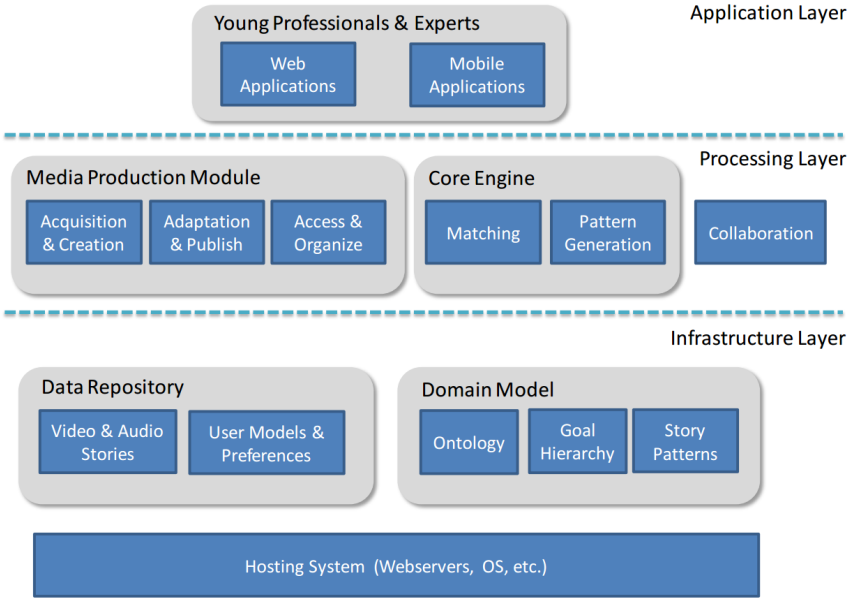


Abbildung 3.1: S-Mate Übersicht

Um eine genauere Sicht auf die für diese Diplomarbeit zentrale Komponente zu bekommen wird nun die Core Engine genauer betrachtet. Abbildung 3.2 zeigt, dass Experten, Expertinnen, Benutzer, Benutzerinnen (Young Professionals), Antworten und Fragen in einem Topic Space gemappt werden. Dieser zentrale Wissensraum besteht aus einer Menge von Ontologien, welche die Konzepte und Beziehungen zwischen diesen enthalten. Für Fragen und Antworten werden noch zusätzliche Metadaten gespeichert um zum Beispiel die Ähnlichkeiten berechnen zu können.

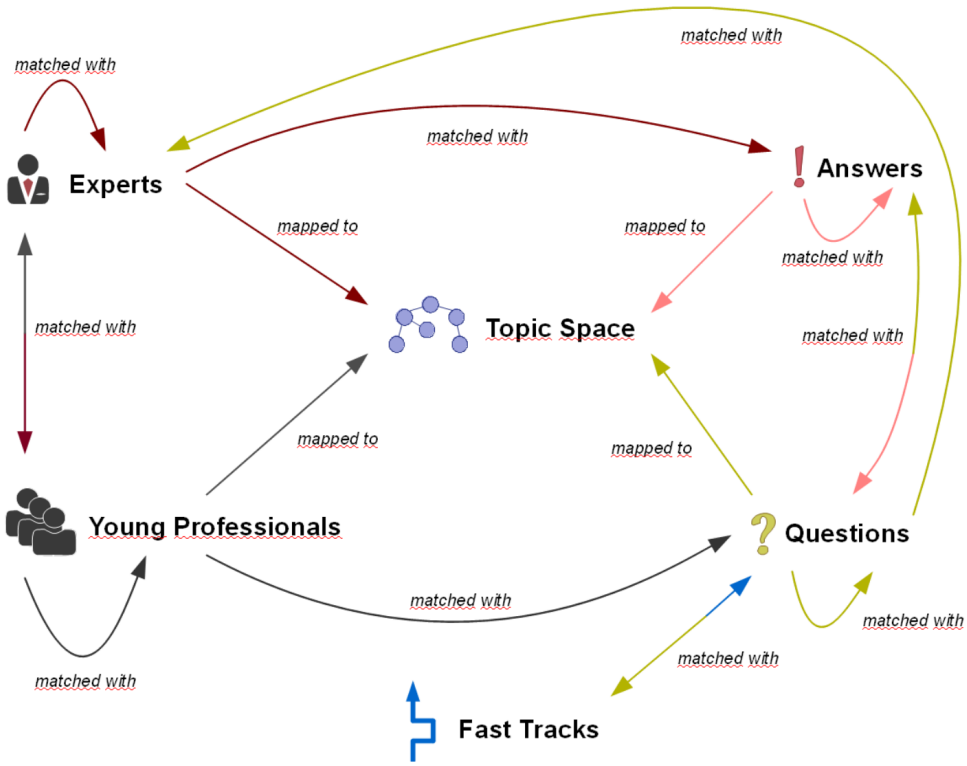


Abbildung 3.2: S-Mate Core Engine

Teile aus diesem S-Mate Projekt, im besonderem Aspekte der „query-recommendation“, wurden bereits in [8, 7, 6, 14] publiziert.

3.5.1 Mapping

Es ist wichtig, dass alle Teile im System eine einheitliche Sprache aufweisen. Dazu werden die Komponenten, die am Matching beteiligt sind, auf den Topic Space abgebildet, was wiederum das Finden von Beziehungen zwischen Elementen erleichtert.

3.5.2 Matching

Da verschiedene Bereiche abgedeckt werden müssen, soll eine Reihe von unterschiedlichen Methoden zum Einsatz kommen. Eine kurze Aufzählung und Beschreibung der einzelnen Ansätze zeigt die Komplexität der zu bewältigenden Aufgaben.

- **Keyword-basiert:** Hierbei handelt es sich um eine Methode, die auf Wort-Ebene vergleicht ob zwei Strings gleich sind. Um das Problem von andere Formen, Schreibweisen und ähnlichem zu umgehen, können Stemmer benutzt werden oder mit Hilfe der Levenshtein-Distanz [9] eine Ähnlichkeit gefunden werden.
- **NLP[2]:** Ein Text kann mit Hilfe von NLP-Frameworks auf sprachlicher und grammatikalischer Ebene analysiert werden. Vertreter dieser Kategorie, die für das Projekt in Frage kommen, sind Part-of-Speech Tagger [16] und Named-Entity-Recognition. Eine Idee, wie solche Ansätze für S-Mate verwendet werden können findet sich in [17] und [10].
- **Häufigkeits-basiert:** Bei den Ansätzen in diesem Bereich geht es hauptsächlich um statistische Analysen der Daten. Es wird beispielsweise analysiert, wie wahrscheinlich es ist, dass bestimmte Wörter gemeinsam in einer gewissen Textpassage vorkommen. In dieser Kategorie finden sich in [2] und [13] Denkanstöße für die Verwendung in diesem Projekt.
- **Relationen-basiert:** Diese Methoden nutzen die Abstände von Elementen im Topic Space um ein grobes Matching zu erreichen.

Das S-Mate Projekt hat vor, für ein Gebiet nicht nur eine Methode einzusetzen, sondern eine Kombination aus verschiedenen Ansätzen zu verwenden. Die in Abbildung 3.2 gezeigten Matching Möglichkeiten sollen der Vollständigkeit halber kurz aufgezählt werden.

- **Frage-Frage:** Diese Analyse ist sehr wichtig, da dem System viel Arbeit abgenommen werden kann, wenn ähnliche Fragen erkannt werden können. Die Analyse basiert in diesem Fall auf den Schlüsselwörtern, NLP, Häufigkeiten und Relationen. Welche Komponenten in welchem Ausmaß zum Einsatz kommen hängt allerdings vom verwendeten Matching Algorithmus ab.
- **Frage-Experte:** Es soll im System vermieden werden, dass Experten und Expertinnen nur Fragen zur Beantwortung vorgelegt bekommen, die dem eigenen Spezialgebiet entsprechen.
- **Frage-Antwort:** Wenn für eine Frage keine ähnliche Frage gefunden werden, kann es aber dennoch sein, dass im System bereits die Antwort bereitsteht.

[2] Natural Language Processing

- **Antwort-Antwort:** Um dem User verschiedene Antworten präsentieren zu können müssen auch zwischen diesen Ähnlichkeiten analysiert werden.
- **Experte-Benutzer:** Damit man dem Benutzer und der Benutzerin die für sein/ihr Fachgebiet passende Antwort präsentieren kann ist es von Vorteil den am besten passenden Experten oder die Expertin zuordnen zu können.
- **Experte-Experte:** Ein Matching zwischen den Experten und Expertinnen wird ebenfalls für die optimale Zuordnung von Fragen genutzt, da es nötig sein kann, dass eine Frage aus zwei verschiedenen Sichtweisen beantwortet wird.
- **Benutzer-Benutzer:** Will man den Benutzern und Benutzerinnen die Möglichkeit bieten, sich mit Gleichgesinnten auszutauschen, ist es notwendig die Ähnlichkeiten innerhalb dieser zu verarbeiten.

KAPITEL 4

Similar Queries in SCIO

4.1 Einleitung

Dieses Kapitel behandelt den Hauptteil der Diplomarbeit, nämlich die schrittweise Umsetzung einer Lösung für die Problemstellung.

Abbildung 4.1 zeigt einen groben Überblick des Ansatzes, der im Laufe des Kapitels Schritt für Schritt als Lösungsidee erarbeitet wird. Da die vorhandenen Matcher unterschiedliche algorithmische Komplexität haben, ist die einzige Chance, die notwendige Performance zu erreichen, in irgendeiner Art und Weise eine Vorberechnung durchzuführen. Die Ergebnisse der Vorausberechnungen werden in Indexstrukturen abgelegt.

Im ersten Teil (Abschnitt 4.2) wird zuerst die Vorberechnung beschrieben, in der die Daten für die spätere Suche aufbereitet werden. Zum Aufbau des Indexes werden hier Matchingalgorithmen des S-Mate Projekts benutzt um ähnliche Fragen zu finden. Der zweite Teil des Kapitels (Abschnitt 4.3) beschäftigt sich mit der Erarbeitung einer Lösung für die dazu gehörige Suche in dieser Datenstruktur.

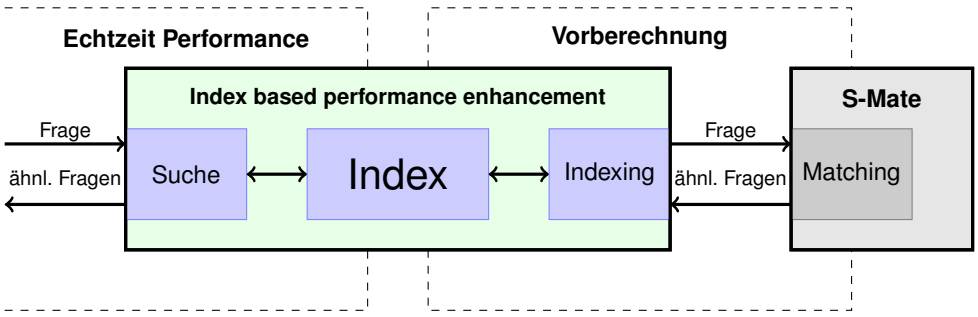


Abbildung 4.1: Lösungsidee Übersicht

4.2 Indexing

Das Umsetzen des Lösungsansatzes, der in Abschnitt 4.1 vorgestellt wird, startet mit der zentralen Klasse `InbaSep`. Das Singleton dieser Klasse erstellt beim Erzeugen für jeden Channel eine Instanz der Klasse `ChannelAC` und legt diese für die spätere effiziente Verwendung in einer Map ab. Im Sequenzdiagramm in Abbildung 4.2 ist der grobe Ablauf des Index-Prozesses mit allen beteiligenden Komponenten zu sehen. Der Vorgang beginnt mit dem Aufruf von `createIndex` in `InbaSep`. Mit Hilfe der übergebenen `channelId`, die zu einem bei der Initialisierung geladenen Channel gehören muss, wird das richtige `ChannelAC` Objekt geladen. Beim Aufruf von `crateIndex` in `InbaSep` wird die `buildIndex` Methode des entsprechenden Channels aufgerufen.

Jeder Channel hat einen eigenen `DataSourceManager` mit dessen Hilfe sich eine neue Datenquelle für den Index-Prozess erzeugen lässt. Je nach Art und Implementierung der von einem Channel verwendeten Datenquelle können gleichzeitige Zugriffe zu Inkonsistenzen und anderen Problemen führen. Da davon ausgegangen wird, dass nicht dauernd ein neuer Index aufgebaut wird, besteht keine auch Notwendigkeit es dem System zu ermöglichen, mehrere Indexing Prozesse auf demselben Channel auszuführen. Deshalb wird der gesamte Prozess mit Hilfe von Java-Locks abgesichert, womit garantiert ist, dass maximal 1 Indexierung pro Channel läuft. Wie Listing 4.1 zeigt wird der Lock erst wieder freigegeben, wenn alle Sub-Indexer durchgelaufen sind und die neue Datenquelle aktiv gesetzt wurde.

Listing 4.1: Java-Locks im Index-Prozess

```
1  if (indexingLock.tryLock()) {
2      try {
3          IDatasource datasource = inbasepDatasourceManager.newVersion();
4
5          List<IIndexer> indexer = new LinkedList<IIndexer>();
6          indexer.add(new QueryIndexer());
7          indexer.add(new KeywordWordformIndexer());
8          indexer.add(new SemanticIndexer());
9
10         try {
11             for (IIndexer idx : indexer) {
12                 idx.index(datasource);
13             }
14             datasource.setActive();
15         }
16         catch (DaoCloseException e) {...}
17         catch (DaoOpenException e) {...}
18     } finally {
19         indexingLock.unlock();
20     }
21
22 } else {
23     ...
24 }
```

Die Sub-Indexer `QueryIndexer`, `KeywordWordformIndexer` und `SemanticIndexer` werden erzeugt und in eine Liste eingefügt. Der Grund warum alle Sub-Indexer dasselbe In-

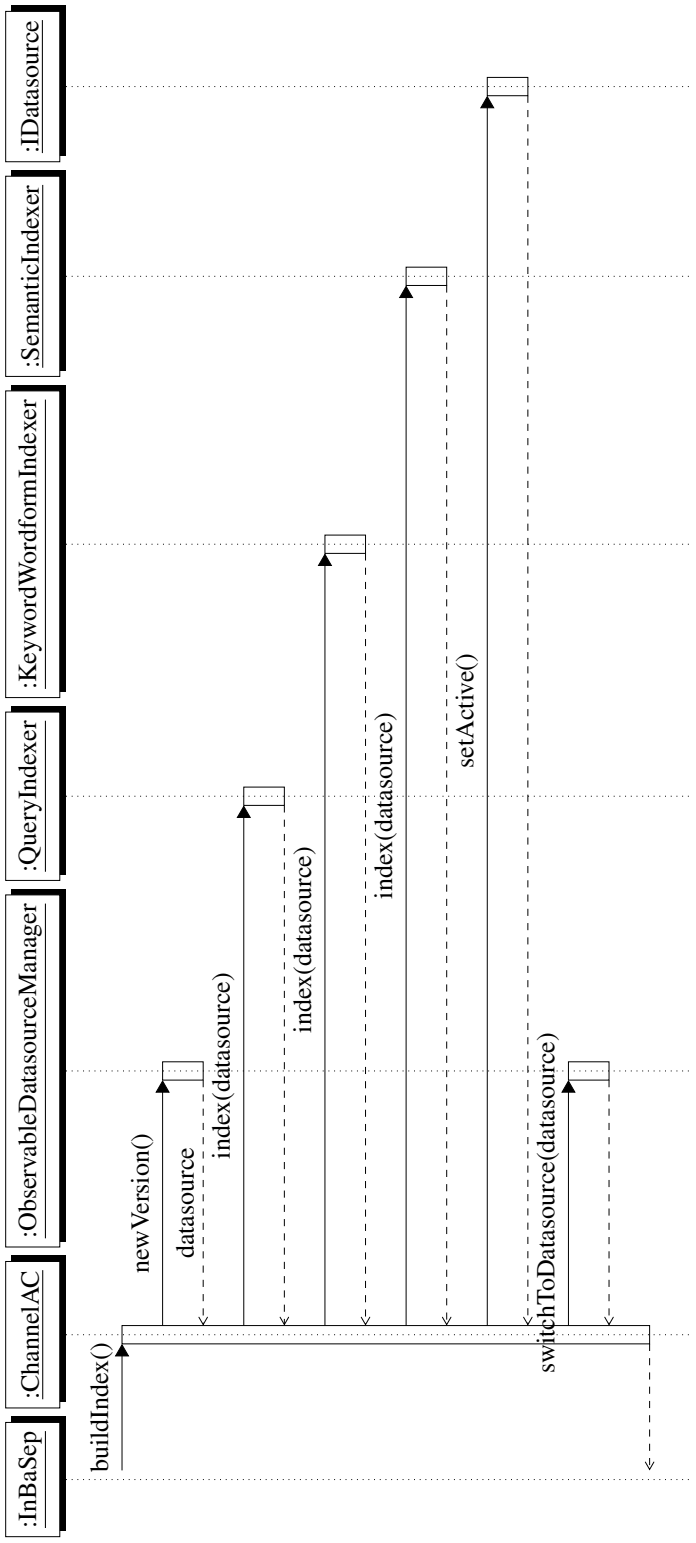


Abbildung 4.2: Sequenzdiagramm Indexgenerierung

terface implementieren und unabhängig voneinander sind, ist vom Gedanken geprägt den Index-Prozess später leicht erweitern zu können. Anschließend wird die `index` Methode bei all diesen Sub-Indexern gestartet. Letztere haben untereinander keine Abhängigkeiten und könnten daher gleichzeitig gestartet und parallel abgearbeitet werden. Dieser Ansatz ist nicht zielführend, da der semantische Indexer in der Regel wesentlich länger zur Komplettierung benötigt als die beiden anderen. Außerdem wären QuadCore Prozessoren, auf denen problemlos 8 Threads gleichzeitig abgearbeitet werden, mit den drei vorhandenen Sub-Indexern nicht optimal ausgelastet. Daher werden die einzelnen Indexer sequenziell abgearbeitet und die Parallelisierung ist innerhalb dieser implementiert, sofern dadurch ein gewisser Speedup erzielt werden kann.

Im Folgenden werden die drei Sub-Indexer im Detail beschrieben. Der Zugriff auf Daten und Algorithmen der Core-Engine erfolgt über REST um die Search Engine komplett entkoppeln zu können.

4.2.1 Query Indexing

Es gibt zwei Gründe für den Aufbau des Indexes über alle Fragen. Der erste Grund ist die Entkoppelung der SearchEngine von Core-Engine. Die Suche soll später ohne Zugriffe auf die Datenbank der Core-Engine erfolgen. Der zweite Grund ist die Verwendung des Indexes für die syntaktische Suche nach Fragen. Zu diesem Zweck werden alle Fragen, die zu dem entsprechenden Channel gehören, für die spätere Suche vorverarbeitet und in der InBaSep Datenquelle abgelegt.

Abbildung 4.3 zeigt grob den Ablauf der Erstellung des Indexes der später unter anderem für die semantische Suche verwendet wird und geht auf die einzelnen Komponenten ein. Um an die benötigten Daten zu kommen, muss der `QueryController` auf Grund der Entkoppelung über REST auf die Core-Engine zugreifen. Der Aufruf von `findAllByChannel` bekommt alle Fragen, die dem entsprechendem Channel zugeordnet sind, und gibt diese an den Query Indexer weiter. Um Daten in die `LuceneQueryDao` schreiben zu können muss dieser zuerst mit Hilfe von `startCreate` fürs Schreiben initialisiert werden. Im Fall dieser Lucene Implementierung wird für die Erstellung der Index-Struktur ein `IndexWriter` generiert und geöffnet. Mit Hilfe eines Wrappers kann das Frage-Objekt in ein gültiges Lucene Document umgewandelt und anschließend in den neuen Index eingefügt werden. Da der `IndexWriter` wieder geschlossen werden muss um einen validen Lucene Index zu erhalten, ist der letzte Schritt des Query Indexing der Methodenaufruf `endCreate`, der die Datenstruktur schließt und, wenn die Option aktiviert ist, Optimierungen vornimmt.

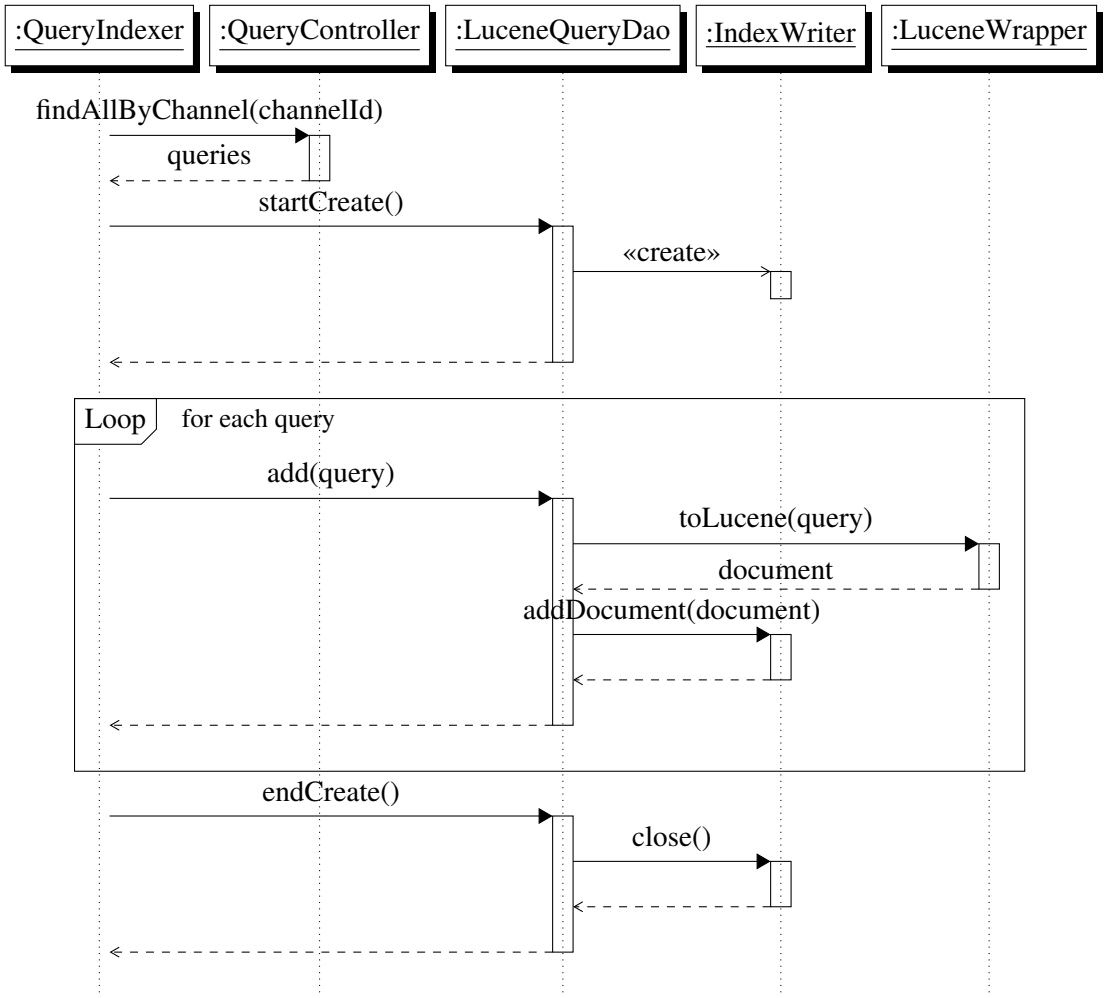


Abbildung 4.3: Sequenzdiagramm Indexgenerierung Queries

4.2.2 Keyword/Word Indexing

Für die spätere semantische Suche werden Schlüsselwörter verwendet, daher müssen diese erkannt werden können. Dafür gibt es verschiedene Matching Algorithmen in der Core-Engine (siehe Kapitel 3), die aber wie bereits erwähnt keine akzeptable Real-Time Performance bieten. Um bei der späteren Suche Schlüsselwörter in verschiedenen Wortformen erkennen zu können, werden diese ebenfalls in einer Indexstruktur abgelegt. Das semantische Matching kennt auch zusammengesetzte Keywords, die sich wiederum aus Schlüsselwörtern und Nicht-Schlüsselwörtern zusammensetzen können. Daher werden alle der Core-Engine bekannten Wörter abgelegt und entsprechend gekennzeichnet. Um im späteren Verlauf Schlüsselwörter von Nicht-Schlüsselwörtern unterscheiden zu können werden letztere fortan als Words bzw. Wörter bezeichnet.

Die dritte Gruppe an Worten die in diesem Index gespeichert werden sind Synonyme. Diese sind notwendig, da verschiedene Personen ein und die selbe Frage unterschiedlich formulieren und dabei auch oft gleichbedeutende Ausdrücke verwenden. Diese Ersatzworte werden wie Schlüsselwörter abgelegt und zusätzlich mit einem Feld im Index gekennzeichnet.

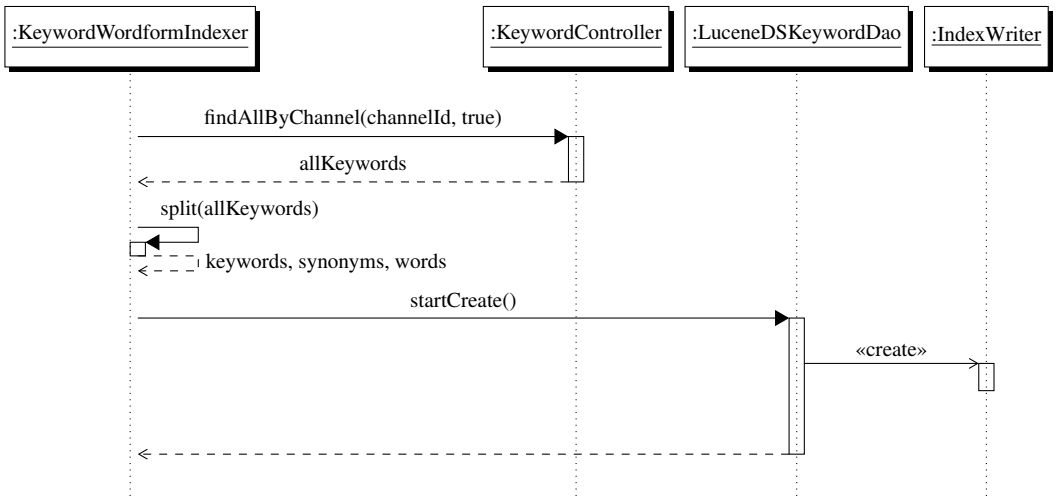


Abbildung 4.4: Sequenzdiagramm Indexgenerierung Queries Teil 1

Es gibt zwei Gründe warum bekannte Wörter, Schlüsselwörter und deren Synonyme in einer InBaSep Datenquelle abgelegt werden. Der erste ist die spätere Unabhängigkeit von der Core-Engine. Da man bei der Suche Wörter auf ihre Grundform reduzieren möchte, benötigt man eine Möglichkeit diese Aufgabe effizient zu erledigen, was auch gleichzeitig den zweiten Grund darstellt. Das Indexieren der Wörter, Schlüsselwörter und Synonyme selbst ist nur geringfügig aufwändiger als das der Fragen.

Wie man in Abbildung 4.4 und Abbildung 4.5 erkennen kann entsteht der Mehraufwand dadurch, dass vom `KeywordController` eine Menge an `Keyword` Objekten zurückkommen, die noch verarbeitet werden müssen. Diese Menge kann in Schlüsselwörter und Wörter aufgeteilt werden, wobei alle Teile eines zusammengesetzten Keywords in der Menge der Wörter landen. Aus den Synonymen, die wiederum in `Keyword` Objekten abgelegt sind, entsteht

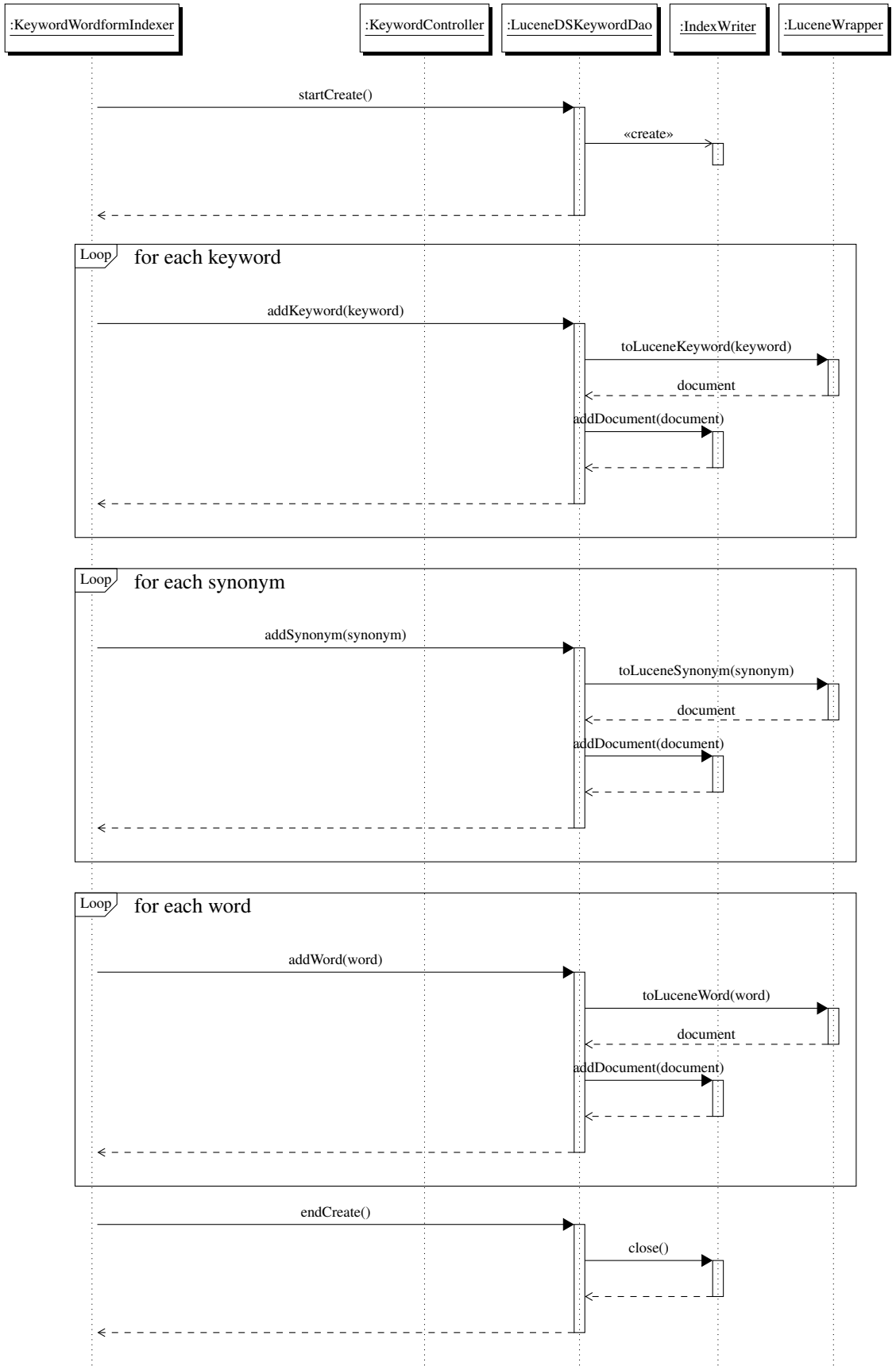


Abbildung 4.5: Sequenzdiagramm Indexgenerierung Queries Teil 2

die Menge der Synonyme. Um doppelte Einträge zu vermeiden, wird, wie im Code ersichtlich (siehe Listing 4.2), zusätzlich noch die Menge der Synonyme und Keywords von den Words subtrahiert. Das Starten des Schreibvorgangs auf die Datenquelle erfolgt analog zum Query Indexing. Anschließend wird jedes Schlüsselwort, Synonym und Wort in ein Lucene Dokument umgewandelt und in den Index eingefügt. Wie auch schon beim Query Indexing wird zum Ende noch der `IndexWriter` geschlossen um den Prozess zu beenden.

Listing 4.2: Aufteilen in Keywords Words und Synonyms

```
1  for (Keyword w : allWords) {
2      if (w.isShowWord()) {
3          if (w.isCompound()) {
4              Set<Keyword> kwParts = w.getKeywordParts();
5              words.addAll(kwParts);
6          }
7          keywords.add(w);
8      } else {
9          words.add(w);
10     }
11     if (w.hasSynsets()) {
12         synonyms.addAll(IndexingTools.getSysnonyms(w));
13     }
14 }
15 words.removeAll(synonyms);
16 words.removeAll(keywords);
```

In Abbildung 4.6 sieht man wie die Schlüsselwörter im Index abgebildet werden. Alle gefundenen Wortformen sind sowohl im Original als auch in einer normalisierten Variante abgelegt, die keine Umlaute enthält. Relativ ähnlich sieht das Word aus, wie man in Abbildung 4.7 erkennen kann. Wie zuvor bereits erwähnt, kann ein Keyword auch aus mehreren Wörtern bestehen. Abbildung 4.8 zeigt, dass bei einem zusammengesetzten Schlüsselwort neben einer Version ohne Leerzeichen (original und normalisiert) auch alle einzelnen Teil-Worte gespeichert werden. An Abbildung 4.9 erkennt man, dass sich Synonyme im Index nur durch das vorhin erwähnte Feld `synonym` von einem Schlüsselwort unterscheidet.

Doc #: 11 **Flags:** I - Indexed (docs,freqs,pos) T - Tokenized; S - Stored; V - Term Vector (offsets,pos) N - with Norms; L - Lazy; B - Binary;

Field	IdfpTSVopNLB	Norm	Value
form	IdfpTS---N--	0.4375	sättigungen
form	IdfpTS---N--	0.4375	sättigunges
form	IdfpTS---N--	0.4375	sättigunge
form	IdfpTS---N--	0.4375	sättigungem
form	IdfpTS---N--	0.4375	sättigunger
form_simple	IdfpTS---N--	0.4375	saettigungen
form_simple	IdfpTS---N--	0.4375	saettigunges
form_simple	IdfpTS---N--	0.4375	saettigunge
form_simple	IdfpTS---N--	0.4375	saettigungem
form_simple	IdfpTS---N--	0.4375	saettigunger
keyword	IdfpTS---N--	1.0	sättigung
word	IdfpTS---N--	1.0	sättigung
word_simple	IdfpTS---N--	1.0	saettigung

Abbildung 4.6: Darstellung eines Keyword im Index

Doc #: 3104 **Flags:** I - Indexed (docs,freqs,pos) T - Tokenized; S - Stored; V - Term Vector (offsets,pos) N - with Norms; L - Lazy; B - Binary;

Field	IdfpTSVopNLB	Norm	Value
word	IdfpTS---N--	1.0	erleben
word_simple	IdfpTS---N--	1.0	erleben

Abbildung 4.7: Darstellung eines Nicht-Keyword (Word) im Index

Doc #: 5 **Flags:** I - Indexed (docs,freqs,pos) T - Tokenized; S - Stored; V - Term Vector (offsets,pos) N - with Norms; L - Lazy; B - Binary;

Field	IdfpTSVopNLB	Norm	Value
compound	IdfpTS---N--	1.0	T
cpart	Idfp-S---N--	0.625	esse
cpart	Idfp-S---N--	0.625	milchprodukt
keyword	IdfpTS---N--	1.0	milchprodukt esse
keywordnows	IdfpTS---N--	1.0	milchproduktesse
keywordnows_simple	IdfpTS---N--	1.0	milchproduktesse

Abbildung 4.8: Darstellung eines zusammengesetzten Keyword im Index

Doc #: 401 Flags: I - Indexed (docs,freqs,pos) T - Tokenized; S - Stored; V - Term Vector (offsets,pos) N - with Norms; L - Lazy; B - Binary;

Field	IdfpTSVopNLB	Norm	Value
form	IdfpTS---N--	1.0	saltos
form_simple	IdfpTS---N--	1.0	saltos
keyword	IdfpTS---N--	1.0	salto
synonym	IdfpTS---N--	1.0	T
word	IdfpTS---N--	1.0	salto
word_simple	IdfpTS---N--	1.0	salto

Abbildung 4.9: Darstellung eines Synonyms im Index

4.2.3 Semantic Indexing

Da man dem Benutzer oder der Benutzerin eine möglichst natürliche Bedienung bieten will, erwartet man, dass eine vollständige Frage als Suchstring eingeben wird. Aus dieser Annahme ergibt sich eine extrem große Menge an möglichen Eingaben, die unmöglich vorberechnet werden kann. Daher werden für die semantische Suche alle Wörter außer acht gelassen, die dem System nicht bekannt sind. Da es außerdem für den Matching-Algorithmus keine Rolle spielt in welcher Reihenfolge die Schlüsselwörter in der Eingabe zu finden sind, reduziert sich die Menge der möglichen Eingaben weiter. Auf Grund dieser Annahmen kann die Menge aller theoretisch möglichen Eingaben auf Kombinationen von Schlüsselwörtern und Synonymen in allen Wortformen abgebildet werden. Da festgelegt wurde, dass die Wortform nicht entscheidend, ist müssen die Komponenten dieser Kompositionen nur in ihrer Grundform gespeichert werden.

Auf semantischer Ebene dienen Schlüsselwörter zur Findung von ähnlichen Fragen. Aus diesen dem System bekannten Keywords werden beliebig lange Kombinationen gebildet und die Ähnlichkeiten mit gespeicherten Fragen berechnet. Um diese Aufgabe zu bewältigen bedient sich die Core-Engine verschiedener Matching-Algorithmen (siehe Kapitel 3). Lösungen die zufriedenstellende Ergebnisse liefern weisen eine Laufzeit auf, die für Echtzeitanwendungen nicht akzeptabel ist. Da es keine Möglichkeit gibt die Berechnungen zu beschleunigen ohne die Hardware zu erweitern, liegt die Idee nahe, die Matchings vorzuberechnen.

Brute-Force

Um die semantischen Matches vorzubereiten müssen die Schlüsselwörter kombiniert werden. Eine solche Kombination muss aus mindestens einem Keyword bestehen und maximal aus allen. Wenn man ohne Selektion einfach alle theoretisch möglichen Kombinationen berechnet will, dann ist das nur bei einer kleinen Menge an Keywords möglich. Anhand der Tabelle 4.1, in der man die Anzahl der möglichen Kombinationen bis zu einer Länge 4 ablesen kann, zeigt sich bereits, wie schnell die Mengen anwachsen. Geht man von einem Matching-Algorithmus aus, der im Durchschnitt zwei Sekunden benötigt hat, können die Kombinationen der Länge 3 bei 100 Elementen schon nicht mehr in einer sinnvollen Zeit berechnet werden (siehe Tabelle 4.2).

In Realität werden aber mehrere tausend Schlüsselwörter zu erwarten sein, was die Methode, alle möglichen Kombinationen zu untersuchen, unmöglich macht.

Tabelle 4.1: Brute-Force: Anzahl der Kombinationen

Keywortmenge	2 Keyword	3 Keywords	4 Keyword
10	45	120	210
100	4950	161700	3921225
1000	499500	$\approx 1.6 * 10^8$	$\approx 4.1 * 10^{10}$
2000	$2 * 10^6$	$\approx 1.3 * 10^9$	$\approx 6.6 * 10^{11}$

Tabelle 4.2: Brute-Force: Berechnungsdauer der Matches (2 Sekunden pro Match)

Keywortmenge	2 Keyword	3 Keywords	4 Keyword
10	1, 5 Minuten	4 Minuten	7 Minuten
100	2, 75 Stunden	$\approx$ 90 Stunden	90 Tage
1000	$\approx 11, 6$ Tage	$\approx 10, 5$ Jahre	$\approx$ 2627 Jahre
2000	$\approx 46, 3$ Tage	$\approx 84, 4$ Jahre	$\approx$ 42153 Jahre

Ideen für Optimierungen

Man kann davon ausgehen, dass nur Schlüsselwort-Kombinationen Sinn machen, wenn sie auf ähnliche Fragemengen matchen. Aus dieser Annahme geht die erste Verbesserung hervor, die auf verschiedene Weisen eingesetzt werden kann. Die einfachste Möglichkeit ist es, Keywords nur dann zu kombinieren und in Folge daraus Matchings zu berechnen, wenn die Schnittmenge der einzelnen Fragen-Matches nicht leer ist. In Tabelle 4.3 kann man erkennen, dass von drei Möglichkeiten, die man als Ergebnis der Brute-Force Variante erhält, nur eine sinnvolle Kombination bleibt. Das „Pre-Matching“ wird in der Darstellung auch vom eigentlichen Matching unterschieden, da es sich bei ersterem nur um einen Filter-Mechanismus handelt, der die Problemgröße reduzieren soll. Hat man diese sinnvollen Kombinationen gefunden, kann die eigentliche Berechnung stattfinden. Auch wenn diese Strategie im Prototypen umgesetzt wird, soll noch kurz auf weitere Möglichkeiten eingegangen werden.

Wenn diese eben erwähnte Einschränkung nicht ausreicht, gibt es die Möglichkeit mehr als eine nicht leere Schnittmenge zu verlangen. Es könnte also Bedingung sein, dass die Schnittmenge eine gewisse Mächtigkeit hat. Dieser Wert könnte absolut bzw. prozentual festgelegt werden und je nach Kombinationslänge angepasst werden. Wenn man Tabelle 4.4 ohne weitere Beschränkungen betrachtet, wären alle drei Kombinationen sinnvoll. Setzt man beispielsweise aber voraus, dass die Mächtigkeit der Schnittmenge mindestens 30% der Mächtigkeit der Vereinigungsmenge sein muss, wäre nur „K1, K2“ ein Kandidat für die weitere Berechnung.

Welche der beiden Möglichkeiten im Endeffekt dann benötigt wird, hängt davon ab wie sehr der Matching Algorithmus konfigurierbar ist. Denn wenn für jede mögliche passende Frage ein

Tabelle 4.3: Sinnvolle Kombinationen (Möglichkeit 1)

Keywords	Pre-Matching	Matching
K1		$\{F1\}$
K2		$\{F2, F3\}$
K3		$\{F3\}$
K1, K2	$\emptyset$	–
K1, K3	$\emptyset$	–
K2, K3	$\{F3\}$	$\{F3\}$

Tabelle 4.4: Sinnvolle Kombinationen (Möglichkeit 2)

Keywords	Pre-Matching	Matching
K1		$\{F1, F2, F3\}$
K2		$\{F1, F3, F4\}$
K3		$\{F1, F5, F6\}$
K1, K2	$\{F1, F3\}$ aus $\{F1, F2, F3, F4\}$	$\{F1, F3\}$
K1, K3	$\{F1\}$ aus $\{F1, F2, F3, F5, F6\}$	–
K2, K3	$\{F1\}$ aus $\{F1, F3, F4, F5, F6\}$	–

Ähnlichkeitsmaß mitgeliefert wird, kann auch über einen Grenzwert die Menge der passenden Fragen entsprechend eingeschränkt werden. Sollte dies nicht der Fall sein, muss unter Umständen die zweite Variante gewählt werden.

Beide Möglichkeiten wurden auf generische Testdaten angewandt und die Ergebnisse mit der maximalen Anzahl an Kombinationen verglichen. Diese 10, 20 bzw. 30 Keywords zeigen jeweils auf 0 bis 10 verschiedene Fragen. Im Test (siehe Tabelle 4.5) mit einem sehr kleinen Fragenpool konnte die Anzahl der möglichen Kombinationen, die aus 3 Schlüsselwörtern bestehen, durch die ersten Variante auf bis zu 6% der Originalgröße reduziert werden. Verlangt man allerdings, dass die Mächtigkeiten der Schnitt- und Vereinigungsmenge ein gewisses Verhältnis zueinander haben, so wird in diesem Beispiel eine Verkleinerung auf 4% möglich.

Tabelle 4.5: Reduktion der Keyword-Kombinationen - Test

Keywortmenge	2 Keywords			3 Keywords		
	o. Einschr.	Variante 1	Variante 2	o. Einschr.	Variante 1	Variante 2
10	45	20 (44%)	9 (20%)	120	24 (20%)	6 (5%)
20	190	63 (33%)	29 (15%)	1140	178 (16%)	114 (10%)
30	435	148 (34%)	78 (18%)	4060	253 (6%)	150 (4%)

Der Nachteil dieser Optimierungen ist, dass die Qualität der Einschränkungen stark davon abhängt, dass die richtigen Parameter für den jeweiligen Matching-Algorithmus gewählt werden.

Aus Hardware-Gründen rentiert sich als weitere Verbesserung natürlich eine Parallelisierung, da man mit einer heute weit verbreiteten Hardware mit 4-Kern CPUs[1] vier Berechnungen gleichzeitig durchführen kann. Wenn eine Cloud zur Verfügung steht, können vielleicht hunderte Matchings zur selben Zeit gestartet werden. Natürlich würde das bei der Brute-Force Variante nur ein Tropfen auf den heißen Stein sein, mit den eben erwähnten Optimierungsmöglichkeiten bringt es aber nochmal eine sinnvolle Beschleunigung.

Synonyme

Ersetzt man ein Schlüsselwort durch ein Synonym so hat dies keinen Einfluss auf die Ergebnisse des Matchings. Für jedes Keyword gibt es eine Menge dieser bedeutungsgleichen Worte. Da es keine Möglichkeit gibt von einem Element dieser Menge wieder zurück zum Schlüsselwort zu kommen, reicht es hier nicht aus eine Kombination nur mit Schlüsselwörtern zu speichern. Es werden daher nach einem erfolgreichen Matching durch Substitution alle möglichen Kombinationen gebildet und gespeichert. Auch wenn in Tabelle 4.6 zur einfacheren Berechnung gleich viele Synonyme bei jedem Keyword angenommen wurden, kann man gut erkennen mit wie vielen Variationen in etwa zu rechnen ist. Die Berechnungszeit steigt dadurch nicht, da die Bildung der Kombinationen kaum Zeit in Anspruch nimmt. Natürlich nimmt die Anzahl der Einträge dadurch zu, weshalb mit wesentlich mehr Speicherverbrauch gerechnet werden muss.

Tabelle 4.6: Synonyme: Anzahl der Kombinationen

Synonyme	2 Keywords	3 Keywords	4 Keywords
1	4	8	16
2	9	27	81
3	16	64	256
4	25	125	625
5	36	216	1296
10	121	1331	14641
20	441	9261	194481

Implementierung

Es gibt 2 verschiedene Möglichkeiten alle Kombinationen in Lucene zu speichern, die beide Vor- und Nachteile haben. Entweder es wird eine Indexstruktur angelegt in der alles gespeichert wird, oder es findet eine Unterteilung statt, die sich auf die Anzahl der Schlüsselwörter in einer Kombination bezieht. Auch wenn die erste Methode mehr Möglichkeiten zur Optimierung des Indexes bietet, ist die Suche mit der zweiten Methode wesentlich effizienter (siehe Abschnitt 4.3).

[1] z. B. Intel Core i7

Abbildung 4.10: Sequenzdiagramm Indexgenerierung Queries

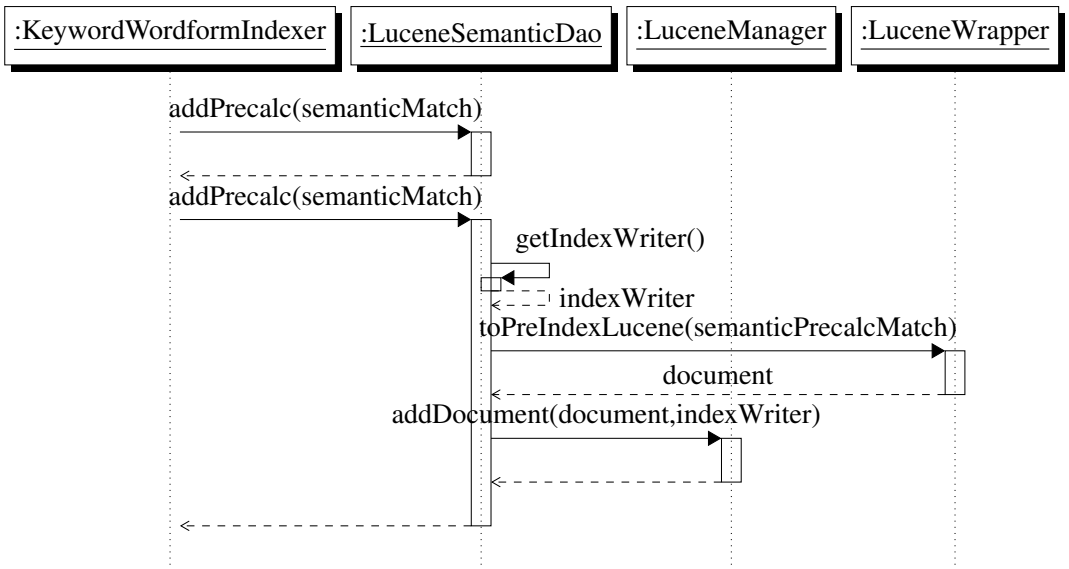


Abbildung 4.11: Sequenzdiagramm addPrecalc

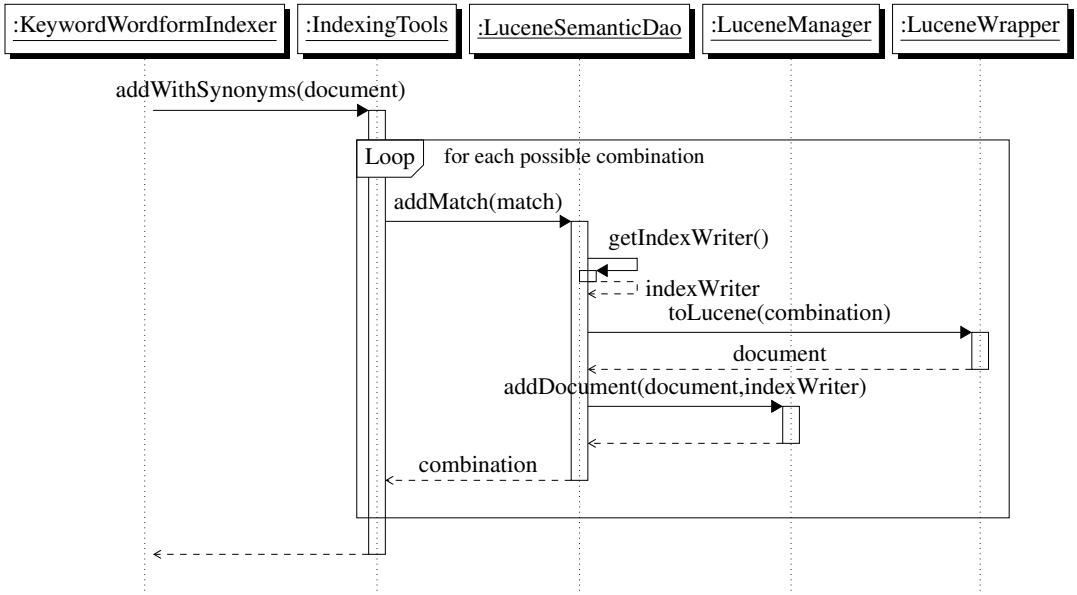


Abbildung 4.12: Sequenzdiagramm addWithSynonyms

Da der Semantische Index aus verschieden langen Kombinationen von Keywords besteht, ist die Berechnung der Daten etwas komplexer, da man vorberechnete temporäre Daten benötigt um die eigentlichen Matches kalkulieren zu können. Beim Bauen des semantischen Indexes soll keine Abhängigkeit zu den anderen beiden Indizes bestehen, um die Möglichkeit einer späteren Parallelisierung der einzelnen Indexer erreichen zu können. Der Ablauf dieses Indexierungsprozesses wird in Abbildung 4.10 dargestellt. Über die Datenzugriffsschicht der Core-Engine werden mit Hilfe des `KeywordDao` die Schlüsselwörter geholt. Für die spätere Verarbeitung werden außerdem noch die inaktiven Queries benötigt, die dann über `QueryDao` abgerufen werden. Im Gegensatz zu den anderen beiden Indexern wird beim Ausführen von `startCreate` kein `IndexWriter` erstellt, da in dieser Implementierung mehrere Indizes verwendet werden.

In der Folge werden zu verschiedenen Kombinationen von Keywords mit Hilfe eines Matchers die passenden Fragen inklusive Bewertung gesucht. Um die spätere Suche zu beschleunigen werden die Kombinationen nach der Länge (die Anzahl der Schlüsselwörter) in verschiedene Indexstrukturen geschrieben. Da die Kombinationen wie zuvor erwähnt durch ein Prematch einschränkt werden, benötigt man für die Länge n die Vorberechnung der Länge $n-1$, was eine sequenzielle Abarbeitung diktiert. Da bei der Länge 1 diese Kalkulationen noch nicht vorliegen, müssen zwei verschiedene Fälle unterschieden werden. Im ersten werden zu den einzelnen Schlüsselwörtern, die als Kombination der Länge 1 gesehen werden können, die passenden Fragen gesucht und diese anschließend als Vorberechnung mit Hilfe von `addPrecalc` (siehe Abbildung 4.11) in einem Hilfsindex abgelegt. Außerdem werden die Matches mit Hilfe von `addWithSynonyms` im späteren Suchindex ablegt. Wie man in Abbildung 4.11 und 4.12 sieht, wird beim Hinzufügen eines Dokuments `getIndexWriter` aufgerufen. Diese Funktion holt den entsprechenden `IndexWriter` aus einem Set und erzeugt ihn, sollte er noch nicht existieren.

Listing 4.3 zeigt den entscheidenden Teil der Funktion `addWithSynonyms`. Es wird zuerst die Stelle n einmal durch-iteriert und anschließend dasselbe noch einmal für das nächste Wort an Position $n-1$. Die aktuelle Kombination ist immer in `tempCombinations` abgelegt, die jedes Mal gleich abgelegt wird. Ein Beispiel für die Kombinationen, die der Reihe nach gebildet werden, ist in Listing 4.4 zu sehen.

Listing 4.3: Code Auszug addWithSynonyms

```
1  List<ResetableIterator<MyKeyword>> synonymIterators = new LinkedList<
       ResetableIterator<MyKeyword>>();
2
3  ...
4
5  int i = 0;
6  MyKeyword[] tempCombinations = new MyKeyword[numKeywords];
7  while(true) {
8      try {
9          MyKeyword k = synonymIterators.get(i).next();
10         tempCombinations[i] = k;
11
12         if (i+1 == numKeywords) {
13             List<String> cParts = new LinkedList<String>();
14             List<String> keywordList = new LinkedList<String>();
```

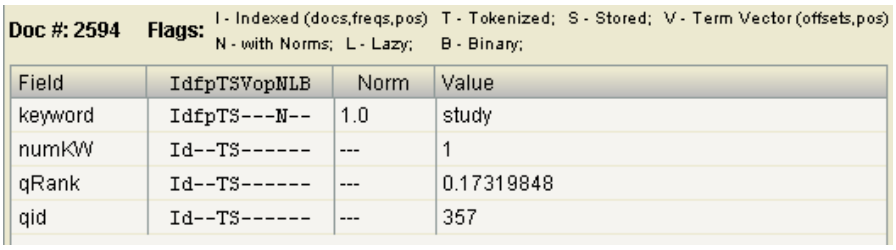
```
15              for (int j = 0; j < numKeywords; j++) {
16                  try {
17                      keywordList.add(tempCombinations[j].getBaseForm());
18                  } catch (NullPointerException e) {}
19
20                  Set<MyKeyword> parts = tempCombinations[j].getParts();
21                  for (Keyword kp : parts) {
22                      cParts.add(kp.getBaseForm());
23                  }
24              }
25              SemanticMatch match = new SemanticMatch(keywordList, cParts,
                    queries);
26              semanticDao.addMatch(match, inactiveQueryMap);
27          }
28
29          if (i+1 < numKeywords) i++;
30      } catch (NoSuchElementException e) {
31          if (i == 0) break;
32          synonymIterators.get(i).reset();
33          i--;
34      }
35
36  }
```

Listing 4.4: Beispiel für addWithSynonyms

```
Keywords: K1, K2, K3
Synonyme:
    K1: S1a
    K2: S2a
    K3: S3a, S3b

Kombinationen:
K1  K2  K3
K1  K2  S3a
K1  K2  S3b
K1  S2a K3
K1  S2a S3a
K1  S2a S3b
S1a K2  K3
S1a K2  S3a
S1a K2  S3b
S1a S2a K3
S1a S2a S3a
S1a S2a S3b
```

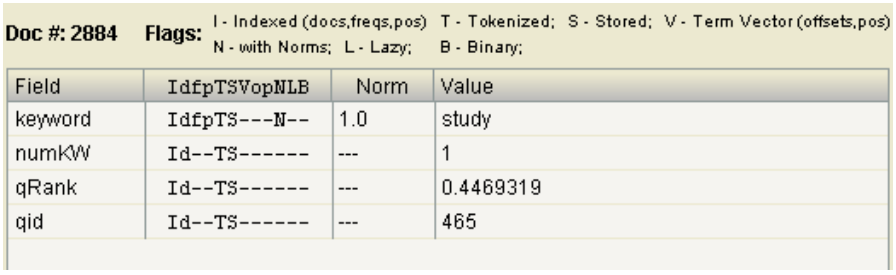
Abschließend muss noch ein Blick darauf geworfen werden wie diese Einträge dann im Lucene Index repräsentiert werden. Abbildung 4.13 und 4.14 zeigen wie Einträge für einzelne Schlüsselwörter aussehen können. Neben dem Keyword ist auch der vom Matcher ermittelte Wert sowie die zugehörige Frage-ID gespeichert. Wie Kombinationen von Schlüsselwörtern mit Hilfe des Lucene Frameworks abgelegt werden ist in Abbildung 4.15 zu sehen. Analog dazu werden Kombinationen beliebiger Länge in den Indices gehalten.



Doc #: 2594 **Flags:** I - Indexed (docs,freqs,pos) T - Tokenized; S - Stored; V - Term Vector (offsets,pos) N - with Norms; L - Lazy; B - Binary;

Field	IdfpTSVopNLB	Norm	Value
keyword	IdfpTS---N--	1.0	study
numKW	Id--TS------	---	1
qRank	Id--TS------	---	0.17319848
qid	Id--TS------	---	357

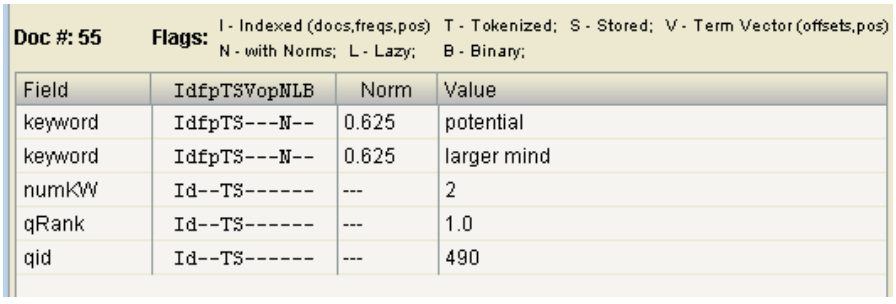
Abbildung 4.13: Eintrag im Lucene Index: Schlüsselwort und Frage-Match 1



Doc #: 2884 **Flags:** I - Indexed (docs,freqs,pos) T - Tokenized; S - Stored; V - Term Vector (offsets,pos) N - with Norms; L - Lazy; B - Binary;

Field	IdfpTSVopNLB	Norm	Value
keyword	IdfpTS---N--	1.0	study
numKW	Id--TS------	---	1
qRank	Id--TS------	---	0.4469319
qid	Id--TS------	---	465

Abbildung 4.14: Eintrag im Lucene Index: Schlüsselwort und Frage-Match 2



Doc #: 55 **Flags:** I - Indexed (docs,freqs,pos) T - Tokenized; S - Stored; V - Term Vector (offsets,pos) N - with Norms; L - Lazy; B - Binary;

Field	IdfpTSVopNLB	Norm	Value
keyword	IdfpTS---N--	0.625	potential
keyword	IdfpTS---N--	0.625	larger mind
numKW	Id--TS------	---	2
qRank	Id--TS------	---	1.0
qid	Id--TS------	---	490

Abbildung 4.15: Eintrag im Lucene Index: Schlüsselwort-Kombination und Frage-Match

4.3 Suche

Die Vorstellung bei der Suche ist es, dass der Benutzer oder die Benutzerin eine Frage im Form eines Satzes eingibt und das System dann laufend ähnliche Fragen vorschlägt. Um eine effiziente Abarbeitung zu ermöglichen, muss die Eingabe des Benutzers oder der Benutzerin in einem ersten Schritt vorverarbeitet werden. Die syntaktische Suche verwendet einen großen Teileingabestring in gestemmter Form, für das semantische Gegenstück wird eine Reduktion auf Schlüsselwörter in ihrer Grundform durchgeführt.

Wie in Abbildung 4.16 zu sehen ist, besteht die Suche aus der Vorverarbeitung der Daten und der eigentlichen Suche. Ersteres beinhaltet Arbeitsschritte, die für die beiden implementierten Suchen von Relevanz sind und wird im anschließenden Unterkapitel kurz erläutert. Da das Hauptziel ist, die Suchzeit zu minimieren, und da keine Abhängigkeit zwischen den semantischen und syntaktischen Komponenten besteht, können diese parallel durchgeführt werden.

4.3.1 Vorverarbeitung

Als Vorbereitung für die Suche werden die Eingabe-Strings erst entsprechend aufbereitet. Listing 4.5 zeigt die Funktion mit deren Hilfe jede neue Anfrage vom System in die fürs Suchen richtige Form gebracht wird. Zuerst wird die Eingabe in einzelne Wörter zerstückelt um anschießend Wort für Wort verarbeitet werden zu können. Für jedes Stück wird ein `QueryItem`-Objekt angelegt, das Platz für alle später benötigten Meta-Informationen bietet. Es wird überprüft, ob es sich um ein vordefiniertes Stoppwort handelt, was mittels boolscher Variable festgehalten wird. Außerdem wird für jedes Wort eine gestemmte Form generiert. Wenn dem System das Wort bekannt ist, kann mit Hilfe des Indexes die entsprechende Grundform ermittelt und gespeichert werden. Aus dem Eingabe-String entsteht bei diesem Prozess eine Liste von QueryItem-Objekten, die alle Informationen für die spätere Suche enthalten. Die Rückgabe der `parse`-Methode ist ein Objekt, das neben dieser Liste und der Originaleingabe Hilfsmethoden bietet.

Aus der Eingabe „`What are the connections between morphic resonance and collective unconscious?`“ erzeugt der Parser folgende QueryItem-Instanzen:

```
what (start-pos 0) [basicform: what, stemmedForm: what, stopword:true ]
are (start-pos 4) [basicform: are, stemmedForm: are, stopword:true ]
the (start-pos 7) [basicform: the, stemmedForm: the, stopword:true ]
connections (start-pos 10) [basicform: connection, stemmedForm: connect, stopword:false ]
between (start-pos 20) [basicform: between, stemmedForm: between, stopword:true ]
morphic (start-pos 27) [basicform: morphic, stemmedForm: morphic, stopword:false ]
resonance (start-pos 34) [basicform: resonance, stemmedForm: reson, stopword:false ]
and (start-pos 43) [basicform: and, stemmedForm: and, stopword:true ]
collective (start-pos 46) [basicform: collective, stemmedForm: collect, stopword:false ]
unconscious (start-pos 56) [basicform: unconscious, stemmedForm: unconsci, stopword:false ]
```

Betrachtet man die gefundenen Stoppwörter `what`, `are`, `the`, `between` und `and`, lässt sich leicht erkennen, dass diese nicht zur näheren Kategorisierung der Frage dienen.

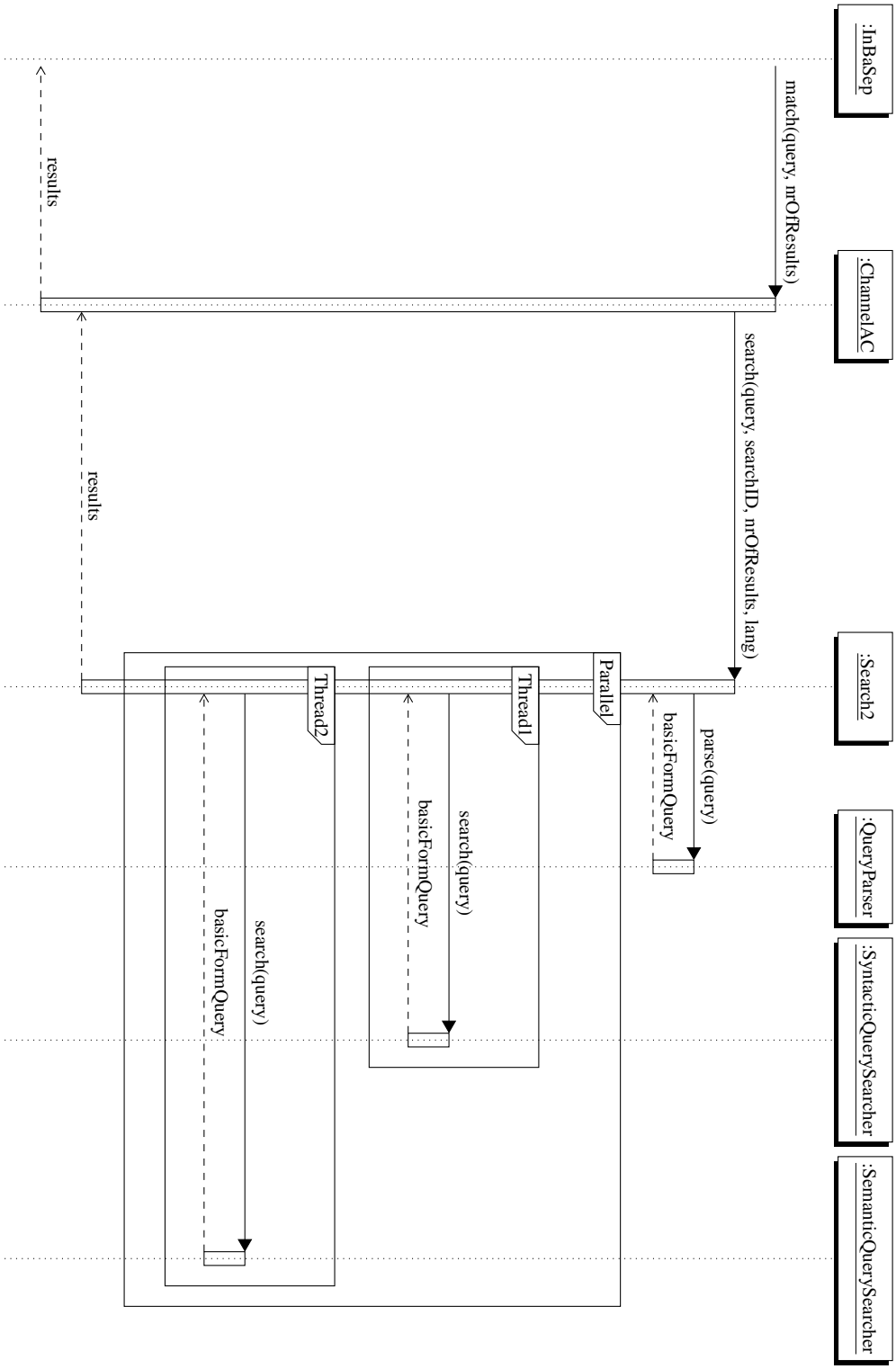


Abbildung 4.16: Sequenzdiagramm Suche

Listing 4.5: Parsen einer Query

```
1  public BasicFormQuery parse(String query) {
2      LinkedList<QueryItem> queryItems = new LinkedList<QueryItem>();
3      StringBuffer basicQueryString = new StringBuffer();
4
5      StringTokenizer tokenizer = new InbasepTokenizer(query);
6
7      int start = 0;
8      while (tokenizer.hasMoreElements()) {
9          String inputForm = tokenizer.nextToken();
10         String stemmedForm, basicForm;
11         boolean stopword = false;
12
13         try {
14             stemmer.setCurrent(inputForm);
15             stemmer.stem();
16             stemmedForm = stemmer.getCurrent();
17         } catch (NullPointerException e) {
18             stemmedForm = inputForm;
19             logger.debug("No stemmer available for Language " + language);
20         }
21
22         stopword = wordFinder.isStopword(inputForm);
23         basicForm = findBasicForm(inputForm, stopword);
24         int stringLength = basicForm == null ? 0 : basicForm.length();
25
26         QueryItem qi = new QueryItem(inputForm, stemmedForm, basicForm ,
               stopword);
27
28         queryItems.add(qi);
29         if (stringLength > 0) {
30             qi.setStart(start);
31             start += stringLength;
32             basicQueryString.append(qi.getBasicForm());
33         }
34     }
35
36     return new BasicFormQuery(queryItems, basicQueryString);
37 }
```

4.3.2 Syntaktische Suche

Die für diese Anwendung benötigte syntaktische Suche soll von einer Eingabe ausgehend syntaktisch ähnliche Fragen liefern. Da der Suchstring von einem menschlichen Benutzer oder einer Benutzerin kommt, ist mit Tipp- und Grammatikfehlern sowie einem abweichenden Fall bzw. Numerus zu rechnen. Es sollen also nicht nur Einträge gefunden werden, die zu 100% mit den hinterlegten Fragen übereinstimmen, sondern auch jene, die syntaktisch sehr nahe dran sind.

An der Beispielfrage „`What is the connection between morphic resonance and collective unconscious?`“ wird im Folgenden die syntaktische Suche mit Hilfe von Lucene besprochen und entwickelt. Bei der Bewertung der Suchergebnisse der einzelnen Experimente muss die gesuchte Frage den höchsten Lucene-Score erhalten. Bei den Testtabellen sind alle Abweichungen des Suchstrings, wenn man die zu suchende Frage als Ausgangsbasis nimmt, fett hervorgehoben.

Es werden die verschiedenen Lösungen betrachtet und in Hinsicht auf deren Fehler-Toleranz und Geschwindigkeit bewertet. Anhand der Ansätze wird dann die syntaktische Suche entstehen.

Präfix und Suffix Suche

Wenn man an syntaktische Ähnlichkeiten denkt, dann ist die einfachste Möglichkeit ein String-Vergleich. Da bei der Suche auch ein Präfix einer Frage ein Ergebnis liefern soll, ist es also nicht sinnvoll, auf eine komplette Gleichheit zu bestehen. Der erste Lösungsansatz ist also eine Präfix-Suche (siehe [4]), welche Fragen findet, die gleich beginnen wie der Suchstring.

In Tabelle 4.7 sieht man die Ergebnisse von Experimenten mit dieser Suchmethode, die mit Hilfe von Lucene umgesetzt wurden. Da es entweder eine Übereinstimmung gibt oder nicht, bekommen alle gefundenen Fragen einen Punkt im Lucene-Scoring, weshalb hier nicht weiter auf die Bewertung eingegangen wird. Die ersten beiden Testläufe liefern die erwarteten Einträge mit einer durchschnittlichen Suchzeit von unter einer Millisekunde zurück. Leider zeigt der dritte Test bereits den Nachteil dieser Variante, da eine Suche mit minimal abweichendem Suchstring bereits nichts mehr finden kann. Einen weiteren Nachteil sieht man im letzten Versuch, der einen Teil aus der Mitte einer Frage als Eingabe hat. Auch in solch einem Fall zeigt sich, dass die Präfix-Suche die Anforderungen nicht erfüllt.

Da es eine starke Einschränkung darstellt, dass der Suchstring am Beginn der Frage stehen muss, wird die Präfix-Suche mit ihrem Gegenstück, der Suffix-Suche, kombiniert. Wie sich in Tabelle 4.8 erkennen lässt, beseitigt dieser Ansatz zwar das erwähnte Problem, erhöht aber gleichzeitig die Laufzeit. Trotz dieser Änderung können die gewünschten Ergebnisse nicht erzielt werden, was auch diese Variante ungeeignet macht.

Tabelle 4.7: Präfix-Suche - Test

Query	Ergebnisse	Suchzeit
What is*	What is morphic resonance? What is the connection between morphic resonance and collective unconscious? What is the seventh sense? What is special about nkisi? What is parapsychology? What is telepathy? What is the difference between spirits and ghosts? What is your new book about? What is the hypothesis of formative causation? What is presentiment? What is the collective unconscious? What is consciousness? What is psi? What is an illusion? What is the connection between ghosts, paranormal activity, the human brain and science? What is the science behind telepathy? What is the science behind telekinesis? What is the difference between dream and supernatural? What is the difference between paranormal and supernatural? What is the science community's view on ghosts or spirits? What is research? is the search for the paranormal research? What is premonition? What is the sheep-goat effect?	$83295ns$
What is the connection*	What is the connection between morphic resonance and collective unconscious? What is the connection between ghosts, paranormal activity, the human brain and science?	$47478ns$
What is* **a** connection	$\emptyset$	$17791ns$
connection between morphic resonance	$\emptyset$	$30414ns$

Tabelle 4.8: Suffix-Suche - Test

Query	Ergebnisse	Suchzeit
What is the connection	What is the connection between morphic resonance and collective unconscious? What is the connection between ghosts, paranormal activity, the human brain and science?	$332872ns$
*What is **a** connection*	$\emptyset$	$200306ns$
connection between morphic resonance	what is the connection between morphic resonance and collective unconscious?	$344257ns$

Wort Konjunktion und Disjunktion

Nachdem die vorhergehenden Methoden, die sich auf die Verwendung des gesamten Eingabestrings zur Suche ähnlicher Fragen konzentriert haben, keine zufriedenstellenden Ergebnisse liefern konnten, ist der logische nächste Schritt, die Eingabe zu zerlegen. Beim Aufbau der Indexstruktur wird jede Frage für die spätere Suche in ihre Einzelteile zerlegt und so abgelegt. Bei der Suche wird der Eingabestring ebenfalls in einzelne Wörter aufgeteilt, von denen entweder alle (Konjunktion) oder zumindest eines (Disjunktion) in den Ergebnissen vorkommen muss. Für die Vorverarbeitung der Daten wird Lucenes `SimpleAnalyzer` [12] eingesetzt, der die Fragen und Suchstrings an Nicht-Buchstabenzeichen unterteilt und die einzelnen Teile in Kleinbuchstaben umwandelt.

Tabelle 4.9 zeigt die Ergebnisse von Testläufen, die für die Suche alle Worte disjunkt verknüpft verwendet haben. Es muss nicht jedes Wort mit der zu findenden Frage übereinstimmen, um das gesuchte Element als erstes Resultat zu erhalten. Wie hoch die Anzahl der Wörter ist, die im Suchstring abweichen dürfen, hängt davon ab wie ähnlich sich gespeicherte Fragen sind. Natürlich ist es auch entscheidend welche Worte falsch geschrieben wurden bzw. aus

anderen Gründen abweichen. Je höher der `idf`[2] Wert falsch geschriebener Elemente desto unwahrscheinlicher ist es die richtige Frage zu finden. Im vierten Test mussten alle Wörter bis auf „what“, „is“, „morphic“ und „resonance“ falsch geschrieben werden um die Frage nicht zu finden, da die meisten anderen enthaltenen Wörter immer zum gesuchten Element geführt haben. Anders sieht es im letzten Versuch aus, in dem nur ein Wort falsch geschrieben wurde, was zur Folge hatte, dass gleich zwei Fragen höher bewertet wurden als die tatsächlich gesuchte.

Tabelle 4.9: Disjunktion-Suche - Test

Lucene Query	Results	gefunden	Suchzeit
what is the connection between morphic resonance and collective unconscious	145	ja	170441 ns
what **are** the **connections** between morphic resonance and collective unconscious	123	ja	149157 ns
what is **a** connection between morphic resonance and collective **unconsciousn**	136	ja	162180 ns
what is the **conection betwen** morphic resonance **adn collectiv unconscios**	130	nein	197824 ns
what is the difference between **paranormall** and supernatural	143	nein	171805 ns

Tabelle 4.10: Konjunktion-Suche - Test

Lucene Query	Results	gefunden	Suchzeit
what is the connection between morphic resonance and collective unconscious	1	ja	21730 ns
what **are** the **connections** between morphic resonance and collective unconscious	0	nein	5683 ns
what is **a** connection between morphic resonance and collective **unconsciousn**	0	nein	8564 ns
what is **a conection betwen** morphic resonance **adn collectiv unconscios**	0	nein	50494 ns
what is the difference between **paranormall** and supernatural?	0	nein	12194 ns

Tabelle 4.11: Verbesserung der Laufzeit bei Verwendung von Konjunktion gegenüber Disjunktion

Lucene Query	Disjunktion	Konjunktion	Reduktion
what is the connection between morphic resonance and collective unconscious	170441	21730	87,25%
what **are** the **connections** between morphic resonance and collective unconscious	149157	5683	96,19%
what is **a** connection between morphic resonance and collective **unconsciousn**	162180	8564	94,72%
what is **a conection betwen** morphic resonance **adn collectiv unconscios**	197824	50494	74,48 %
what is the difference between **paranormall** and supernatural?	171805	12194	92,90 %

Die Suche mit einer Konjunktion der einzelnen Wörter wurde mit denselben Eingabe-Strings durchgeführt (siehe Tabelle 4.10). In Tabelle 4.11 kann man deutlich die Unterschiede zwischen

[2]Inverse Dokumenten Frequenz, ein Maß für die Einzigartigkeit eines Terms

dieser und der zuvor getesteten Methode erkennen. Der erste Unterschied zeigt sich in der Laufzeit. Die Variante mit den Term-Konjunktionen ist wesentlich spezifischer und liefert daher das Ergebnis in 3,8% bis 25,5% der Zeit, die der Ansatz mit disjunkten Wörtern für die Suche benötigt. Betrachtet man die Suchergebnisse, findet die oder-verknüpfte Suche im Test das Ergebnis, solange nicht zu viele Abweichungen von der gesuchten Frage herrschen. Allerdings treten schnell Probleme auf, wenn einzelne für die Frage charakteristische Wörter, Buchstabendreher oder andere kleine Abweichungen aufweisen. Ganz deutlich zeigt dies der fünfte Versuch, der zwar 143 Fragen - aus einem Pool von 208 - liefert, aber das gesuchte Element nicht das erste in der Ergebnisliste ist. Auch wenn die ersten drei Suchen erfolgreich waren, erkennt man an einer Ergebnismenge von beinahe 60% aller hinterlegten Fragen, dass man mit der Methode, die Disjunktionen verwendet, schnell in Probleme mit der Qualität laufen kann.

Ein weiterer Aspekt der Term-Konjunktion ist die „Exaktheit“ des Ergebnisses, denn nur wenn alle Worte des Suchstrings in der eingegebenen Form in einer Frage zu finden sind, kann die Suche das gewünschte Ergebnis finden. Da das System aber resistent gegen minimale Variationen (Tippfehler, Einzahl-Mehrzahl, usw.) sein soll, ist auch diese Idee weit weg von einer idealen Lösung.

Entfernen von Stoppwörtern

Bei Stoppwörtern handelt es sich um Wörter die um einiges öfter in analysierten Texten vorkommen als andere[5]. Ein Wort, das in fast jedem Dokument (bzw. in unserem Fall in jeder Frage) vorkommt, trägt nichts zur näheren Bestimmung bei. Auf der anderen Seite grenzt eine Suche mit einem Wort, das in Bezug auf den vorhandenen Textkorpus selten ist, die Ergebnismenge schnell ein. Für jede Sprache gibt es eine Liste, die aus Artikeln, Konjunktionen, Präpositionen und ein paar anderen Wörtern besteht. Je nach Anwendungsdomäne kann es notwendig sein diese allgemeine Liste noch um Elemente zu ergänzen, die eine hohe Frequenz in den verwendeten Dokumenten aufweisen. Die Verwendung von Stoppwörtern geht im Normalfall mit der Gewinnung von Speicherplatz und einer erhöhten Suchperformance einher. Im Fall der syntaktischen Suche spielen noch weitere Faktoren eine Rolle, die den Einsatz lohnenswert machen. Durch das Entfernen dieser Wörter aus gespeicherten Fragen und Sucheingaben kann die Frage auch gefunden werden, wenn die Eingabe z. B. andere Konjunktionen enthält.

Tabelle 4.12: Konjunktion-Suche mit Stoppworten - Test

Lucene Query	Results	gefunden	Suchzeit
~~what~~ ~~is~~ ~~the~~ connection ~~between~~ morphic resonance ~~and~~ collective unconscious	1	ja	12968 ns
~~is~~ **~~there~~** **~~a~~** connection ~~between~~ morphic resonance ~~and~~ collective unconscious	1	ja	12263 ns
~~what~~ ~~is~~ **~~a~~** connection ~~between~~ morphic resonance ~~and~~ collective unconscious	1	ja	12131 ns
~~what~~ ~~is~~ ~~the~~ connection ~~between~~ morphic resonance **adn** collective unconscious	0	ja	5514 ns

In Tabelle 4.12 lässt sich erkennen, welche Vor- und Nachteile aus dieser Variante entstehen. Die ersten drei Zeilen zeigen mehr oder minder dieselbe Frage, deren Repräsentation sich durch einige Wörter unterschiedet. Vom Umbau waren aber nur Wörter betroffen, die vor der Suche

aus der Anfrage entfernt wurden, da sie im System als Stoppwörter gekennzeichnet wurden. Da die Vorverarbeitung dieser drei Eingaben im gleichen Suchstring resultieren, stellt die nahezu gleiche Suchzeit auch keine Überraschung dar. Wie das letzte Beispiel zeigt, kann auch diese Methode nicht mit Tippfehlern umgehen. Ungewollte Resultate können auch entstehen, wenn der Stoppwort-Filter relativ viel aus der ursprünglichen Eingabe entfernt. Tabelle 4.13 demonstriert dieses Problem anhand zweier Beispiele. Trotz einer hundertprozentigen Übereinstimmung des Such-Strings mit einer der hinterlegten Fragen ist die Mächtigkeit der Ergebnismenge größer als eins.

Tabelle 4.13: Konjunktion-Suche mit Stoppworten - Probleme

Query	Results	Suchzeit
~~What~~ ~~are~~ poltergeists	What are poltergeists? Are poltergeists found all over the world?	$8840ns$
~~what~~ ~~is~~ ~~the~~ science ~~behind~~ telepathy	Is 'telepathy' a science? What is the science behind telepathy? Why are things like dream telepathy or the sense of being stared at taboo in the science community?	$13334ns$

Der Einsatz eines Stoppwort-Filters ist natürlich auch für disjunkte Queries möglich. Um diese Methode mit der vorhergehenden vergleichen zu können, wurden dieselben Testfälle abgearbeitet. Die ersten Testfälle, die in Tabelle 4.14 zu sehen sind, laufen hier alle einwandfrei durch. Dass im vierten Beispiel ein Ergebnis zurückgeliefert wird, ist allerdings nicht dem Stoppwort-Filter zu verdanken. Da die Suche durch die oder-verknüpften Elemente ungenauer wird, wächst gegenüber der Variante mit den per Konjunktion verbundenen Abfrageelementen die Ergebnismenge und die benötigte Suchzeit.

Tabelle 4.14: Disjunktion-Suche mit Stoppworten - Test

Lucene Query	Results	Gefunden	Suchzeit
~~what~~ ~~is~~ ~~the~~ connection ~~between~~ morphic resonance ~~and~~ collective unconscious	11	ja	37163 ns
~~is~~ ~~**there**~~ ~~**a**~~ connection ~~between~~ morphic resonance ~~and~~ collective unconscious	11	ja	35255 ns
~~what~~ ~~is~~ ~~**a**~~ connection ~~between~~ morphic resonance ~~and~~ collective unconscious	11	ja	34989 ns
~~what~~ ~~is~~ ~~the~~ connection ~~between~~ morphic resonance **adn** collective unconscious	11	ja	36280 ns

Um nach einigen Nachteilen dieser Variante zu den Vorteilen zu kommen, muss ein Vergleich mit einer anderen Such-Methode gezogen werden, der in Tabelle 4.15 gezeigt werden soll. Betrachtet man die disjunkten Queries mit und ohne vorgeschaltetem Stoppwort-Filter zeigt sich gleich, dass durch die Entfernung einiger Wörter aus der Suche die Ergebnismenge wesentlich keiner wird. Analog dazu sinkt auch die Zeit, die für die Abfrage benötigt wird.

In kurzen oder trivialen Fragen fallen oft beinahe alle Wörter dem Stoppwort-Filter zum Opfer. Da beim ersten Beispiel in Tabelle 4.16 nach dem Filter nur ein Wort übrig bleibt, ändert

Tabelle 4.15: Vergleich Disjunktion-Suche mit und ohne Stoppworte

Lucene Query	mit Filter	ohne Filter
what is the connection between morphic resonance and collective unconscious	145 (169829 ns)	11 (35710ns)
what is the science behind telepathy	149 (180570ns)	59 (84066ns)

sich hier nichts gegenüber der und-verknüpften Suche. Der zweite Test zeigt aber, dass trotz dieser Reduktion diese Methode auch keine zufriedenstellende Lösung darstellt.

Tabelle 4.16: Disjunktion-Suche mit Stoppworten - Probleme

Query	Results	Suchzeit
~~What are~~ poltergeists	What are poltergeists? Are poltergeists found all over the world?	$9300ns$
~~what is the~~ science ~~behind~~ telepathy	(59 results) is 'telepathy' a science? what is the science behind telepathy? ... life science has proven that all life on earth was created by bacteria into dna to us. is this right?	$79173ns$

Fuzzy Faktor

Um zu ermitteln wie ähnlich zwei Worte sind, kann die Levenshtein-Distanz [9] verwendet werden, die ein Maß dafür ist, wie viele Buchstaben verändert werden müssen um von einem String zum anderen zu kommen. Jedes Löschen, Ersetzen oder Einfügen eines Zeichens erhöht den Wert um eins. Will man beispielsweise von „Haus“ zu „Maus“ gelangen, muss das `H` durch ein `M` ersetzt werden, was also einer Distanz von 1 entspricht. Auf einen Wert von 2 kommt man beim Vergleich von Los und Laus (Los $\rightarrow$ Las $\rightarrow$ Laus).

Um das Ganze zu testen wird die Implementierung von Lucene verwendet, die statt der reinen Distanz einen Schwellenwert verwendet [12]. Mit Hilfe dieses Wertes skaliert die Distanz mit steigender Wortmenge nach oben. Tabelle 4.17 zeigt die Distanz abhängig von der Wortlänge mit einigen Schwellenwerten als Beispiel. Für die Verwendung in diesem Projekt ist ein Vorteil dieser Variante, dass sich bei längeren Worten mit Sicherheit mehr Tippfehler einschleichen als bei kurzen. Da dieser Schwellenwert linear steigt (Formel siehe unten) tritt aber ein Problem bei kurzen Strings auf. Vertauscht man beim Eingeben von `the` das `e` und das `h` so resultiert dies in einer Distanz von 2. Soll dieser „Buchstaben-Dreher“ abgefangen werden, müsste ein Schwellenwert von $0,3$ gewählt werde. Will man vom Wort „unconscious“ auf „consciousness“ kommen sind nur 6 Lösch- bzw. Einfügeoperationen notwendig. Aufgrund einer Wortlänge von 11 würde dieses Beispiel unter dem für das andere Beispiel verwendeten Wert von $0,3$ liegen.

Für die folgenden beiden Tests wurde ein Schwellenwert von $0,7$ gewählt um die Ergebnismenge bei der disjunkten Term-Verknüpfung nicht zu groß werden zu lassen. Es wird sich auch

herausstellen, welchen Einfluss dieser konservativ gewählte Schwellenwert auf die Lösung mit den Konjunktionen hat.

$$Schwellenwert = 1 - \frac{Distanz}{Länge}$$

Tabelle 4.17: Zusammenhang von Schwellenwort, Wortlänge und Distanz

Wortlänge \ Schwellenwert	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
2	1	1	1	1	1	0	0	0	0
3	2	2	2	1	1	1	0	0	0
4	3	3	2	2	2	1	1	0	0
5	4	4	3	3	2	2	1	1	0
6	5	4	4	3	3	2	1	1	0
7	6	5	4	4	3	2	2	1	0
8	7	6	5	4	4	3	2	1	0
9	8	7	6	5	4	3	2	1	0
10	9	8	7	6	5	4	3	2	1

Wie auch schon bei den anderen Suchmethoden soll hier sowohl die Konjunktion als auch die Disjunktion von Suchtermen getestet werden. Die in Tabelle 4.18 zu sehenden Ergebnisse einer und-verknüpften Suche lassen kaum Verbesserungen gegenüber der Suche ohne Fuzzy-Terme erkennen. In allen Beispielen steigt die Suchzeit ca. um das hundertfache. Bei dieser Variante ist aber bei den fünf Testfällen nur im letzten eine Verbesserung zu erkennen, denn ein Austausch von „the“ gegen „a“ verhindert eine erfolgreiche Suche. Obwohl diese Methode auf den ersten Blick vielversprechend wirkt, erreicht man keine zufriedenstellenden Ergebnisse.

Tabelle 4.18: Konjunktion-Suche mit Fuzzywert 0,7 - Test

Lucene Query	Results	gefunden	Suchzeit
what is the connection between morphic resonance and collective unconscious	1	ja	841461 ns
what **are** the **connections** between morphic resonance and collective unconscious	0	nein	819293 ns
what is **a** connection between morphic resonance and collective **unconsciousn**	0	nein	807295 ns
what is **a conection betwen** morphic resonance **adn collectiv unconscios**	0	nein	794840 ns
what is the difference between **paranormall** and supernatural?	1	ja	608051 ns

Bevor diese Variante verworfen wird, muss ein Test mit einem weniger einschränkenden Schwellenwert durchgeführt werden. Um auch „Buchstabendreher“ in Wörtern, die nur aus zwei oder drei Zeichen bestehen, zu erlauben, kann daher ein Schwellenwert von $0,3$ gewählt werden. Um den Einfluss dieses Wertes zu testen wurden Eingaben mit Tipp- und Rechtschreibfehlern verwendet. In Tabelle 4.19 sind die Ergebnisse im Vergleich zu sehen. Der „Preis“ für die weniger einschränkende Suche ist eine Erhöhung der Laufzeit auf beinahe das Doppelte.

Das erste Beispiel zeigt, dass die Suche wie erwähnt trotz Tippfehler bei einem Fuzzy-Wert von $0,3$ erfolgreich ist. Da sich die erlaubte Distanz mit der Wortlänge verändert entsteht bei dieser Methode ein Problem bei langen Wörtern. Im zweiten Test erkennt die Variante mit dem niedrigeren Schwellenwert eine Ähnlichkeit zwischen „collateral“ und „collective“. Das zuvor besprochene Problem, das der Austausch von Stoppwörtern darstellt, kann durch die Änderung des Schwellenwerts natürlich auch nicht behoben werden.

Tabelle 4.19: Konjunktion-Suche mit Fuzzywert Vergleich 0,7 und 0,3

Lucene Query	Ergebnis (SW 0,7)	Ergebnis (SW 0,3)
waht is the connection between morphic resonance **adn** collective unconscious	nein (807226 ns)	ja (1576249 ns)
what is the connection between morphic ressonanz and **collateral** unconscious	nein (772513 ns)	ja (1473231 ns)
what is **a** connection between morphic resonance and collective unconscious	nein (826488 ns)	nein (1432112 ns)

In Abschnitt 4.3.2 sieht man einen deutlichen Anstieg der Suchzeit von der Konjunktion zur Disjunktion. Aufgrund dieser Tatsche liegt die Vermutung nahe, dass es sich in Kombination mit den Fuzzy-Terms ähnlich verhalten wird. Die Tests zeigen allerdings, dass diese Annahme nicht stimmt und dass es bei der Verwendung von Fuzzy-Terms nur einen Unterschied von ca. 15% in der Laufzeit beider Varianten gibt. In allen 5 Versuchen kann außerdem das gewünschte Ergebnis gefunden werden, wie in Tabelle 4.20 zu sehen ist. Aus diesen Gründen kann diese Methode positiv bewertet werden.

Tabelle 4.20: Disjunktion-Suche mit Fuzzywert 0,7 - Test

Lucene Query	Results	Gefunden	Suchzeit
what is the connection between morphic resonance and collective unconscious	146	ja	955004 ns
what **are** the **connections** between morphic resonance and collective unconscious	131	ja	918059 ns
what is **a** connection between morphic resonance and collective **unconsciousn**	138	ja	924560 ns
what is **a conection betwen** morphic resonance **adn collectiv unconscios**	124	ja	904608 ns
what is the difference between **paranormall** and supernatural?	160	ja	732332 ns

Der „richtige“ Threshold bei den jeweiligen Varianten der Fuzzy-Term Suche hängt stark mit den gesteckten Zielen zusammen. Wie schon zuvor in Tabelle 4.19 zu sehen war, kann das Verringern des Wertes zu einer höheren Laufzeit und einer größeren Anzahl an unerwünschten Ergebnissen führen. Ein Ansatz wäre es, den von Lucene gewählten Weg zu umgehen und statt des „dynamischen“ Grenzwertes einen fixen Wortabstand zu wählen.

Da die Suche in Echtzeit während der Eingabe des Benutzers oder der Benutzerin ablaufen soll, muss ein Problem angesprochen werden, das dadurch bei der Verwendung der Fuzzy-Term Suche entstehen kann. Es ist schwierig festzustellen ob das letzte Wort schon fertig geschrieben wurde oder noch nicht. Betrachtet man die Suche, die Fuzzy-Terms mittels Kojunktion verbindet, so führt ein unvollständig geschriebenes Wort oft zu keinen bzw. unerwünschten Ergebnissen. In Tabelle 4.21 sieht man die drei verschiedenen Ausgänge einer solchen Suche, die zu Testzwecken mit einem niedrigen Schwellenwert von $0,3$ durchgeführt wurde. Während das

Tabelle 4.21: Disjunktion-Suche mit Fuzzywert 0,3 - Beispiel

Query	Results
what is the con	life science has proven that all life on earth was created by bacteria into dna to us. is this right? how does a dog know when its owner is returning home at an unexpected time? what is research? is the search for the paranormal research? what is the science community's view on ghosts or spirits? in the trinity model of man as: body-mind, soul (consciousness) and spirit, what is spirit, also referred to as life? is there a way you can learn to see ghosts? is there any way how you can get supernatural powers? why do some people not believe in the supernatural?
what is the connection betw	what is the connection between morphic resonance and collective unconscious? what is the connection between ghosts, paranormal activity, the human brain and science? is it true that science has found proof of a correlation between consciousness and brain activity?
what is the connection betwe	∅

erste Beispiel zwar eine Menge Fragen zurückliefert, ist die Gesuchte nicht dabei. Wird der Suchstring ein wenig erweitert, ist das Gesuchte in den Ergebnissen an erster Stelle. Ein einziger Buchstabe mehr kann aber schon wieder Probleme verursachen, wie im letzten Test zu sehen ist.

Stemming

Um beim Vergleichen weniger Probleme mit unterschiedlichen Wortformen zu haben, können Algorithmen eingesetzt werden, die Eingaben in eine Grundform normalisieren. Für dieses Verfahren, das im Information Retrieval Stemming genannt wird, existieren in diversen Sprachen verschiedene Ansätze.

Lucene setzt auf Snowball, eine von Dr. Martin Porter entwickelte Sprache zur String-Verarbeitung, um die verschiedenen Implementierungen effizient anzubieten. Für die englische Sprache steht neben der Umsetzung des original Porter-Stemmers [15] eine verbesserte Version zur Verfügung. Diese wird auch für die folgenden Tests eingesetzt um die Möglichkeiten und Grenzen dieser Methode aufzuzeigen. In Tabelle 4.22 sieht man, dass regelmäßige Verben und Nomen kein Problem darstellen und immer auf die selbe gestemmte Form reduziert werden können. Sobald man aber auf irreguläre Wörter trifft, stößt der Stemming-Algorithmus an seine Grenzen. Da mit grammatikalischen Bildungsregeln gearbeitet wird stellen auch Buchstabendreher wie im letzten Beispiel für den Algorithmus kein Hindernis dar, lediglich der „Fehler“ ist dann auch in der gestemmten Form zu finden.

Tabelle 4.22: Stemming Beispiele

Original	Stemmed
believe	believ
believed	believ
ghost	ghost
ghosts	ghost
cactus	cactus
cacti	cacti
beleive	beleiv
beleived	beleiv

4.3.3 Implementierung - Syntaktische Suche

Nachdem jetzt einige Methoden gezeigt wurden, die in einer syntaktischen Suche eingesetzt werden können, wird auf den folgenden Seiten erläutert, aus welchen Komponenten sich die Implementierung zusammensetzt. Um Einbußen, die man bei einer bestimmten Prozedur in Kauf nehmen muss, zu minimieren, können verschiedene Verfahren gekoppelt werden. Da alle vorgestellten Ansätze Vor- und Nachteile bezüglich der Ergebnisse, der Suchzeiten und der Stabilität haben, gibt es viele Kombinationsmöglichkeiten.

Auswahl von Methoden

Betrachtet man die Ergebnisse aller erwähnten Methoden so zeigt sich, dass die Laufzeit nur in einem Fall über 1 ms liegt. Aus diesem Grund spielt die Laufzeit bei der Auswahl der Kombinationen eine untergeordnete Rolle. Die in Unterabschnitt 4.3.2 aufgezeigten Möglichkeiten werden im Folgenden für die gewünschte Anwendung bewertet. Natürlich ist die *Präfix- und Suffix-Suche* für den Einsatz hier nicht geeignet, da hier nur bei hundertprozentiger Übereinstimmung Ergebnisse gefunden werden. Die Konjunktion von Wörtern bietet nur den Vorteil, dass die Reihenfolge der einzelnen Worte beliebig ist und ist daher für sich alleine gesehen auch nicht einsetzbar. Die Disjunktion liefert ohne weitere Einschränkung eine große Menge an Fragen zurück und reiht die gesuchte Frage bei fehlerhaften Eingaben nicht immer an die erste Stelle der Rückgabe. Da es das Ziel ist, syntaktisch ähnliche Fragen zu finden, würde durch das Entfernen von Stoppwörtern bei Fragen, die hauptsächlich aus solchen bestehen, ein Problem auftreten. Aus diesem Grund findet dieser Ansatz in der Lösung keinen Platz.

Der Fuzzy Faktor ist offensichtlich der ideale Ansatz um das Problem der Tippfehler in der Eingabe zu lösen und ist daher Bestandteil der Syntaktischen Suche. Offen bleibt noch die Frage wie die einzelnen Terme verknüpft werden und wie man den Fuzzy-Wert für ideale Ergebnisse wählt. Wenn man die Ergebnisse der oder-verknüpften Fuzzy Suche in Tabelle 4.20 betrachtet, wirken diese auf den ersten Blick vielversprechend. Das Problem bei der großen Menge an Fragen die zurückgeliefert wird ist es, die „richtigen“ von den „falschen“ zu unterschieden. Auch

wenn der eingegebene String nur in einem Wort mit einem Element übereinstimmt ist dieses in der Ergebnismenge enthalten. Dass die Entscheidung, wie Ergebnisse einer disjunkten Fuzzy Suche bewertet werden sollen, nicht einfach ist, zeigt Tabelle 4.23. Während der erste Test einen String verwendet, der nur im ersten Teil mit einer hinterlegten Frage übereinstimmt, enthält die zweite Suchanfrage nur in einem Wort Tippfehler. Bei den Rückgaben ist zu erkennen, dass in beiden Fällen die Treffer von Lucene ähnlich bewertet wurden.

Tabelle 4.23: Disjunktion-Suche mit Fuzzywert - Probleme

Query	Results	Score
what is the diffenrece between obesity and eating	what is the connection between morphic resonance and collective unconscious?	1.20
	what is the difference between spirits and ghosts?	1.20
	what is the difference between dream and supernatural?	1.20
	what is the difference between paranormal and supernatural?	1.20
	what is the connection between ghosts, paranormal activity, the human brain and science?	1.01
	...	
what is the diffenrece between **parannormall** and supernatural	what is the difference between dream and supernatural?	1.61
	what is the difference between paranormal and supernatural?	1.61
	what is the connection between morphic resonance and collective unconscious?	1.07
	what is the difference between spirits and ghosts?	1.07
	what is the connection between ghosts, paranormal activity, the human brain and science?	0.90
	...	

Da eine (verbesserte) syntaktische Suche entwickelt werden soll, eignet sich diese hohe Unsicherheit bei der disjunkt verknüpften Suche nicht, daher werden die Terme in der finalen Implementierung per Konjunktion verknüpft. Diese Entscheidung beruht auch auf der Tatsache, dass mit dieser Implementierung „nur“ die semantische Komponente unterstützt und ergänzt werden soll.

Da Stemming den Vorteil bietet „grammatikalisch unabhängiger“ zu sein, würde sich auch dieser Ansatz für die Implementierung anbieten. In dieser Variante kann das Problem entstehen, dass durch Stemming die Distanz eines beinahe richtig eingegebenen Wortes (z. B. nur 1 Buchstabe fehlt) zu einem Feld im Index erhöht wird, was wiederum die Fuzzy Suche behindert. Um dieses Problem zu umgehen benötigt man einen Suchterm der das gestemmte und das originale Wort für die Suche verwendet. Dieser Ansatz wird in diesem Kapitel als „Variante 1“ bezeichnet. Um einen Vergleich zu erreichen wird bei der „Variante 2“ der Term mit dem Originalwort weggelassen.

Da die gewünschte Anwendung auch für unfertig eingegebene Fragen gute Ergebnisse liefern soll, muss ein besonderes Augenmerk auf das aktuell „letzte“ Wort gelegt werden. Um Probleme zu umgehen, kann dieses aus der Suche weggelassen werden oder durch einen Präfix Suchterm abgedeckt werden.

Tabelle 4.24 zeigt, dass beide Möglichkeiten kein Problem haben, eine fehlerfrei eingegebene Frage zu finden. Gegen Fehler, die am Wortende angesiedelt sind, ist die erste Variante wesentlich resistenter und daher von einem Qualitätsstandpunkt aus zu bevorzugen.

Tabelle 4.24: Test von Variante 1 und 2

	Variante 1			Variante 2		
Fuzzy Wert	0.3	0.5	0.7	0.3	0.5	0.7
What is the connection between morphic resonance and collective unconscious?	1	1	1	1	1	1
What is the difference between **paranormall** and supernatural?	1	1	1	1	1	0
Is it possible that me and a person in another country are **experiencingen** paranormal things at the same time?	1	1	1	1	0	0

Implementierung

In Listing 4.6 ist eine Hilfsmethode der syntaktischen Suche zu sehen, die verwendet wird um die entsprechende Lucene Query zu erzeugen. Alle Worte, bis auf das letzte, werden in eine `FuzzyQuery` umgewandelt und als verpflichtendes Element (`BooleanClause.Occur.MUST`) in die `BooleanQuery` eingehängt. Für das abschließende Wort wird eine Subquery angelegt, die neben der `FuzzyQuery` auch eine `PrefixQuery` enthält. Die am Ende eingefügte `TermQuery` ist für die Suche hier nicht von Bedeutung, sondern wird verwendet um gegebenenfalls deaktivierte Fragen zu filtern.

Listing 4.6: Hilfsmethode Query

```
1  public static BooleanQuery querySimilarQueries(BasicFormQuery bfq,
       boolean onlyActiveQuery, float fuzzyFactor) {
2      BooleanQuery q = new BooleanQuery();
3      String basicForm;
4      int bfqSize = bfq.getSize();
5
6      for (int i = 0; i < bfqSize-1; i++){
7          basicForm = bfq.getQueryItem(i).getBasicForm();
8          q.add(new FuzzyQuery(new Term("query", basicForm), fuzzyFactor),
               BooleanClause.Occur.MUST);
9      }
10
11     BooleanQuery qsub = new BooleanQuery();
12     basicForm = bfq.getQueryItem(bfqSize-1).getBasicForm();
13     qsub.add(new FuzzyQuery(new Term("query", basicForm), fuzzyFactor),
           BooleanClause.Occur.SHOULD);
14     qsub.add(new PrefixQuery(new Term("query", basicForm)), BooleanClause.
           Occur.SHOULD);
15     q.add(qsub, Occur.MUST);
16
17     if (onlyActiveQuery)
18         q.add(new TermQuery(new Term("active", "T")), BooleanClause.Occur.
               MUST);
19
20     return q;
21 }
```

Die syntaktische Suche, die in Listing 4.7 zu sehen ist, verwendet die soeben gezeigte Hilfsmethode zur Erstellung der Queries. Die Besonderheit dieser Implementierung ist, dass die von Lucene errechneten Scores mit einem Faktor multipliziert werden um vor den semantischen Ergebnissen zu landen.

Listing 4.7: syntaktische Suche

```
1  public PriorityQueueMap searchSyntactic(BasicFormQuery bfq, long sID, int
       nrOfResults, Lang language, boolean onlyActiveQuery) {
2      PriorityQueueMap pqm = new PriorityQueueMap(nrOfResults);
3      BooleanQuery q = LuceneWrapper.querySimilarQueries(bfq,
           onlyActiveQuery, FUZZY_FACTOR);
4
5      TopDocs hits = LuceneManager.getHits(q, is, nrOfResults);
6      // No hits for the query
7      if (hits == null) return pqm;
8
9      for (ScoreDoc scoreDoc : hits.scoreDocs) {
10         Document d = LuceneManager.getDoc(scoreDoc.doc, is);
11         if (d == null) continue;
12
13         Long queryID = LuceneWrapper.fromQueryQueryID(d);
14         String qString = LuceneWrapper.fromQueryQueryString(d);
15         double score = scoreDoc.score * SCORE_MULTIPLIER;
16         pqm.put(queryID, score, qString);
17     }
18     return pqm;
19 }
```

4.3.4 Semantische Suche

Die Beschleunigung der semantischen Suche wie sie für dieses Projekt realisiert wurde besteht aus zwei wesentlichen Komponenten, der Vorberechnung möglicher semantischer Kombinationen und dem Abruf dieser Informationen. Während in Unterabschnitt 4.2.2 und Unterabschnitt 4.2.3 die Vorberechnung und Speicherung dieser Daten erklärt wurde, soll dieses Unterkapitel auf die Such-Komponente eingehen.

Wenn jede mögliche Kombination im Vorhinein berechnet werden würde, wäre es sehr einfach diese im entsprechenden Index zu finden und die passenden Fragen zurückzuliefern. Wie allerdings im entsprechenden Kapitel erwähnt wurde ist die Menge der tatsächlich abgelegten Stichworte stark eingeschränkt.

Indexstruktur

Für den Aufbau des Indexes gibt es zwei verschiedene Ansätze, die einen Einfluss auf die dazugehörige Suche und deren Laufzeit haben. Die Struktur des Indexes wurde bereits in einem anderen Kapitel (siehe Abschnitt 4.2) erläutert, allerdings wurde auf Vor- und Nachteile für die Suche nicht eingegangen. Deshalb sollen in den folgenden Absätzen die beiden Möglichkeiten erläutert werden.

Die erste Idee legt alle Schlüsselwort-Kombinationen in einem gemeinsamen Index ab und lässt eine Suche darüber laufen. Eine Abfrage in Lucene kann die einzelnen Sub-Queries mittels Konjunktion und Disjunktion verbinden. Verwendet man ersteres bei einer Suche mit n Keywords, so kann kein Eintrag mit $n-1$ gefunden werden, selbst wenn diese Schlüsselwörter eine Teilmenge der ursprünglichen Query darstellen. Als Beispiel würde die Query $A \wedge B \wedge C$ die Kombination A, B nicht finden. Die Tatsache, dass nicht alle theoretisch möglichen Kombinationen erzeugt und abgespeichert werden, kann dazu führen, dass ein Element der Suche dafür verantwortlich ist, dass kein Ergebnis gefunden wird. Natürlich lässt sich dieses Problem mit Hilfe der Disjunktion lösen, es entsteht allerdings ein neues. Eine Suche würde ohne weitere Einschränkungen alle Einträge zurückliefern, bei denen die Schnittmenge der Schlüsselwörter nicht leer ist. Die Query $A \vee B \vee C \vee D$ liefert zum Beispiel auch Einträge mit den Keywords A, X, Y oder A, B, Z zurück. Da die Folgen, eine extrem große Ergebnismenge und eine hohe Suchzeit, nicht den Anforderungen entsprechen, kann nicht auf diese Art gesucht werden. In Tabelle 4.25 werden die Ergebnisse der beiden vorgestellten Suchmöglichkeiten in einem einzelnen Index anhand einer kleinen Beispielmenge gezeigt.

Kombination	Frage
A B C D	Q1
A B C X	Q2
A X Y Z	Q3
A B C	Q4
A B Y	Q5
A B	Q6
A C	Q7
A X	Q8
X Y	Q9
A	Q10
B	Q11
C	Q12
X	Q13

(a) Single Index

Kombination	Frage
A B C D	Q1

(b) Suche: $A \wedge B \wedge C \wedge D$

Kombination	Frage
A B C D	Q1
A B C X	Q2
A X Y Z	Q3
A B C	Q4
A B Y	Q5
A B	Q6
A C	Q7
A X	Q8
A	Q10
B	Q11
C	Q12

(c) Suche: $A \vee B \vee C \vee D$

Tabelle 4.25: Beispiel für die semantische Vorberechnung mit einer Indexstruktur

Der zweite Ansatz legt für jede berechnete Kombinationslänge (=Anzahl der Schlüsselwörter) eine Indexstruktur an. Ohne weitere Einschränkungen würden aus n dem System bekannten Keywords auch bis zu n dieser Indizes entstehen. Da in der Praxis eine Anfrage meistens wenige Schlüsselwörter enthält, kann die Erstellung bei einer definierten Länge abgebrochen werden. Die Suche selbst wird zuerst im dem Index begonnen dessen Länge der Anzahl der übergebenen Keywords entspricht. Da die einzelnen Terme mit Hilfe von Konjunktionen verknüpft werden ist diese Anfrage schnell und performant. Da natürlich auch für diese Variante nicht alle theoretisch möglichen Kombinationen berechnet und abgelegt werden können, ist die Suche damit

nicht vollständig. Aus diesem Grund müssen schrittweise auch alle Indizes mit einer kürzeren Länge durchsucht werden. Auch hier gilt, dass eine Suche mit n konjunktiv verknüpften Schlüsselwörtern in einem Index mit Kombinationslängen kleiner n zu keinen Ergebnissen führen würde. Der Vorteil der aufgeteilten Indexstrukturen erlaubt der Abfrage mit disjunkt verbundenen Keywords eine spezielle Randbedingung. Die Suche kann verlangen, dass von den n Termen in einem Index mit der Kombinationslänge m (wobei $m < n$) genau m Terme erfüllt werden müssen.

Komb.	Frage
A	Q10
B	Q11
C	Q12
X	Q13

(a) Index 1

Komb.	Frage
A B	Q6
A C	Q7
A X	Q8
X Y	Q9

(b) Index 2

Komb.	Frage
A B C	Q4
A B Y	Q5

(c) Index 3

Kombination	Frage
A B C D	Q1
A B C X	Q2
A X Y Z	Q3

(d) Index 4

Kombination	Frage	Sub-Suche	Suchort
A B C D	Q1	$A \wedge B \wedge C \wedge D$	Index 4
A B C	Q4	$A \vee B \vee C \vee D$ (3 müssen übereinstimmen)	Index 3
A B	Q6	$A \vee B \vee C \vee D$ (2 müssen übereinstimmen)	Index 2
A C	Q7		
A	Q10	$A \vee B \vee C \vee D$	Index 1
B	Q11		
C	Q12		

(e) Suche

Tabelle 4.26: Beispiel für die semantische Vorberechnung mit mehreren Indizes

Obwohl zu Beginn des Unterkapitels von nur zwei Lösungen die Rede war, gibt es noch eine weitere. Da bei dieser allerdings die Suche aus der zweiten Variante auf einen einzelnen erweiterten Index angewendet wird, handelt es sich dadurch nicht um eine eigenständige Möglichkeit. Die Länge, die zu jeder Kombination abgelegt wird, stellt einen zusätzlichen verpflichteten Suchterm dar. Da dadurch allerdings kein erkennbarer Vorteil entsteht wird nicht weiter auf diese Mischlösung eingegangen.

Scoring

Bei der Vorberechnung des semantischen Indexes erhält man vom verwendeten Matching-Algorithmus einen Wert der die Ähnlichkeit von einer Kombination zu einer semantisch ähnlichen Frage beschreibt. In einem theoretischen System, das für jede mögliche Schlüsselwortverknüpfung diese Kalkulation durchgeführt hat, könnte man diesen Score eins zu eins für das Suchergebnis verwenden. Nachdem allerdings die Suche über eine unvollständige Menge von

unterschiedlich langen Kombinationen durchgeführt wird müssen die erhaltenen Punkte neu bewertet werden. Die Werte des von Lucene im Normalfall verwendeten Scoring basieren auf Ähnlichkeitsmaßen wie z. B. der Termfrequenz und haben keinerlei Bedeutung für die semantische Suche. Um die Weiterverarbeitung zu vereinfachen ist es der beste Ansatz, für jeden Suchterm der in einem Treffer vorkommt einfach 1 zu addieren.

Die erste Möglichkeit dies zu erreichen ist es, maßgeschneiderte Scoring Klassen zu entwickeln. Dieser Ansatz hat den Vorteil, dass sich die Subroutinen auf das notwendigste reduzieren lassen und somit keine zusätzliche Zeit für unnötige Berechnungen anfällt. Eine solche Implementierung ist in einem Bereich angesiedelt der stark mit Lucene Kernkomponenten verknüpft ist und müsste daher bei jeder Änderung des Frameworks getestet und möglicherweise neu überarbeitet werden.

Um diese Probleme zu umgehen bietet Lucene mit `ConstantScoreQuery` die Möglichkeit den Score einer erfüllten Query (bzw. Subquery) mit konstanten Punkten zu bewerten. In diesem konkreten Fall würde jedes Schlüsselwort der Query, das mit einem Keyword eines Treffers übereinstimmt, beispielsweise den Lucene-Score um 1 erhöhen. Die vorberechneten Punkte des semantischen Matching-Algorithmus werden anschließend mit dem Wert der Lucene Suche multipliziert. Der erhaltene Wert kann für die Reihung der Suchergebnisse verwendet werden.

Im Kernbereich der Arbeit wird allerdings nicht weiter drauf eingegangen, wie die semantischen Scores mit unterschiedlicher Kombinationslänge neu bewertet werden und was als konstanter Wert angenommen werden soll. Das Problem und mögliche Lösungen und Optimierungen werden in Abschnitt 4.5.2 kurz erläutert.

Implementierung

Für die Implementierung der semantischen Suche werden einige Hilfsklassen benötigt, die in der Folge kurz besprochen werden sollen. Bei `InbasepQuery` handelt es sich um eine Erweiterung der `BooleanQuery`, einer Standardklasse von Lucene, zum komfortableren Erstellen von Suchen in den semantischen Vorberechnungen. In Listing 4.8 sieht man, dass dem Konstruktor mittels Parameter mitgeteilt wird, ob die später hinzugefügten Schlüsselwörter alle vorkommen müssen oder nicht. Die ebenfalls benötigte boolesche Variable `onlyActive` ist für die eigentliche Suche nicht weiter von Bedeutung, sondern wird nur dafür verwendet um einen zusätzlichen Suchterm hinzuzufügen um gewisse Einträge zu filtern. Die `add` Methode kapselt das übergebene Keyword in einer `ConstantScoreQuery`.

Listing 4.8: Hilfsklasse InbasepQuery

```
1  public class InbasepQuery extends BooleanQuery {
2      private Occur occur;
3      private static final ConstantScoreQuery activeQuery =  new
           ConstantScoreQuery(new TermQuery(new Term ("active","T")));
4
5      static {
6          activeQuery.setBoost(0);
7      }
8
9      public InbasepQuery(boolean must, boolean onlyActive) {
10         if (must) occur = Occur.MUST;
11         else occur = Occur.SHOULD;
12         if (onlyActive) add(activeQuery, Occur.MUST);
13     }
14
15     public void add(String keyword) {
16         add(new ConstantScoreQuery(new TermQuery(new Term(LuceneFieldConst
               .FIELD_KEYWORD, keyword))), occur);
17     }
18
19 }
```

Die Berechnung der Scores, die aus Informationen des Indexes, den Berechnungen des `ConstantScoreQuery` und einem Multiplikator bestehen, benötigen mehr Funktionalität als Lucene bei einer normalen Query anbietet. Daher wurde ein Collector implementiert um das Sammeln der Treffer zu übernehmen. Ein weiterer Grund ist die nicht Standard-konforme Art die Ergebnisse zu speichern. In Listing 4.9 ist die Umsetzung dieses `ScoreCollector` zu sehen. Die Funktion `collect` berechnet die neuen Punkte für einen Treffer und ruft mit diesen Informationen die `put` Methode der Hilfsklasse `PriorityQueueMap` auf.

Listing 4.9: Hilfsklasse ScoreCollector

```
1  public class ScoreCollector extends Collector{
2      private static final int SCORE_MULTIPLICATOR = 2;
3      private long[] qIds;
4      private float[] qRank;
5      private Scorer scorer;
6      private PriorityQueueMap pqm;
7
8      public ScoreCollector(int size) {
9          pqm = new PriorityQueueMap(size);
10     }
11
12     public boolean isFull() {
13         return pqm.isFull();
14     }
15
16
17     @Override
```

```
18      public boolean acceptsDocsOutOfOrder() {
19          return true;
20      }
21
22      public PriorityQueueMap getPriorityQueueMap () {
23          return pqm;
24      }
25
26      @Override
27      public void setScorer(Scorer scorer) throws IOException {
28          this.scorer = scorer;
29      }
30
31      @Override
32      public void collect(int doc) throws IOException {
33          float score = SCORE_MULTIPLICATOR * scorer.score() * qRank[doc];
34          pqm.put(qIds[doc], score);
35      }
36
37      @Override
38      public void setNextReader(IndexReader reader, int docBase) throws
            IOException {
39          qIds = FieldCache.DEFAULT.getLongs(reader, LuceneFieldConst.
                FIELD_QUERY_ID);
40          qRank = FieldCache.DEFAULT.getFloats(reader, LuceneFieldConst.
                FIELD_SEMANTIC_QUERY_RANK);
41      }
42
43  }
```

Die QueryID und die dazugehörigen Punkte der Treffer während der semantischen Suche müssen effizient verwaltet werden. Da die Vorberechnungen auf mehrere Datenstrukturen mit verschieden langen Schlüsselwortkombinationen verteilt sind, passiert es, dass während einer Suche ein und dieselbe Frage (QueryID) mit unterschiedlichen Scores mehrmals gefunden wird. Zu den Anforderungen gehört es daher, dass eine QueryID in der Ergebnismenge nur einmal vorkommen darf. Deswegen wird nur eine eingeschränkte Zahl an Ergebnissen benötigt, und zwar die Elemente mit den höchsten Scores.

Da die `PriorityQueueMap` (siehe Listing 4.10) ein Herzstück der semantischen Suche darstellt, muss auch diese im Detail besprochen werden. Bei der Instanzierung dieser Klasse wird eine Größe übergeben, die der maximalen Anzahl an Fragen entspricht, welche die Datenstruktur halten kann. Beim Hinzufügen eines Treffers müssen zwei Fälle unterschieden werden. Der erste Teil der `put` Methode entspricht dem Einfügen in eine „volle“ Datenstruktur. Nur wenn der Score besser ist als der des schlechtesten Elements (`bottomScore`) wird die `replace` Methode aufgerufen, die ebenfalls zwei Möglichkeiten unterscheiden muss. Ist die QueryID noch nicht abgelegt, fällt der Eintrag mit dem schlechtesten Score weg, um Platz für das neue Element zu machen. Sollte die Frage bereits vorhanden sein, werden nur die Punkte des Items upgedatet. Der `else` Zweig von `put` wird aufgerufen, wenn die Anzahl der Elemente in der Datenstruktur unter der definierten Größe ist. Das dort ausgeführte `add` aktualisiert Punkte für eine bereits abgelegte QueryID, oder fügt unbekannte Fragen ein. Des Weiteren verfügt die Klasse über ei-

ne Methode um Datenstrukturen dieser Art zusammenzuführen, wenn die Ergebnisse anderer Suchen vereint werden sollen.

Listing 4.10: Hilfsklasse PriorityQueueMap

```
1  public class PriorityQueueMap {
2      private Map<Long,InbasepSearchResult> matches = new HashMap<Long,
           InbasepSearchResult>();
3      private PriorityQueue<InbasepSearchResult> pq = new PriorityQueue<
           InbasepSearchResult>(10);
4      private double bottomScore = -1;
5      private int maxSize;
6      private int size;
7
8      public PriorityQueueMap(int maxSize) {
9          size = 0;
10         this.maxSize = maxSize;
11     }
12
13     public boolean isFull() {
14         return maxSize == size;
15     }
16
17     public PriorityQueue<InbasepSearchResult> getQueue () {
18         return pq;
19     }
20
21
22     public boolean put(long queryID, double score) {
23         return put(queryID, score, "");
24     }
25
26     public boolean put(long queryID, double score, String qString) {
27         if (size == maxSize) {
28             if (score > bottomScore) {
29                 replace (queryID, score, qString);
30             }
31         } else {
32             //simply add
33             add(queryID, score, qString);
34         }
35
36         return true;
37     }
38
39     private void replace (long queryID, double score, String qString) {
40         InbasepSearchResult qsr =matches.get(queryID);
41         try {
42             //already in map
43             updateScore(qsr, score, qString);
44         } catch (NullPointerException e) {
45             matches.remove(pq.poll().getId());
46             addElement(queryID, score, qString);
47             bottomScore = pq.peek().getScore();
```

```
48          }
49      }
50
51      private void add (long queryID, double score, String qString) {
52          InbasepSearchResult qsr =matches.get(queryID);
53          try {
54              //already in map
55              updateScore(qsr, score, qString);
56
57          } catch (NullPointerException e) {
58              addElement(queryID, score, qString);
59              size++;
60              if (size == maxSize) bottomScore = pq.peek().getScore();
61          }
62      }
63
64      private void updateScore(InbasepSearchResult qsr, double score, String
            qString) {
65
66          qsr.updateScore(score, qString);
67          pq.remove(qsr);
68          pq.add(qsr);
69      }
70
71      private void addElement (Long queryID, double score, String qString) {
72          InbasepSearchResult newQSR = new InbasepSearchResult(queryID,
                qString, score);
73          pq.add(newQSR);
74          matches.put(queryID, newQSR);
75      }
76
77      private int getSize() {
78          return size;
79      }
80
81      public void add(PriorityQueueMap pqm) {
82          try {
83              int otherSize = pqm.getSize();
84              bottomScore = -1;
85              maxSize += otherSize;
86              for (InbasepSearchResult sr : pqm.getQueue()) {
87                  add(sr.getId(), sr.getScore(), sr.getQuery());
88              }
89              maxSize = size;
90          } catch (NullPointerException e) {
91              return;
92          }
93      }
94
95  }
```

Die Methode in der die eigentliche Suche abläuft ist in Listing 4.11 zu sehen. Anfangs werden zwei Instanzen der Klasse `InbasepQuery` erstellt, eine für die konjunktive und eine für

die disjunktive Verknüpfung von Schlüsselwörtern. Da erstere nur benötigt wird, wenn die Anzahl der Schlüsselwörter nicht die maximal berechnete Kombinationslänge überschreitet, wird diese auch nur im gegebenen Fall initialisiert und verwendet. Die disjunktive Suche wird iterativ auf Mengen an vorberechneten Kombinationen angewandt, deren Kombinationslänge immer kürzer wird, bis die gewünschte Anzahl an Ergebnissen gefunden oder alle Ebenen durchsucht wurden. Das aus dieser Methode möglicherweise entstehende Problem und Lösungsansätze werden in Abschnitt 4.5.2 erläutert.

Listing 4.11: Sematische Suche

```
1  private PriorityQueueMap iterativeSearch(Set<String> queryKeywords, int
       level, ScoreCollector collector) {
2          InbasepQuery qMust = null;
3          InbasepQuery qShould = new InbasepQuery(false, true);
4          if (level <= semanticMaxLength) {
5              qMust = new InbasepQuery(true, true);
6          }
7          for (String queryKeyword : queryKeywords) {
8              try {
9                  qMust.add(queryKeyword);
10             } catch (NullPointerException e) {
11             }
12
13             qShould.add(queryKeyword);
14         }
15
16         int i = level > semanticMaxLength ? semanticMaxLength : level - 1;
17         if (qMust != null) {
18             LuceneManager.getHits(qMust, matcherIS[i], collector);
19         }
20
21         for (; i > 0; i--) {
22             if (collector.isFull()) break;
23             qShould.setMinimumNumberShouldMatch(i);
24             LuceneManager.getHits(qShould, matcherIS[i-1], collector);
25         }
26
27         PriorityQueueMap pqm = collector.getPriorityQueueMap();
28         return pqm;
29     }
```

4.4 Tests

4.4.1 Vorstellung der GUI

Bevor mit den eigentlichen Demonstrationen und Eingabetests begonnen werden kann, muss zuerst die für diese Zwecke entwickelte GUI gezeigt werden. Diese wurde zwar explizit nicht für Endbenutzer und Endbenutzerinnen entwickelt, kommt dem für ein Endprodukt gewünschten Input-Output Verhalten schon nahe.

Wie das Formular vor der ersten Suche aussieht wird in Abbildung 4.17 gezeigt. Zuerst muss der entsprechende Themenbereich (Channel) gewählt werden (siehe Abbildung 4.18). Sobald der ersten Buchstabe eingegeben wurde, ist bereits eine Vorschlagsliste mit den Such-Ergebnissen vorhanden. In Abbildung 4.19 ist neben dieser außerdem die benötigte Suchzeit zu finden. Für jeden zusätzlich eingegebenen Buchstaben wird eine neue Suche gestartet und die Ergebnisse in Echtzeit aktualisiert (siehe Abbildung 4.20 und Abbildung 4.21).

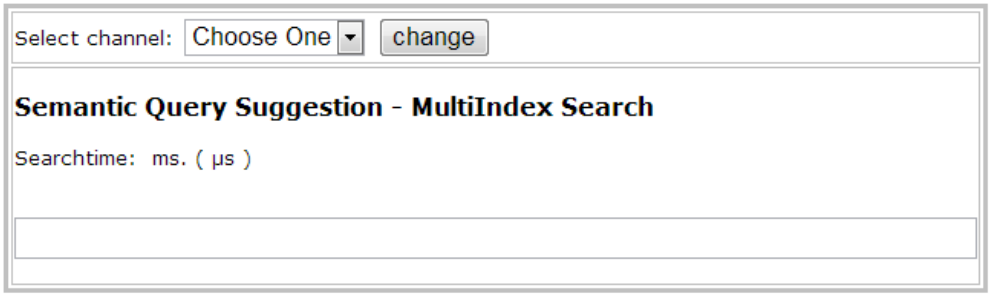


Abbildung 4.17: Vorstellung der GUI - Schritt 1

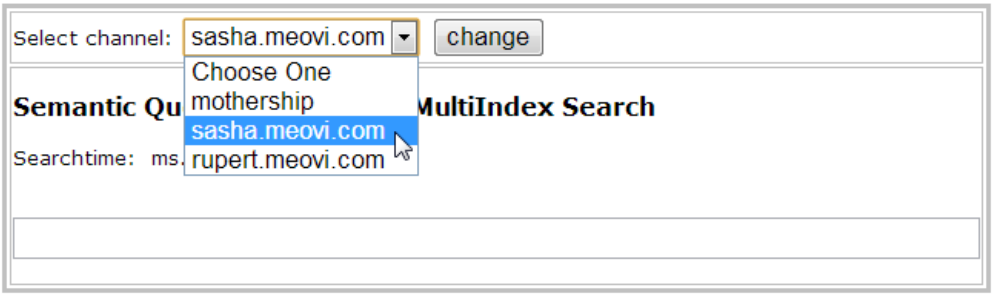


Abbildung 4.18: Vorstellung der GUI - Schritt 2

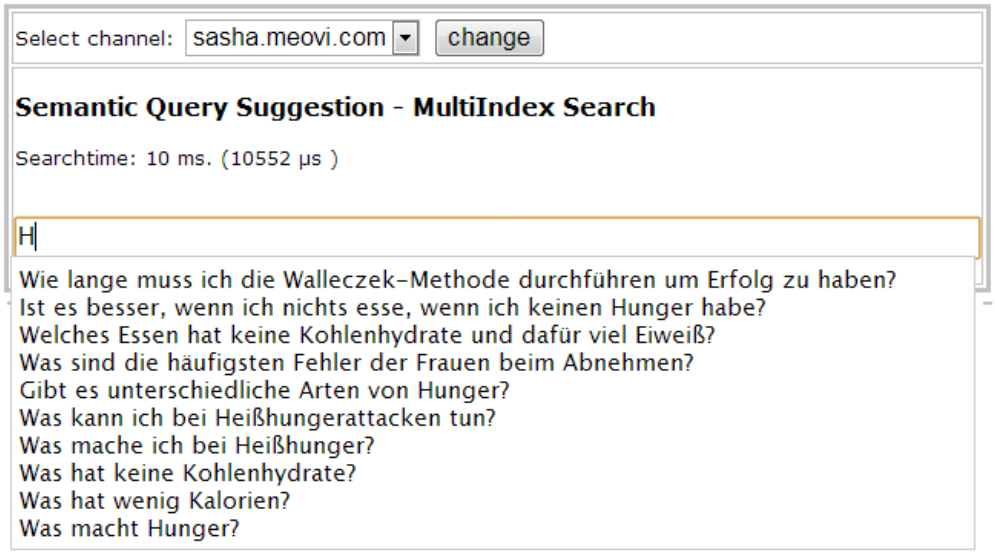


Abbildung 4.19: Vorstellung der GUI - Schritt 3

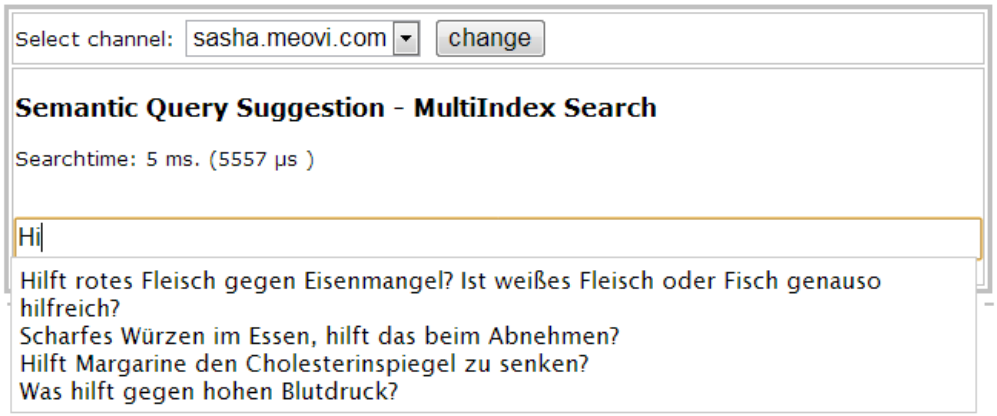


Abbildung 4.20: Vorstellung der GUI - Schritt 4

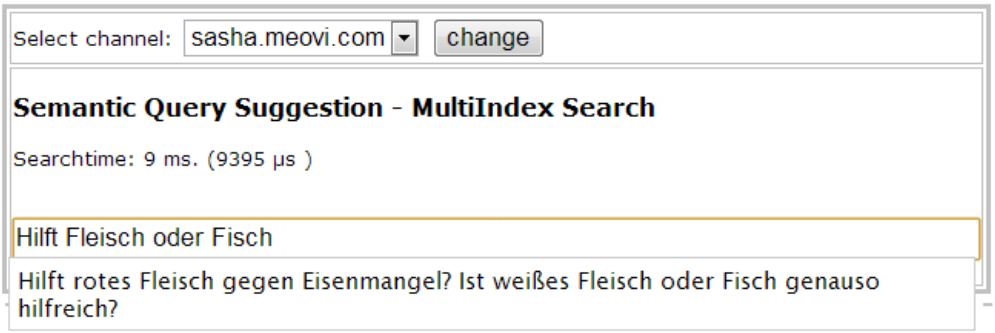


Abbildung 4.21: Vorstellung der GUI - Schritt 5

4.4.2 Demonstration von Stärken der Implementierung

Die Funktionen der Suche werden innerhalb dieses Unterkapitels anhand von drei Beispielen gezeigt. Die komplette Suche besteht aus einem semantischen und einem syntaktischen Teil. Um zeigen zu können, wie die Ergebnisse zustande kommen müssen die Komponenten zuerst separat getestet werden. In den Demonstrationen lässt sich leicht erkennen wo die Stärken der Teil-Suchen liegen und wieso beide für ein optimales Gesamtergebnis benötigt werden.

Da es nicht sinnvoll wäre einen Screenshot nach jedem eingegebenem Buchstaben zu machen, sind auf den nun folgenden Seiten einige repräsentative Zwischenergebnisse der Testserien zu sehen.

Syntaktische Suche

Mit der syntaktischen Suche erhält man bereits ab dem ersten eingegebenen Buchstaben ein Ergebnis, das mit jedem weiteren Buchstaben bzw. Wort ständig präziser wird. Dieser Vorteil lässt sich gleich erkennen, wenn man Abbildung 4.22 und Abbildung 4.24 gegenüberstellt.

Da der Datensatz für die Demonstrationen aus dem Bereich Ernährung kommt, beginnen viele Fragen mit dem String "`Wie viel`". Wie in Abbildung 4.22 zu sehen ist, liefert eine syntaktische Suche mit dieser Phrase allerdings nicht nur diese Fragen zurück, sondern auch jene, in denen beide Worte an irgendeiner beliebigen Stelle vorkommen. Abbildung 4.23 zeigt, dass durch das Erweitern der Suchphrase um "`Obst`" nur mehr ein Eintrag gefunden werden kann, in dem die nunmehr drei Worte vorkommen. Da jeder weitere Buchstabe die Ergebnismenge nur verkleinern könnte, kann die Demonstration an dieser Stelle schon erfolgreich abgebrochen werden.

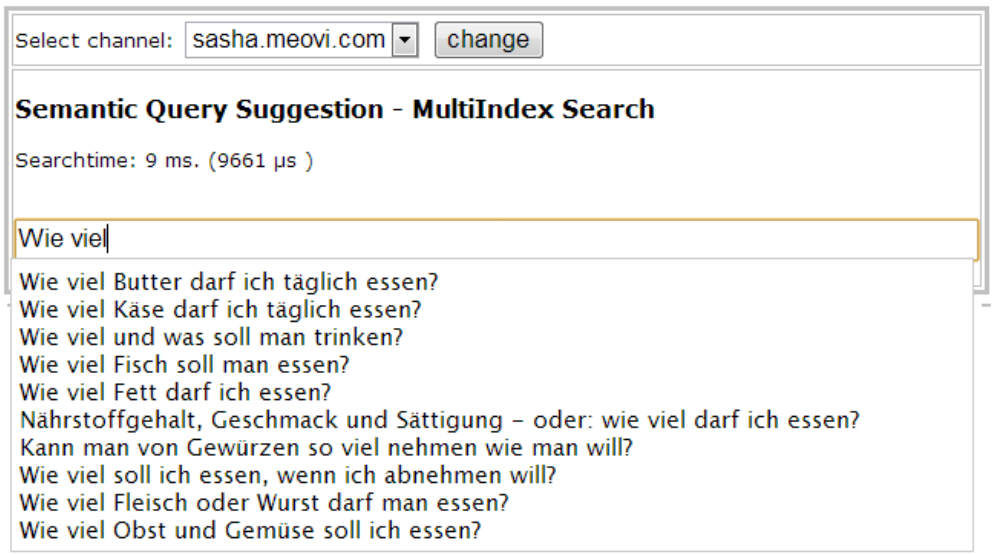


Abbildung 4.22: Demonstration Syntaktische Suche 1

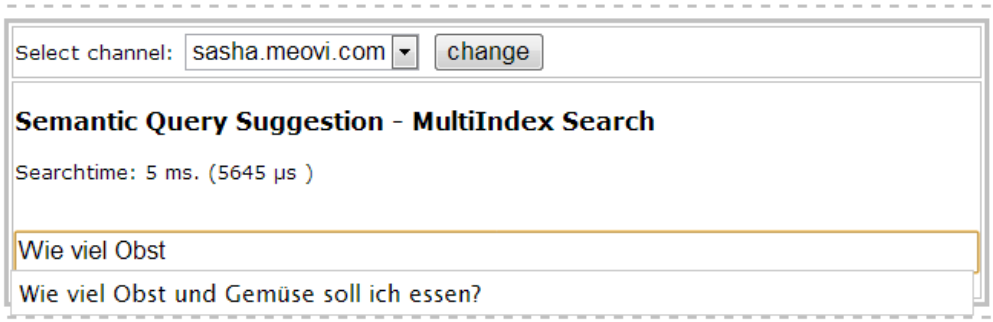


Abbildung 4.23: Demonstration Syntaktische Suche 2

Semantische Suche

Die semantische Suche benötigt Schlüsselwörter um arbeiten zu können. Da im String "`Wie viel`" allerdings keines enthalten ist, muss in diesem Fall die Suche nicht gestartet werden. In der Folge erhält man auch in der Testumgebung für diese Eingabe kein Ergebnis, wie in Abbildung 4.24 zu sehen ist. Sobald nun das erste Schlüsselwort erkannt wurde (siehe Abbildung 4.25) kann die Suche durchgeführt werden. Abbildung 4.26 zeigt, dass die Eingabe eines weiteren Buchstaben am Ergebnis gar nichts ändern kann, da immer noch das mit dem einen Keyword gesucht wird.

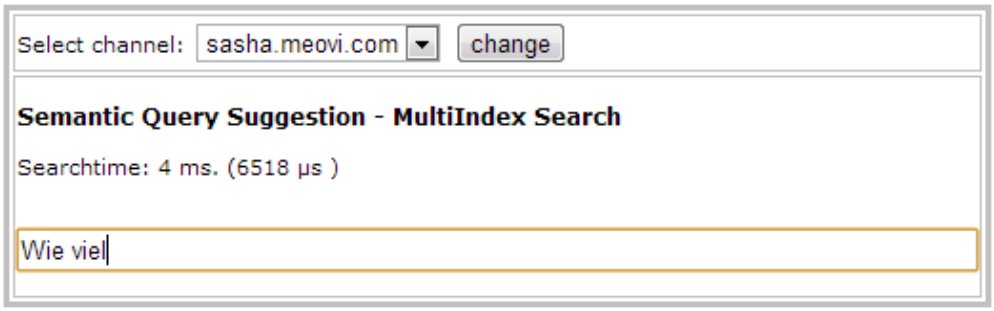


Abbildung 4.24: Demonstration Semantische Suche 1

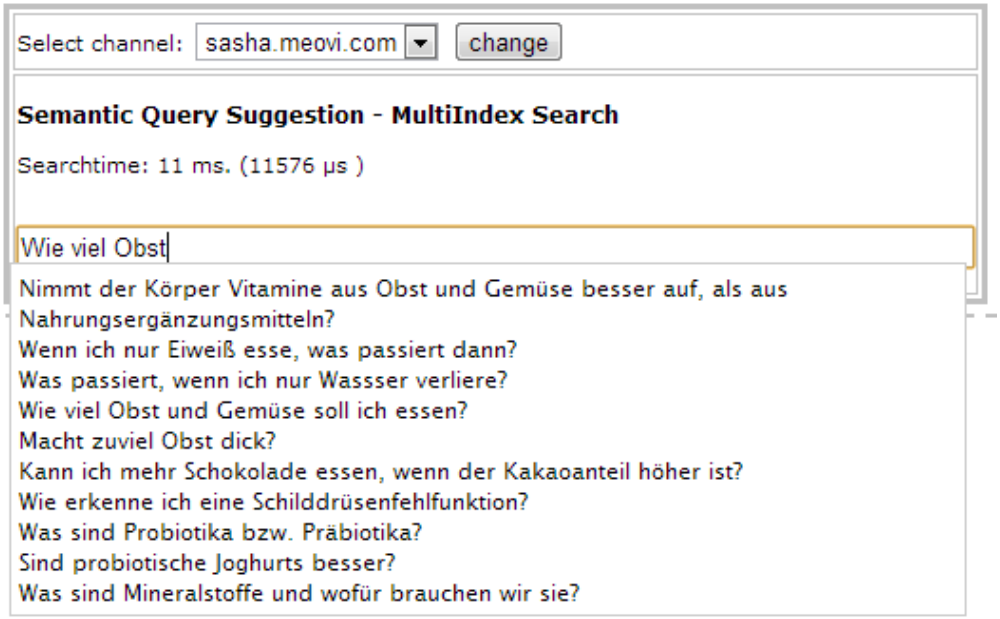


Abbildung 4.25: Demonstration Semantische Suche 2

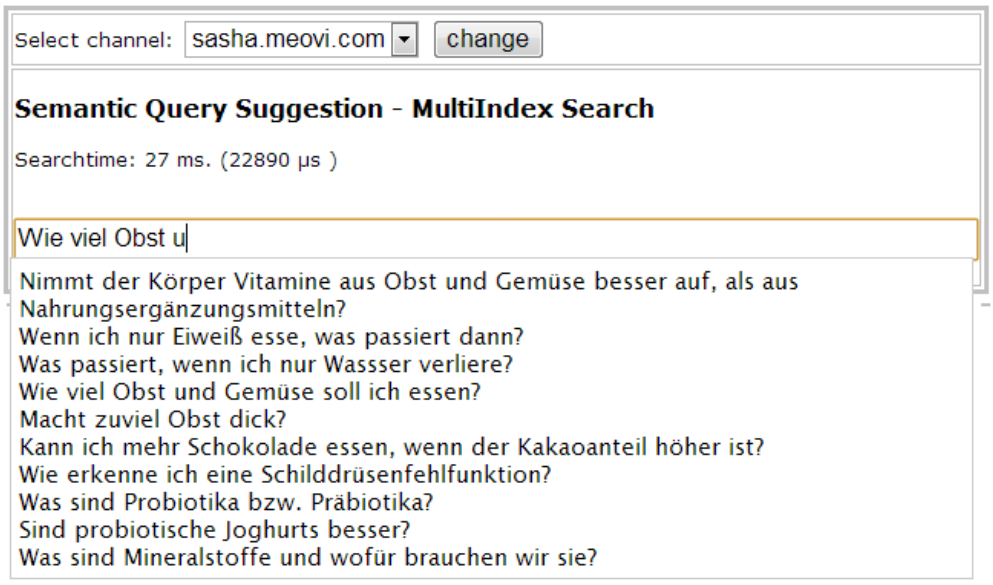


Abbildung 4.26: Demonstration Semantische Suche 3

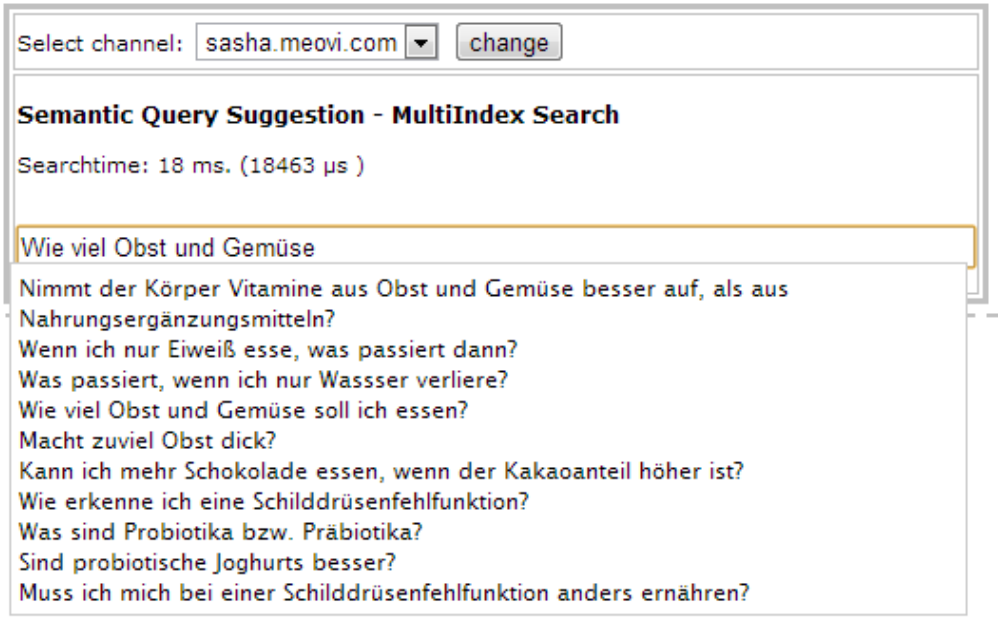


Abbildung 4.27: Demonstration Semantische Suche 4

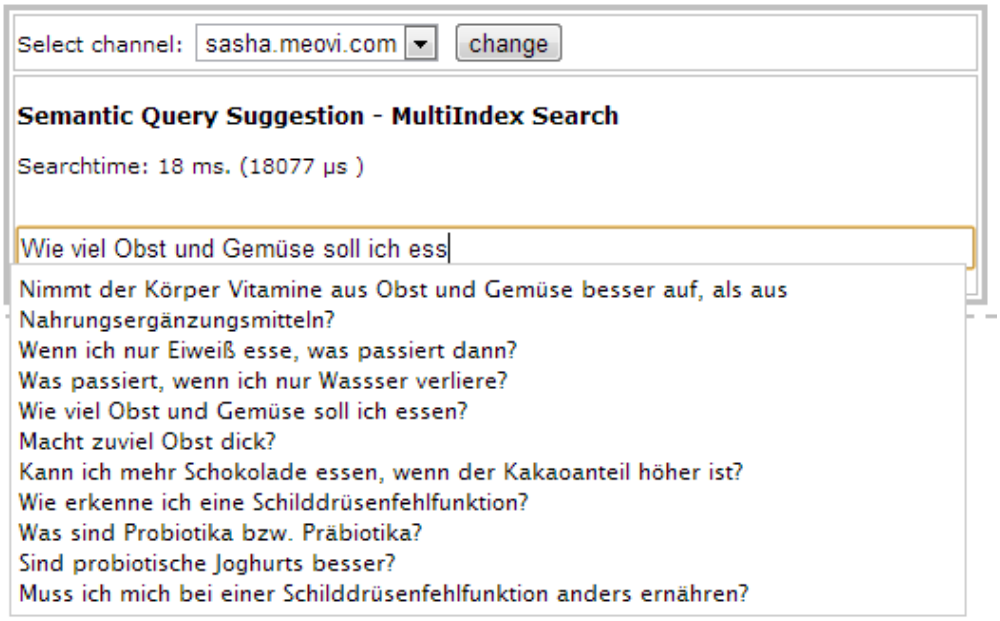


Abbildung 4.28: Demonstration Semantische Suche 5

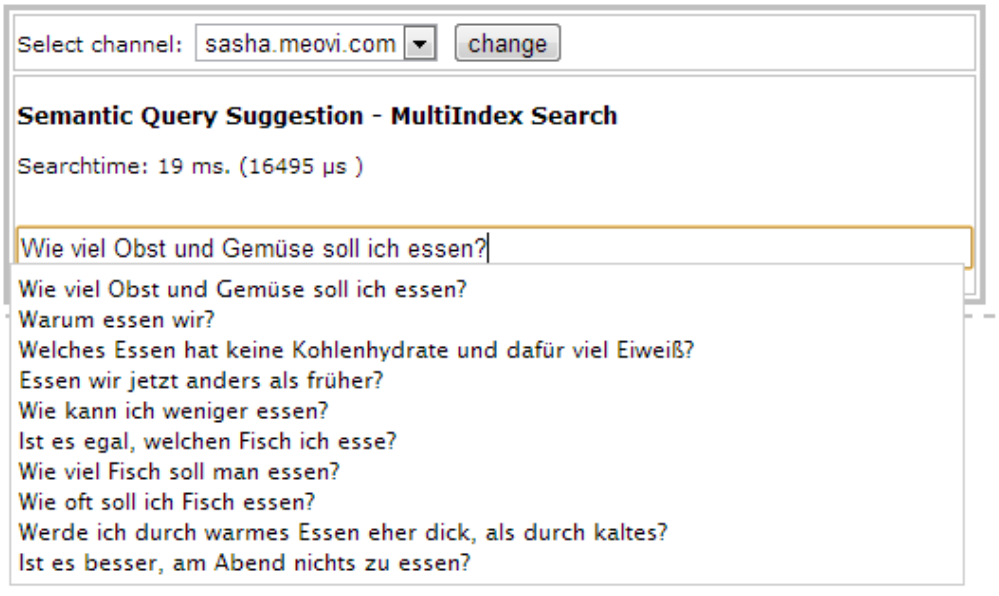


Abbildung 4.29: Demonstration Semantische Suche 6

Komplette Suche

Nachdem jetzt die beiden Suchen getrennt voneinander demonstriert wurden, werden in diesem Unterkapitel die Vorteile aufgezeigt, die durch die Kombination zustande kommen. In den vorhergehenden Unterkapiteln wurde die Funktionalität der semantischen und syntaktischen Suche jeweils getrennt voneinander demonstriert. Dieser Teil der Demonstration beschäftigt sich mit der kompletten Suche. Aus Gründen der Vergleichbarkeit wurde natürlich dieselbe Frage als Eingabestring verwendet. Bei der Betrachtung von Abbildung 4.30 und einem Rückblick auf die Tests der Teilsuchen mit dem Teilsatz (siehe Abbildung 4.24 und Abbildung 4.22) fällt gleich auf, dass hier die syntaktische Suche für die Ergebnisse sorgt. Vergleicht man das Ergebnis von Abbildung 4.31 mit Abbildung 4.27 zeigt sich, dass hier der syntaktische Teil dafür gesorgt hat, dass die „richtige“ Frage an den Anfang gereiht wurde.

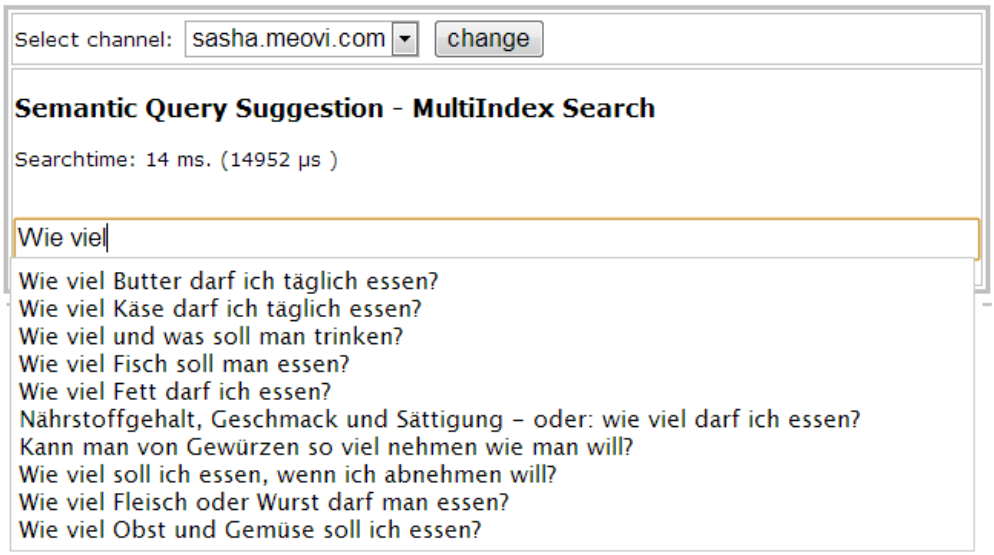


Abbildung 4.30: Demonstration von Stärken der Implementierung 1

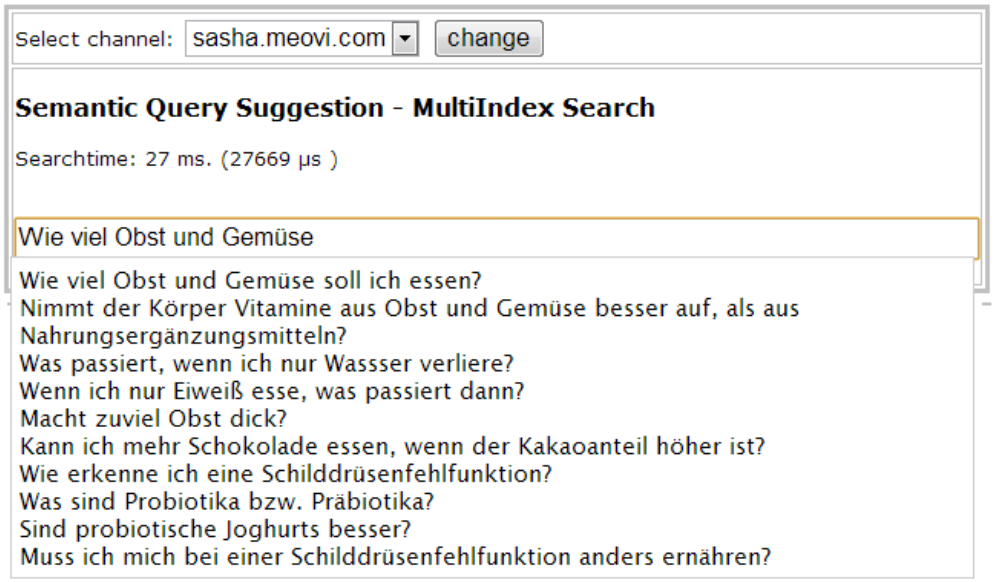


Abbildung 4.31: Demonstration von Stärken der Implementierung 2

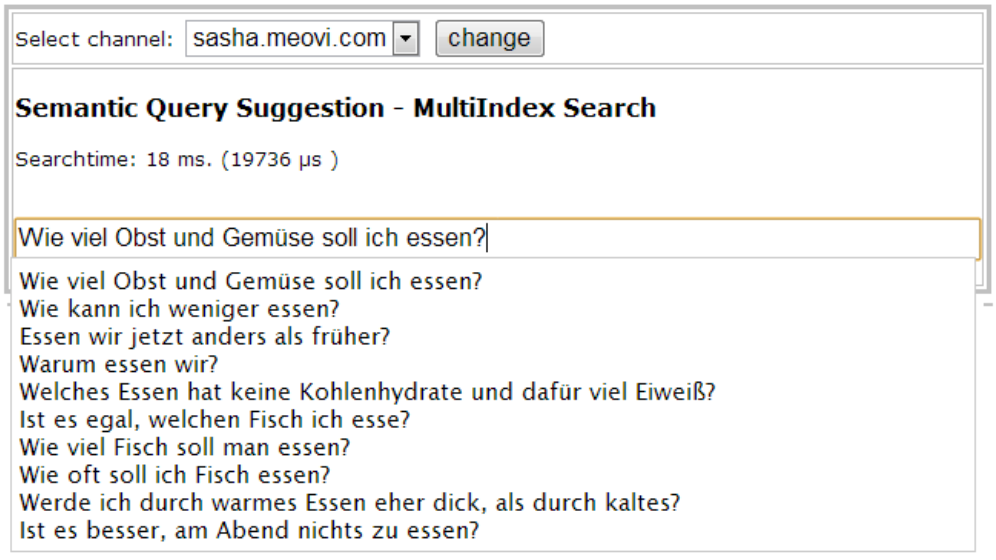


Abbildung 4.32: Demonstration von Stärken der Implementierung 3

4.4.3 Demonstration von Vor- und Nachteilen der Implementierung

In diesem Unterkapitel soll an Beispielen gezeigt werden warum sowohl der semantische als auch der syntaktische Teil eine wichtige Rolle für die komplette Suche spielen. Die Stärken und Schwächen der beiden Suchen werden anhand von „idealen“ Beispielen gezeigt. Situationen in denen die aktuelle Implementierung dennoch versagt und welche Verbesserungen in zukünftigen Versionen vorgenommen werden können werden in Abschnitt 4.5.2 besprochen.

Der String „`Soll ich eine Menge Obst und Gemüse essen?`“, welcher für die erste Demonstration gewählt wurde, beinhaltet die gleichen Schlüsselwörter, die auch in der gesuchten Frage („`Soll ich eine Menge Obst und Gemüse essen?`“)vorkommen. Da allerdings die anderen Worte großteils vom Gesuchten abweichen ist die syntaktische Suche, wie in Abbildung 4.33 zu sehen ist, nicht erfolgreich. Bei der semantischen Suche, die nur mit den Keywords `Obst`, `Gemüse` und `essen` durchgeführt wird, sieht das Ergebnis allerdings vielversprechend aus (siehe Abbildung 4.34).

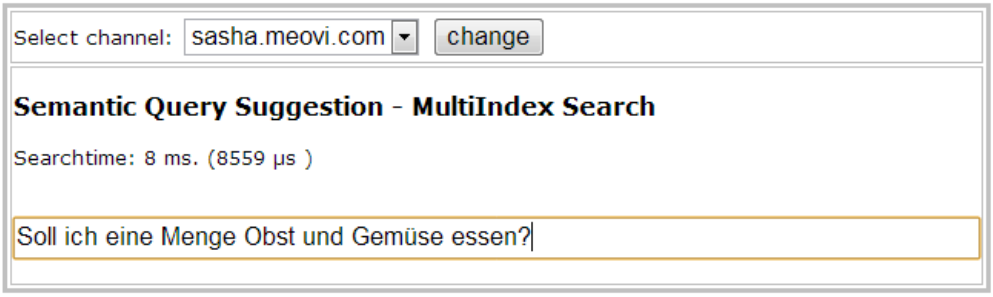


Abbildung 4.33: Demonstration Syntaktische Suche - schlechtes Ergebnis

Um eine Situation zu zeigen, in der die syntaktische Suche ein besseres Ergebnis liefert als der semantische Teil, wurde ein Tippfehler in die Eingabe eingebaut. Aus dem Wort `Gemüse` wird `Gmüse`. Da letzteres nicht mehr als Schlüsselwort erkannt wird kann die semantische Suche nur mit `Obst` und `essen` arbeiten. Die Ergebnismenge enthält zwar die gesuchte Frage, allerdings nicht an der ersten Position (siehe Abbildung 4.35). Für die syntaktische Suche stellt ein „vergessener“ Buchstabe kein Problem dar, wie in Abbildung 4.36 zu erkennen ist.

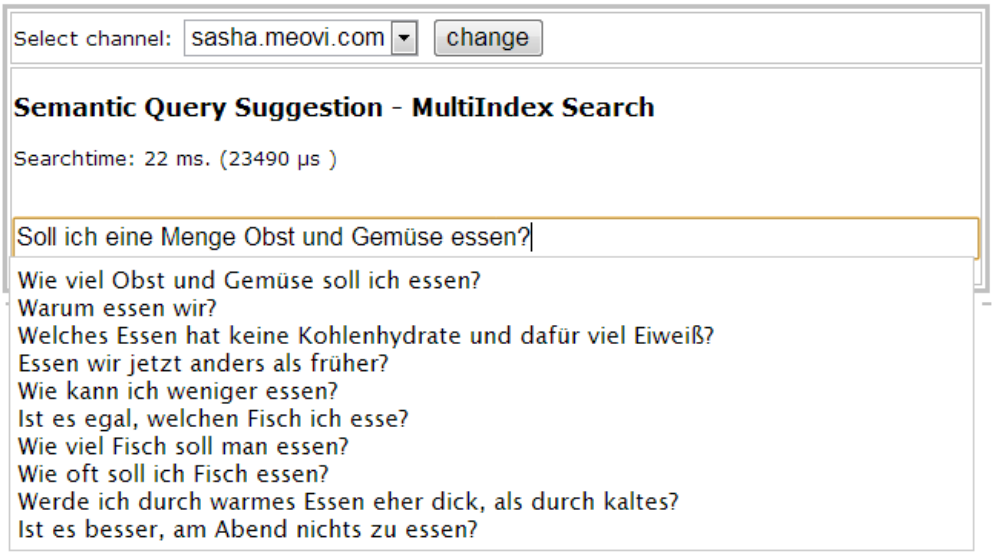


Abbildung 4.34: Demonstration Semantischen Suche - gutes Ergebnis

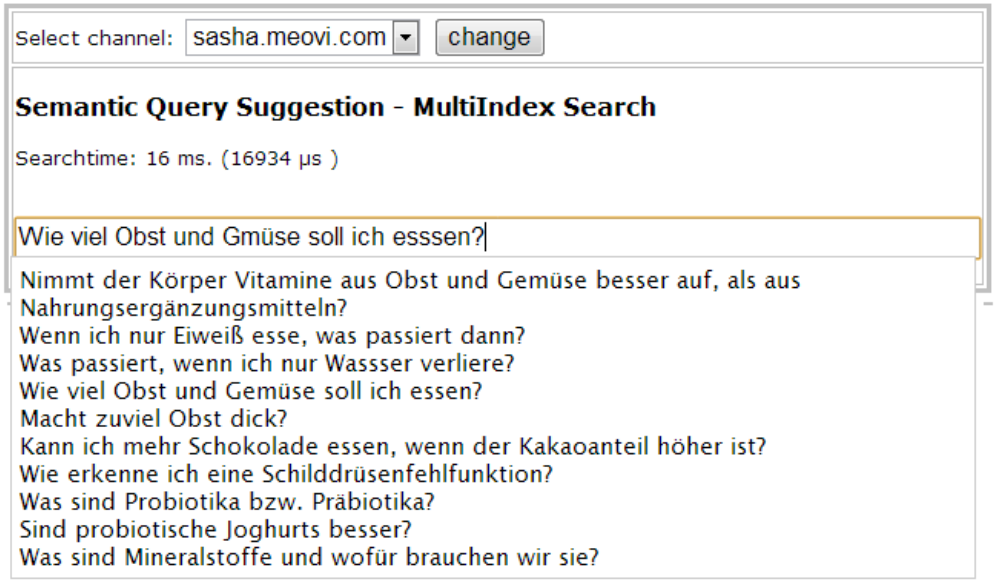


Abbildung 4.35: Demonstration Semantischen Suche - schlechtes Ergebnis

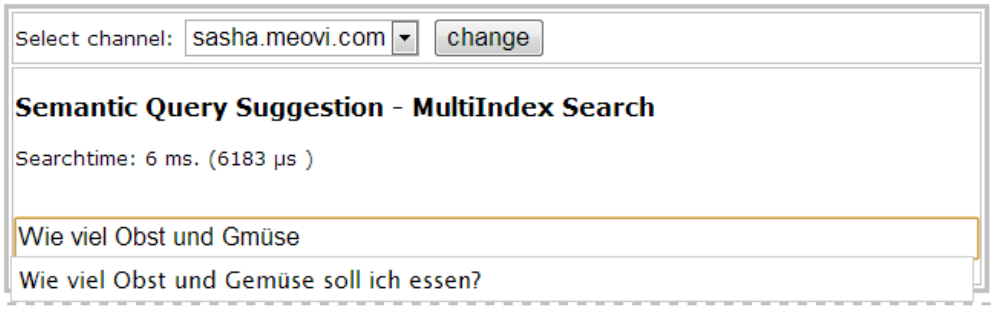


Abbildung 4.36: Demonstration Syntaktische Suche - gutes Ergebnis

4.4.4 Performance Vergleiche

In diesem Kapitel werden Matching Algorithmen direkt aufgerufen um die Ergebnisse anschließend mit denen einer vorberechneten Variante vergleichen zu können. Da es schwierig wäre die nativen Ansätze außerhalb des Systems zu testen, werden alle Vergleiche innerhalb des Gesamtprodukts durchgeführt. Daraus resultiert natürlich ein gewisser Overhead beim Aufruf und der Ausführung, der allerdings sämtliche Testaufbauten gleichermaßen betrifft und daher das Ergebnis nicht verfälscht. Für diesen Performance-Vergleich wurden die zwei semantischen Algorithmen `SemanticTopicTreeMatcher` und `CorrelationBasedMatcher` gewählt, die sich in Qualität und benötigter Rechenzeit stark unterscheiden. Ersterer wurde verwendet um den Index zu erstellen der den Vorteil dieser Vorberechnungen zeigen soll. Alle drei Varianten wurden mit Queries, die eine unterschiedliche Anzahl an Wörtern (Schlüsselwörtern) hatten, getestet. Die folgenden Tabellen zeigen das arithmetische Mittel, minimale und maximale Werte der Laufzeit, die über jeweils 100 Testläufe gesammelt wurden.

Tabelle 4.27: Durchschnitt der gemessenen Zeiten

Query	SemanticTopicTreeMatcher	CorrelationBasedMatcher	Indexed
essen	24315 ms	1453 ms	4 ms
essen eiweiß	107791 ms	1424 ms	2 ms
essen eiweiß abend	161399 ms	1288 ms	3 ms
hunger	46433 ms	1381 ms	1 ms
hunger satt	61637 ms	1306 ms	2 ms
hunger satt faustformel	109574 ms	1311 ms	1 ms
essen eiweiß abend hunger	198658 ms	1320 ms	2 ms
essen eiweiß abend hunger satt	190172 ms	1434 ms	2 ms
essen eiweiß abend hunger satt faustformel	237599 ms	1401 ms	2 ms

Tabelle 4.27 zeigt, dass beide Matcher im Durchschnitt für eine Berechnung eine oder gar mehrere Sekunden benötigen und daher wie eingangs erwähnt für den Einsatz in einer Echt-

zeitanwendung ungeeignet sind. Die Laufzeit beim `CorrelationBasedMatcher` bleibt in den 9 Testfällen unabhängig vom Input relativ gleich. Die Zeit, die ein Matching mit Hilfe des `SemanticTopicTreeMatcher` benötigt, ist nicht nur um einiges höher, sondern auch stark vom Input abhängig. Während ein Test mit dem Input „essen“ 24 Sekunden in Anspruch nimmt, benötigt derselbe Algorithmus für die Eingabe „hunger“ beinahe doppelt so lang. In der letzten Spalte finden sich die Zeiten der Suche, welche die vorberechneten Indices verwendet. Unabhängig von der Eingabe, und deren Länge, kann dieser Algorithmus in weniger als $5ms$ das gewünschte Ergebnis liefern.

Tabelle 4.28: Minimal & Maximal gemessene Zeiten

Query	SemanticTopicTreeMatcher		CorrelationBasedMatcher		Indexed	
	min	max	min	max	min	max
essen	24131 ms	24566 ms	1274 ms	2406 ms	2 ms	31 ms
essen eiweiß	105019 ms	111203 ms	1236 ms	1887 ms	2 ms	23 ms
essen eiweiß abend	157707 ms	164531 ms	1216 ms	1453 ms	2 ms	25 ms
hunger	41953 ms	53560 ms	1304 ms	2267 ms	1 ms	2 ms
hunger satt	57080 ms	68557 ms	1248 ms	1519 ms	1 ms	25 ms
hunger satt faustformel	105387 ms	117942 ms	1266 ms	1517 ms	1 ms	24 ms
essen eiweiß abend hunger	177413 ms	215649 ms	1257 ms	1399 ms	1 ms	19 ms
essen eiweiß abend hunger satt	184756 ms	201298 ms	1203 ms	1580 ms	1 ms	23 ms
essen eiweiß abend hunger satt faustformel	228901 ms	254390 ms	1194 ms	1561 ms	1 ms	19 ms

4.5 Zukünftige Optimierungen

4.5.1 Syntaktische Suche

Syntaktische Suche versagt

Im Kapitel rund um die syntaktische Suche (siehe Unterabschnitt 4.3.2) wurde bereits erwähnt, dass auch nur kleine Abweichungen (anderes Bindewort, Tippfehler in kurzen Worten) ein enormes Problem darstellen. Da diese Komponente ursprünglich nur als Unterstützung dienen sollte, wurde dies in Kauf genommen. Es hat sich allerdings gezeigt, dass auch die semantische Suche unter gewissen Umständen schlechte Ergebnisse liefern kann.

Zwei Probleme, die einer syntaktischen Suche schnell zum Verhängnis werden können, sind Stoppwörter und „falsche“ Fuzzy-Werte. Die Behebung des ersten Dilemmas ist in der Theorie relativ trivial. Die beste Lösung ist es, je nach Länge der Eingabe alle bzw. gewisse Füllwörter aus der Suche zu entfernen. Um die Suchzeit unter einem gewissen Wert zu halten wurde der Fuzzy-Faktor deutlich eingeschränkt. Da die Fuzzy Suche in Lucene 4.0 verbessert wurde (siehe [11]) können diese Beschränkungen nochmal überdacht werden. Es wäre beispielsweise möglich, den Prozess mit unterschiedlichen Werten parallel mehrmals zu starten und anschließend das beste Ergebnis zu wählen (siehe Abbildung 4.37).

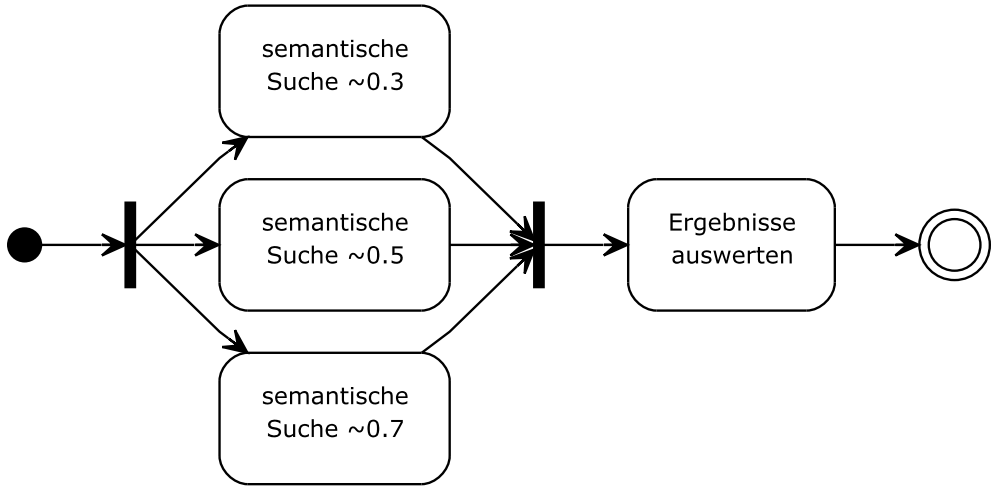


Abbildung 4.37: Parallelisierte syntaktische Suche

Ein weiterer Ansatz, der mit Lucene 4.0 getestet werden müsste, kann mit beiden Problemen in gewisser Weise gleichzeitig umgehen. Anstatt konjunktiv verknüpften Fuzzy Terms wäre es denkbar diese disjunktiv zu verbinden, allerdings mit der Auflage, dass eine gewisse Anzahl (bzw. ein Prozentsatz) erfüllt sein muss.

4.5.2 Sematische Suche

Score Normalisierung über mehrere Themenbereiche

Da für jeden Channel (der einen anderen Themenbereich darstellt) andere semantische Algorithmen für die Vorberechnung der semantischen Treffer verwendet werden können, ist eine Suche im gesamten System nicht so einfach möglich. Es müssen erst Möglichkeiten gefunden werden um die unterschiedlichen Scorings auf einen gemeinsamen Nenner zu bringen.

Der einfachste Ansatz ist, wenn alle Berechnungen abgeschlossen sind, die errechneten Werte zu analysieren und einen Maximalwert zu finden. Anschließend wird jeder Score durch diesen Wert dividiert und das Ergebnis überschreibt den ursprünglichen Eintrag. Da die Vorberechnungen der semantischen Matches in Leerlaufzeiten des Servers oder auf anderen Computern durchgeführt werden, hat dieser Ansatz der Normalisierung keinen Einfluss auf die Suche.

Die zweite Variante sucht beim Laden des jeweiligen Kanals den zuvor erwähnten Höchstwert. Dieser kann in der Folge bei einer Suche verwendet werden um die erhaltenen Scores on-the-fly zu normalisieren. Da hier die Berechnungen zur Suchzeit stattfinden ist mit Einbußen in der Laufzeit zu rechnen. Ob diese ins Gewicht fallen, müsste gegebenenfalls untersucht werden.

Eine dritte Möglichkeit ist es, dieselben Testdatensätze durch alle Matching Algorithmen laufen zu lassen und anhand der Unterschiede Normalisierungsfunktionen zu entwickeln. Diese könnten in Folge entweder on-the-fly bei der Suche oder auch schon anschließend an die Vorberechnung eingesetzt werden.

Neubewertung der Scores

Während sich die Normalisierung mit dem Problem der globalen Suche beschäftigt, geht es bei der Neubewertung um ein Problem, das aus der unvollständigen Berechnung der möglichen Schlüsselwort-Kombinationen entsteht. Wie in Unterabschnitt 4.3.4 erläutert wurde, durchforstet eine Suche mit n Keywords alle Vorberechnungen, die aus n oder weniger Elementen bestehen. Ein Treffer, der die Query zu 100% erfüllt, hat den „richtigen“ Score hinterlegt und könnte daher 1:1 übernommen werden. Bei einem Treffer in der Menge, die Kombinationen aus $n - m$ Schlüsselwörtern enthält, waren diese m Elemente zur Berechnungszeit nicht „bekannt“ und daher können diese Punkte nicht einfach ins Ergebnis übernommen werden. Die einfachste Lösung wäre es daher beispielsweise, diese Multiplikatoren für die einzelnen Kombinationslängen festzulegen.

Wenn der Scoring-Algorithmus nicht-lineare Werte erzeugt und daher diese Multiplikatoren durch eine komplexere Funktion zu ersetzen sind, müssen entsprechende Informationen vorliegen. Entweder ist etwas über das verwendete Scoring bekannt, oder es wird anhand einiger Tests analysiert. Letztere Lösung muss zwar berechnet werden, hat allerdings den Vorteil kein Wissen über das Matching vorauszusetzen.

Optimierung der iterativen Suche

Die Implementierung iteriert die Suche über die Vorberechnungen mit absteigender Kombinationslänge solange durch bis die gewünschte Anzahl an Ergebnissen gefunden wurde. Allerdings

könnte es vorkommen, dass die Ergebnismenge voll ist obwohl nachfolgende Suchen Ergebnisse mit einem höheren Score gefunden hätten. Da dies trotz der Neubewertung der Punkte durchaus möglich ist, kann somit keine optimale Lösung garantiert werden. Es gibt hier auch zwei Varianten das Problem zu lösen.

Die erste Möglichkeit setzt beim Erstellen des Indexes an. Ist die Variante der Neubewertung bekannt, könnte dies durch ein Einschränken der Score-Grenzwerte beim Vorberechnen sichergestellt werden. Jedes Element mit Kombinationslänge n muss bei dieser Vorgehensweise bei einer Suche einen höheren Score liefern als alle vorberechneten Einträge mit weniger Schlüsselworten. Dadurch wäre sichergestellt, dass das Ergebnis in Bezug auf die vorhandenen Vorberechnungen optimal ist.

Der bessere Ansatz wäre es daher, eine parallelisierte Suche zu realisieren, die alle Vorberechnungsmengen gleichzeitig durchsucht und daher alle möglichen Einträge berücksichtigen kann. Ein möglicher weiterer Vorteil dieser Idee wäre eine Verkürzung der Laufzeit, wenn mehr als eine Datenstruktur durchsucht wird. Die Kosten für das zusammenführen der Ergebnismengen liegen aller Wahrscheinlichkeit nach unter dem Gewinn, der erzielt wird, wenn nur zwei Abfragen analog ausgeführt werden.

Vom Benutzer und der Benutzerin lernen

Durch die Verwendung komplexer Matching Algorithmen steigt neben der Qualität der Ergebnisse auch die Rechenzeit die für einen Vergleich aufgewendet werden muss. Trotz der Reduktion theoretisch möglicher Kombinationen von Schlüsselwörtern auf eine wesentlich kleinere Menge kann diese immer noch groß genug sein, sodass der Index erst nach Stunden oder Tagen fertig gestellt werden kann. Alternativ kommt man mit zu restriktiven Parametern in den Matchern zu einem nutzlos kleinen Suchindex. Außerdem werden zahlreiche Keyword Kompositionen gebildet, die ein menschlicher Benutzer oder eine Benutzerin niemals eingeben würde. Jede neue Frage, die in das ständig wachsende System eingepflegt wird, hat einen Einfluss auf diverse semantische Ähnlichkeiten. Daher müssen nach einer Änderung sämtliche Berechnungen neu durchgeführt werden.

Um diesem Problem etwas entgegenzusetzen kam die Idee auf, von den Eingaben des Benutzers oder der Benutzerin zu lernen. Es wird daher nur ein minimaler Index gebaut, der nur Matches von Kombinationen mit der Länge 1 (die reinen Schlüsselwörter) enthält. Jede vom Benutzer oder der Benutzerin eingegebene Schlüsselwortkombination wird, wenn im Index der entsprechenden Länge nichts gefunden wurde, für die spätere Berechnung abgelegt. Bei einem Missmatch gibt es allerdings zwei Möglichkeiten. Es kann ein extrem schneller (und qualitativ schlechter) Matcher verwendet werden, der mit Ergebnissen des vorhandenen Minimalindex aufgebessert wird. Wenn die Scores der Kombination einzelner Schlüsselwort-Fragen Matches, die im Minimalindexes abgelegt sind, für die Rückgabe akzeptable Werte liefern, kann auch diese Möglichkeit verwendet werden. Welche Variante besser geeignet ist müsste bei einer Implementierung untersucht werden. Wenn im System Rechenzeit frei ist, werden die gemerkten Kombinationen aufgearbeitet und im entsprechenden Index abgelegt. Bei der Suche fällt das Iterieren über alle Indices der verschiedenen Längen weg, denn wenn eine Kombination berechnet wurde sind alle Ergebnisse vollständig abgelegt. Auch eine Neuberechnung des kompletten

Indexes stellt kein Problem dar, da einfach alle Kombinationen, die von Benutzereingaben gesammelt wurden, neu berechnet werden.

Tippfehler und Wörterbücher

In Unterabschnitt 4.4.3 war ein Beispiel für eine mögliche Eingabe zu sehen, welche ein Problem für die semantische Suche darstellt. Durch einen vergessenen Buchstaben in einem wichtigen Schlüsselwort wird dieses nicht als solches erkannt. Die gezeigte Demonstration kann aber die komplette Suche durch eine erfolgreiche syntaktische Suche retten. Dieser Umstand ist allerdings einer idealen Eingabe zu verdanken.

Es drängt sich also die Frage auf, wie die semantische Suche in zukünftigen Versionen verbessert werden kann, um auch mit suboptimalen Eingaben umgehen zu können. Ein möglicher Ansatz, um dem Problem zu begegnen, der aber im Zuge des Projekts nicht mehr realisiert werden konnte, wäre die Einführung eines Wörterbuchs. Die Suche könnte, wie in Abbildung 4.38 als Beispiel zu sehen ist, die bisherige Suche parallel zur einer neuen Variante ausführen. Letztere könnte eindeutige Nicht-Schlüsselwörter aussortieren und den Rest analysieren. Für jedes nicht erkannte Wort wird eine Fuzzy-Suche im Wörterbuch durchgeführt. Die Suche im vorberechneten Index kann anschließend mit den erkannten und gefundenen Keywords durchgeführt werden. Die Unschärfe beim Nachschlagen nicht erkannter Wörter kann zu mehr als einem Ergebnis führen. Aus dem Grund muss die anschließende Suche auch mit diesem Problem umgehen können. Nachdem der bisherige und der neue Prozess abgeschlossen sind müssen die Ergebnisse in einem letzten Schritt bewertet und zusammengeführt werden.

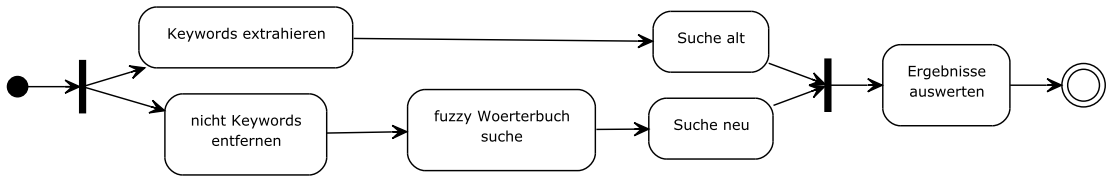


Abbildung 4.38: Suche mit Wörterbuch

KAPITEL 5

Zusammenfassung

Das Ziel dieser Arbeit war es, die Matching Algorithmen eines Frage-Antwort-Systems, die dafür verwendet werden, ähnliche Fragen zu finden, in Echtzeit verwendbar zu machen ohne diese Berechnungen selbst zu optimieren.

Die zentrale Komponente des Lösungsansatzes, der hier verfolgt wurde, ist eine Vorberechnung von Matches. Nachdem die Algorithmen eine bekannte Menge an Schlüsselworten verwenden um Ähnlichkeiten zu berechnen, kann man den Input eines Matches auf die Kombination von Schlüsselworten reduzieren. Da die Anzahl der theoretisch möglichen Zusammensetzungen abhängig von den dem System bekannten Keywords schnell zu groß wird um alle in die Vorberechnung aufzunehmen, musste eine Optimierung gefunden werden. Die Lösung für dieses Problem waren „Prematches“. Da nur Schlüsselwörter miteinander verknüpft wurden, die auf ähnliche Fragen zeigen, konnte die Menge der zu berechnenden Matches drastisch reduziert werden.

Nachdem die semantische Suche auf Schlüsselwort-Kombination basiert, wurde diese durch eine syntaktische Suche ergänzt. Durch diese Erweiterung können für Tippfehler und kurze Eingabestrings bessere Ergebnisse erzielt werden.

In den Performance-Vergleichen wurde ersichtlich, dass aus Sekunden, die verwendete Matcher benötigen, wenige Millisekunden wurden. Das Ziel aus Anwendungssicht, direkt während des Eintippens einer neuen Frage, die Auswahlbox mit den syntaktisch und semantisch ähnlichsten bisher gestellten Fragen bei jedem Tastendruck entsprechend performant anzupassen und einzuschränken, konnte sehr gut erreicht werden. Da die Qualität der Ergebnisse allerdings von diversen Parametern bei der Berechnung des Indexes abhängt, gibt es an dieser Stelle sicher noch Möglichkeiten zur Optimierung.

Literaturverzeichnis

[1] John Bohannon. Searching for the google effect on people's memory. *Science*, 333(6040):277, 7 2011.

[2] Robin Burke, Kristian Hammond, Vladimir Kulyukin, Steven Lytinen, Noriko Tomuro, and Scott Schoenberg. Natural language processing in the faq finder system: Results and prospects. In *In Working Notes, AAAI spring symposium on NLP on the WWW , Menlo Park, CA*, page 26. AAAI Press, 1997.

[3] D. Knuth, J. Morris, Jr., and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[4] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[5] Manu Konchady. *Text Mining Application Programming*. Charles River Media, 1 edition, May 2006.

[6] Andreea-Hilda Kosorus. *Learning-oriented Question Recommendation*. Dissertation, 7 2013.

[7] Andreea-Hilda Kosorus, Andreas Bögl, and Josef Küng. Semantic similarity between queries in qa system using a domain-specific taxonomy. In *Proceedings of the 14th International Conference on Enterprise Information Systems ICEIS 2012*, volume Volume 1 of *ICEIS*, pages 241–246. SciTePress, 6 2012.

[8] Andreea-Hilda Kosorus and Josef Küng. Learning-oriented question recommendation using bloom's taxonomy and variable length hidden markov models. In *International Conference on Advanced Computing and Applications*, 10 2013.

[9] Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10(8):707–710, 1966. Original in *Doklady Akademii Nauk SSSR* 163(4): 845–848 (1965).

[10] Steven L. Lytinen and Noriko Tomuro. The use of question types to match questions in faqfinder. In *In Proc. of AAAI'02*, 2002.

[11] Michael McCandless. Lucene's fuzzyquery is 100 times faster in 4.0, 2011.

[12] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Manning Publications, 2010.

[13] Stanley J. Mlynarczyk and Steven L. Lytinen. Faqfinder question answering improvements using question/answer matching, 2005.

[14] Stefan Nadschläger, Andreea-Hilda Kosorus, Josef Küng, and Andreas Bögl. Content-based recommendations within a qa system using the hierarchical structure of a domain-specific taxonomy. In *Database and Expert Systems Applications (DEXA), 2012 23rd International Workshop on*, pages 88 – 92. IEEE, 2012.

[15] M. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[16] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *HLT-NAACL*, 2003.

[17] Min Wu, Xiaoyu Zheng, Michelle Duan, Ting Liu, and Tomek Strzalkowski. Question answering by pattern matching, web-proofing, semantic form proofing. In *NIST Special Publication, in: The 12th Text Retrieval Conference (TREC), 2003*, pages 255–500, 2003.