Bachelorarbeit

im Studiengang Computerlinguistik

an der Ludwig- Maximilians- Universität München

Fakultät für Sprach- und Literaturwissenschaften

Department 2

Highlighting von Suchmaschinentreffern in der normalisierten HTML Ausgabe des Nachlasses von Ludwig Wittgenstein

vorgelegt von
Behzat Cinar

Betreuer:	Dr. Max Hadersbeck
Prüfer:	Dr. Max Hadersbeck
Bearbeitungszeitraum:	01. April - 11. Juni 2019

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 06. Juni 2019

.....

Behzat Cinar

Danksagung

Zunächst möchte ich mich bei meinem Betreuer, Dr. Maximilian Hadersbeck, bedanken. Während der gesamten Bearbeitungszeit hat Herr Dr. Hadersbeck mich unterstützt und mir jede erdenkliche Hilfe und Information zukommen lassen, um mich im Wittfind-Kosmos besser zurechtzufinden zu lassen. Darüber hinaus möchte ich mich bei Matthias Lindinger und Florian Landes bedanken, die mir aufgrund ihrer früheren Arbeiten am WiTTFind sehr wertvolle Informationen geben konnten.

Abstract

In der vorliegenden Arbeit wird der Frage nachgegangen, ob es möglich ist, mit neuen Inputdateien im HTML-Format das Highlighten von Suchergebnissen der Suchmaschine WiTTFind zu realisieren. Die Partneruniversität Bergen, Norwegen, verwaltet den Nachlass des Philosophen Ludwig Wittgensteins und arbeitet im Rahmen der elektronischen Publikation des Nachlasses ganz eng mit dem Centrum für Informations- und Sprachverarbeitung (CIS) in München. Unter anderem wurde auf Basis der von Bergen bereitgestellten XML-Daten am CIS die Suchmaschine WiTTFind entwickelt. Seit neuestem liefert das Wittgenstein Archiv Bergen (WAB) den Wittgenstein-Nachlass auch in HTML-Form aus. Mit dieser neuen Datenhaltung dürfte insgesamt eine bessere Darstellung des Nachlasses, z. B. in den Suchergebnissen von WiTTFind, möglich sein. Dies zu erforschen ist das Hauptanliegen der vorliegenden Arbeit.

Inhaltsverzeichnis

1. Einleitung

In der vorliegenden Arbeit wird es darum gehen, die interne Arbeitsweise der Suchmaschine WiTTFind im Allgemeinen und die strukturierte Darstellung von Suchergebnissen in der sogenannten normalisierten Form im Besonderen zu beschreiben. Insbesondere das Hightlighting von Suchergebnissen wird als zentrale Problemstellung angegangen. Die Forschungsfrage der zugrundeliegenden Bachelorarbeit leitet sich daher von der Überlegung ab, ob es möglich ist, aus den von der Universität Bergen neuerdings in HTML-Form bereitgestellten „Rohdaten“ entsprechende Inputdateien für die Suchmaschine WiTTFind zu generieren, die am Ende ein Highlighting in der normalisierten Darstellung des Outputs der Suchmaschine ermöglichen.

Diese Frage hat sich in den Vordergrund gedrängt, da seit geraumer Zeit die Kooperationspartner an der Universität Bergen den Wittgenstein-Nachlass nicht nur in XML-Format, sondern auch in HTML-Format für die Verarbeitung im Rahmen des Projekts „Wittgenstein im co-text“ bereitstellen. Somit dürften effizientere Darstellungsmöglichkeiten der Suchergebnisse von WiTTFind möglich sein. Ziel ist es, die HTML-Ansicht der Ergebnisse von WiTTFind so gut es geht von der HTML-Ansicht der aus Bergen gelieferten Daten nicht unterscheiden zu lassen. Die Verarbeitung von HTML-Daten als „Rohstoff“ für die Suchmaschine WiTTFind unterscheidet sich grundlegend zu der Verarbeitung von XML-Daten. Bei der XML-Verfahrensweise werden die XML-Strukturen so sehr aufgebrochen, dass eine spätere Transformation mittels XSLT zu einer sehr rudimentären HTML-Darstellung von Suchergebnissen führt und dementsprechend in hohem Maße von der HTML-Darstellung von Bergen abweicht. Mit der neuen Verfahrensweise soll daher die Lücke in der Darstellung von adäquaten Suchergebnissen von WiTTFind geschlossen werden.

2. Aufgabenstellung

Um die neue Verarbeitungsweise zu implementieren, bedarf es einer Vielzahl von Methoden und Technologien. Zunächst ist der in HTML-Form vorliegende Wittgenstein-Nachlass nach relevanten Abschnitten zu zerlegen und als entsprechende Einzelteile in einer Datenbank abzuspeichern.

Für das Zerlegen der HTML-Dateien wird die Skriptsprache Python und die dazugehörige Bibliothek LXML-HTML verwendet. Dabei werden alle HTML-Seiten mit LXML-Methoden geparst und jene Bereiche extrahiert, die sich innerhalb von <ab> Tags befinden. Diese Extrakte werden im Rahmen der Continuos Integration mittels eines in Perl geschriebenen Skripts in eine MongoDB-Datenbank gespeichert. Die Datensätze in der MongoDB liegen im JSON-Format vor. Das heißt, die jeweiligen Extrakte haben als Key-Wert eine Siglum-Kennzeichnung und als Value-Wert den gesamten jeweiligen HTML-Tags-Block.

Das Parsen und Zerlegen von HTML-Dateien und das anschließende Speichern von Extrakten aus den HTML-Dateien in eine Datenbank stellt, wenn man das Bild einer Pipeline an dieser Stelle bemüht, das erste Drittel der Pipeline dar. Das zweite Drittel ist aus der Perspektive der vorliegenden Bachelorarbeit eher von theoretischer Natur, da der Vorgang im Mittelteil der Pipeline vom WF, dem WiTTFind-Server, also der eigentlichen Search-Engine von WiTTFind, realisiert wird. Dennoch soll darauf eingegangen werden, da das Wissen um den WF elementar ist, um die von ihr zurückgelieferten Ergebnisse wie z. B. Hitlisten zu verstehen und für die neue Methode der Darstellung der normalisierten HTML-Ansicht zu verarbeiten.

Eine solche Hitliste liefert nämlich – für das letzte Drittel der Pipeline sozusagen – ein Siglum, bzw. eine Liste von Siglen, zu dem / denen die entsprechenden Datensätze aus der MongoDB ausgelesen und für die Darstellung der gehighlighteten Suchergebnisse verwendet werden können. Das Auslesen, das Highlighten und die normalisierte Darstellung werden mit der Skriptsprache Javascript realisiert.

3. Die Suchmaschine WiTTFind

Die Suchmaschine WiTTFind findet zu dem vom User eingegebenen search string entsprechende Textpassagen im Werk Ludwig Wittgensteins. In erster Linie eignet sich diese Suchmaschine für Geisteswissenschaftler sowie Forschern aus den Digital Humanities, um ihnen ein weiteres Mittel an die Hand zu geben, das Werk Ludwig Wittgensteins zu erforschen. Darüber hinaus dürften philosophisch Interessierte zum erweiterten Adressatenkreis von WiTTFind gehören. Nicht zuletzt dient es der computerlinguistischen Grundlagenforschung.

3.1. Wittgenstein und sein Nachlass

Ludwig Wittgenstein (* 26. April 1889 in Wien; † 29. April 1951 in Cambridge) entstammt einer großbürgerlichen Familie in Wien. Im Juli 1914 beschloss Wittgenstein, einen Teil seines beachtlichen Erbes für wohltätige Zwecke zu verwenden und übergab [...] 100.000 Kronen mit der Bitte, das Geld nach Gutdünken an bedürftige österreichische Künstler zu verteilen. (Wikipedia). Er ließ sich in England nieder und lehrte in Cambridge. Seine philosophischen Werke legten den Grundstein für zwei philosophische Schulen: der Logische Positivismus und die analytische Sprachphilosophie. [1]

Der Großteil seines Werks – in Manuskripten und Typoskripten festgehalten – wurde zu seinen Lebzeiten nicht veröffentlicht. Die Manuskripte und Typoskripte werden als Nachlass am Wittgenstein Archive Bergen (WAB) an der Universität Bergen verwaltet. Das WAB entschied sich den Nachlass für die Öffentlichkeit freizugeben.

3.2. WABs elektronische Edition in diplomatischer und normalisierter Form

Der Nachlass besteht als Text in zwei Formen: einmal in diplomatischer Form und einmal in normalisierter Form. Die diplomatische Form zeigt die Texte und alle dazugehörigen Details, wie z. B. Streichungen und Ersetzungen bis hin zu Schreibfehlern in Originalform. Die Publikation der diplomatischen Form wird durch Faksimiles, also durch Abbilder von Manuskripten und Typoskripten, realisiert.

Die normalisierte Version hingegen *„verzichtet auf gestrichene oder überschriebene Passagen, ebenso wurden Schreibfehler korrigiert und normalisiert. Ludwig Wittgenstein überarbeitete seine Texte auch häufiger und so entstanden verschiedene Varianten ein und derselben Bemerkung. Von diesen ist im Standardfall nur die chronologisch letzte in der normalisierten Fassung zu finden, frühere können bei Bedarf allerdings auch angezeigt werden.* [2]

3.3. WiTTFind im Rahmen von WAST

WiTTFind stellt das zentrale Produkt von mehreren in den letzten Jahren am CIS (Centrum für Informations- und Sprachverarbeitung) in Kooperation mit WAB (Wittgenstein Archive Bergen) entwickelten Programmen dar. Diese software- und computerlinguistischen Entwicklungen werden am CIS unter dem Projektnamen WAST (Wittgenstein Advanced Search Tools) subsummiert.

Neben der Suchmaschine WiTTFind gibt es u. a. folgende Produkte:

- SIS: Symmetric Index Structure
- feedback
- wab2cis
- WIndex

Die Verarbeitungskette ließe sich kompakt folgendermaßen beschreiben: Das WAB beliefert das CIS mit dem digitalisierten Nachlass von Wittgenstein. Über die eigenen Forschungsinteressen hinaus bearbeitet das CIS noch Anforderungen der philosophischen Fakultät, um die Suchmaschine WiTTFind als ein praxistaugliches Produkt anzulegen.

Die Zusammenarbeit der beteiligten Partner kann folgendermaßen skizziert werden:

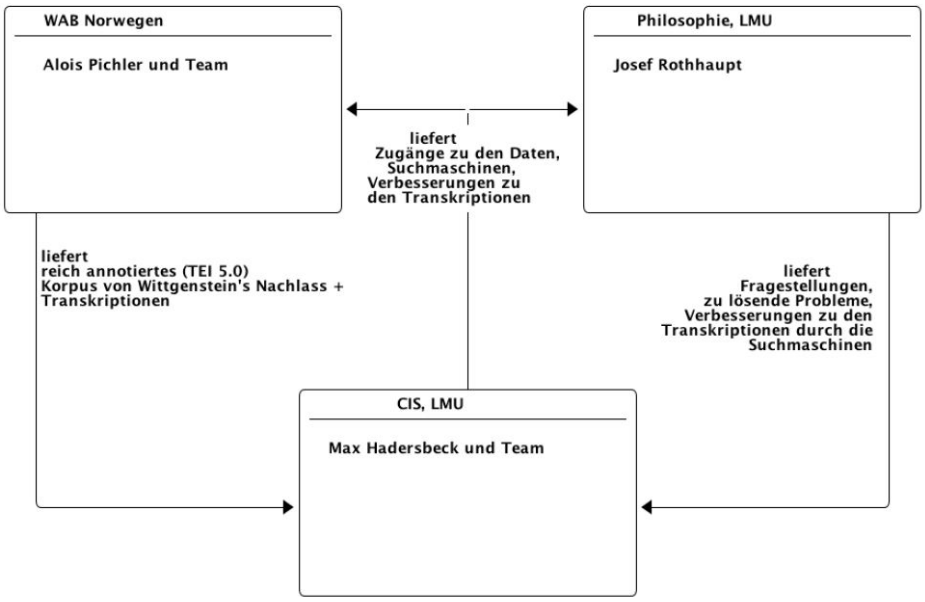

Abbildung 3.1.: Skizze WAST Projektstruktur

3.4. Anwendungsbereiche

Die Softwareentwicklungen der Arbeitsgruppe WAST hat eine Reife erreicht, die dafür sorgt, dass Interessenten aus unterschiedlichsten Bereichen der Digital Humanities Interesse an den Entwicklungen von WAST bekunden. Beispielweise laufen mittlerweile Planungen, die Entwicklungen von WAST als Basis für ein Goethe-Find oder ein Histo-Find zu implementieren. Es lässt sich durchaus schlussfolgern, dass die WAST-Produkte über den Stand computerlinguistischer Grundlagenforschung hinausgekommen sind und somit fertige Softwareprodukte für andere Wissenschaftler im Bereich der Digital Humanities bilden.

3.5. Nachlass im HTML-Format

Der bisherige digitalisierte Datenbestand des Nachlasses bestand in Form von XML-Dateien. Für den diplomatischen Datenbestand besteht natürlich weiterhin die XML-Kodierung, da eine Vielzahl von TEI-Annotationen (Text Encoding Initiation) notwendig ist, um den Nachlass in seiner Komplexität (Streichungen, Verbesserungen, etc.) wiederzugeben. Die normalisierte Form des Nachlasses veröffentlicht das WAB seit geraumer Zeit allerdings auch als HTML-Dateien. [3]

So ergeben sich neue Herangehensweisen für die Mitarbeiter am CIS. Insbesondere die verbesserte Darstellungsmöglichkeit der normalisierten Texte innerhalb der Suchergebnisse von WiTTFind bieten Anlass, die angestammten Wege der Darstellung zu verlassen. Bis jetzt wurden die XML-Dateien, bzw. die relevanten Bereiche davon, mit Hilfe von XSLT in HMTL umgewandelt. Diese Art von Darstellung hat sich in der Vergangenheit immer als suboptimal erwiesen, da eine solche Darstellung in HTML-Form immer eine mit Lücken behaftete Darstellung war. Es traten bei der Umwandlung der Daten technologische Limitierungen zu Tage, die es verhinderten, alle Informationen in den XML-Dateien in adäquates HTML umzuwandeln. Mal fehlte ein Tag, mal ein Absatz, etc. Mit den HTML-Dateien als Input können nun „Umwandlungsverluste“ vermieden werden, da HTML reinkommt und HTML - mit Javascript ergänzt – wieder rauskommt. Die folgenden Absätze dokumentieren die Bemühungen, diesem Ziel gerecht zu werden.

4. Angewandte Technologien und Skriptsprachen

Die Suchmaschine WiTTFind basiert auf einer Vielzahl von Technologien. Für die zugrundeliegende Arbeit waren insbesondere die nachfolgend beschriebenen Technologien von zentraler Bedeutung.

4.1. MongoDB

Für die Speicherung der Input-Dateien in JSON-Format wurde die Datenbank MongoDB eingesetzt. Bei MongoDB handelt sich um einen Vertreter der sogenannten NO-SQL-Datenbanken.

Hier ein Ausschnitt über die Vielfalt der NO-SQL-Datenbanken:

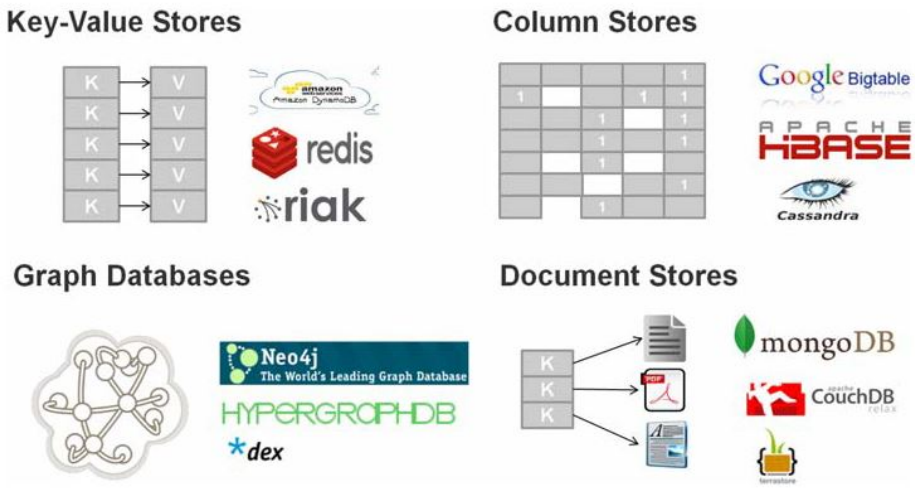

Abbildung 4.1.: Beispiel für NO-SQL-Datenbanken

Das „NO“ ist eher als „not-only-SQL“ statt „Nicht-SQL“ zu verstehen. Es gibt verschiedene Organisationsformen der Daten in NO-SQL-Datenbanken. MongoDB wird zu den dokumentenorientierten Datenbanken zugeordnet. In der MongoDB werden die Daten zwar als Key-Value-Paare organisiert, allerdings besteht der Value-Bereich aus so ausgedehnten Wertebereichen, dass häufig von Dokumenten statt von Key-Values gesprochen wird. Für die weitere analytische Betrachtungsweise soll der Einfachheit wegen dennoch weiterhin von Key-Value-Datensätzen statt von Dokumenten gesprochen werden. (vgl. Trelle:2014, S. 11ff) [4]

Nichtsdestotrotz weist MongoDB eine große Verwandtschaft zu relationalen Datenbanksystemen auf, da Daten im Value-Bereich in JSON-Format modelliert und somit in komplexen Hierarchien verschachtelt werden können. Somit bleiben die Daten immer abfragbar und es können Abfragen gestellt werden, die einer SQL-Abfrage nicht unähnlich sind.

Im Rahmen des Projekts WiTTFind ist die Wahl des Datenbanksystems zugunsten der NO-SQL-Datenbank MongoDB ausgefallen, da es für die zugrundeliegende Arbeit entscheidend war, in wie vielen Dimensionen die Relationen innerhalb der Input-Daten für die Suchmaschine existieren. Im vorliegenden Fall gibt es lediglich nur eine Relation: ein Siglum und den dazugehörigen HTML-Block. Hier wurde also MongoDB nicht wegen dessen zuvor propagierten Komplexitätsperformanz vorgezogen, sondern aufgrund dessen einfacher Datenhaltung in Key-Value-Format. Insofern kam ein relationales Datenbanksystem nicht in Betracht und es wurde die Nischenlösung MongoDB gewählt.

Relationale Datenbanksysteme sind Allzweckwaffen, die sich in der Regel auf eine große Menge von Problemen anwenden lassen. Im Gegensatz dazu stellen NoSQL-Datenbanken tendenziell eher Nischenlösungen dar, die bestimmte Probleme allerdings sehr viel besser lösen können. (Trelle:2014, S. 13) [4]

4.2. XML / JSON

Bei XML und JSON handelt es sich um Daten(austausch)formate. Das XML (Extensible Markup Language) ist in erster Linie als eine Auszeichnungssprache zur Wiedergabe hierarchisch organisierter Datenstrukturen bekannt. Mit dem XML-Dialekt XSLT lässt sich XML-Inhalt problemlos in HTML und somit für Browser in lesbare Form umwandeln.

Darüber hinaus ist XML jedoch auch für den plattformunabhängigen Datenaustausch zwischen Computern bekannt. Insofern kann von jedem Computer die aufwendige XML-Annotierung des Wittgenstein-Nachlasses auf Basis von TEI (Text-Encoding-Initiative) gelesen und maschinell verarbeitet werden. (vgl. Vonhoegen:2015, S. 41) [5]

JSON (JavaScript Object Notation) als Datenaustauschformat ist ebenfalls plattform- und programmiersprachenunabhängig (Buch dazu finden) und wird besonders in der Webentwicklung zum Senden und Speichern von strukturierten Daten zwischen Computern eingesetzt. Hauptcharakteristikum von JSON-Datensätzen ist die Organisation in Key-Value Form. Key und Value werden dabei von einem Doppelpunkt getrennt:

```
{
 "Ms-101,IIr[1]": " <table width=\"100%\">                          <tr>
 "Ms-101,IIr[2]": " <!--s n=\"Ms-101_IIr.2\" ana=\"facs:Ms-101,IIr abnr:2 satznr:2\"-
 "Ms-101,1r[1]": " <div align=\"right\" style=\"margin-right: 50px;\">9.8.14.</div>
 "Ms-101,1r[2]": " <div align=\"right\" style=\"margin-right: 50px;\">10.8.14.</div>
 "Ms-101,2r[1]": " <table width=\"100%\">                          <tr>
 "Ms-101,2r[2]": " <div align=\"right\" style=\"margin-right: 50px;\">11.8.14.</div>
 "Ms-101,2r[3]": " <div align=\"right\" style=\"margin-right: 50px;\">13.8.14.</div>
 "Ms-101,3r[1]": " <table width=\"100%\">                          <tr>
 "Ms-101,3r[2]": " <div align=\"right\" style=\"margin-right: 50px;\">15.8.14<span st
 "Ms-101,5r[2]": " <span style=\"font-style: italic;\">
 "Ms-101,5r[4]": " <span style=\"font-style: italic;\">
 "Ms-101,6r[2]": " <span style=\"font-style: italic;\">
 "Ms-101,7r[2]": " <div align=\"right\" style=\"margin-right: 50px;\">21.8.14.</div>
 "Ms-101,8r[2]": " <table width=\"100%\">                          <tr>
 "Ms-101,8r[3]": " <span style=\"font-style: italic;\">
 "Ms-101,9r[2]": " <div align=\"right\" style=\"margin-right: 50px;\">25.8.14.</div>
 "Ms-101,11r[2]": " <span style=\"font-style: italic;\">
```


Abbildung 4.2.: Beispiel für Key-Value-Paare in JSON-Files

Die Inputdateien aus Bergen erhält das CIS in XML- und neuerdings auch in HTML-Form. Für die Zukunft sollen ausschließlich die HTML-Dateien als Inputdateien für die normalisierte Darstellung der Suchmaschinenergebnisse verwendet werden – wohlgemerkt nur die für normalisierte Darstellung. Für die diplomatische Darstellung werden weiterhin XML-Inputdaten verwendet.

4.3. Python

Die Programmiersprache Python bietet mit seinen umfangreichen Bibliotheken die Möglichkeit Text maschinell optimal zu verarbeiten. In den letzten Jahren gewann Python immer mehr an Zulauf und ist heute ein Quasi-Standard für Programmiersprachen, um Natural Language Processing (NLP) zu betreiben. Für das Parsen von HTML-Dateien hat sich insbesondere die Bibliothek LXML-HTML als zentrales Instrument erwiesen.

Mit LXML-HTML werden die aus Bergen zugelieferten HTML-Dateien, welche den gesamten Nachlass von Wittgenstein beinhalten, nach bestimmten Tags geparst. Es handelte sich um die <ab> Tags (anonymer Block). Jede Fundstelle hinsichtlich eines solchen <ab> Tag-Blocks beinhaltet ein Siglum und den dazugehörigen Text. Ein derartiger Fund beim Parsen stellt folglich ein Key-Value-Paar dar und ist somit geeignet einen Key-Value-Datensatz in der MongoDB zu bilden. [6]

4.4. Perl

Die Programmiersprache Perl war die erste Wahl, um die – mit der Python-Bibliothek LXML-HTML aus HTML-Dateien extrahierten – Datensätze im Rahmen des Continous Integration in die MongoDB zu importieren. Perl ist für diese Aufgabe am CIS noch der Standard, da es mit der Suchmaschine WiTTFind historisch gewachsen ist. Für das Importieren von Daten in die MongoDB schien es daher kontraproduktiv zu sein, eine andere Variante des Datenimports zu entwickeln, vor allen Dingen um einen Flickenteppich an Lösungsmöglichkeiten für *eine* Aufgabe zu vermeiden.

4.5. Javascript

Die meisten Komponenten der Suchmaschine WiTTFind sind als node.js-Projekte angelegt. Hier ist an erster Stelle die Komponente Quadroreader zu nennen, die für die diplomatische und normalisierte Darstellung der Suchergebnisse von WiTTFind zuständig ist. Bei einem node.js-Projekt wird der Zugriff sowohl auf Front- und Backend als auch auf die MongoDB zum Auslesen der darzustellenden Daten in der Programmiersprache Javascript programmiert. (vgl. Hughes-Croucher & Wilson:2012, S. 1) [7] Insofern wurden die zentralen Programmieraufgaben der zugrundeliegenden Arbeit, wie z. B. das Auslesen oder das gehightlightete Darstellen von Suchergebnissen, in Javascript gelöst.

5. Von der Verarbeitung der Rohdaten zur Darstellung der Suchergebnisse

Die Suchmaschine WiTTFind entstand auf der Grundlage des Nachlasses von Ludwig Wittgenstein. Die Universität Bergen, die den Nachlass von Ludwig Wittgenstein verwaltet, hat den Nachlass digitalisiert und den gesamten Korpus auf Basis der TEI-Annotation (Text-Encoding-Initiative) mit XML-Tags versehen. Somit war ein Datenaustausch mit anderen Forschungsinstituten möglich und schließlich wurde der Wittgenstein-Nachlass für die Öffentlichkeit freigegeben. Das CIS hat relativ früh Interesse an einer Zusammenarbeit mit der Universität Bergen bekundet und somit Zugriff auf den digitalisierten Nachlass von Wittgenstein erhalten.

Die XML-annotierten Texte dienen am CIS vor allem als Rohdaten, um im Rahmen von computerlinguistischen Verarbeitungsprozessen unterschiedliche Softprodukte zu erstellen und somit das Werk Wittgensteins auf digitale Weise für Interessenten aus dem Bereich der Digital Humanities zugänglich zu machen. Mit den neuen HTML-Inputdateien wird eine noch bessere Darstellung im WiTTFind angestrebt. Denn bisher war es immer annäherungsweise möglich die Sucherergebnisse von WiTTFind der HTML-Darstellungsweise des Wittgenstein Nachlasses der Universtität Bergen anzugleichen. Es ist allerdings Wunsch des WAB, dass ihre Kooperationspartner die Darstellung des Wittgenstein Nachlasses in einer einheitlichen, von der Darstellungsweise des WAB nicht zu stark abweichenden Form darzustellen.

Die XML-Dateien wurden für die normalisierte Darstellung der Suchergebnisse bisher aufwendig mit der XML-Transformationssprache XSLT in HTML konvertiert. Auf diesem Weg war es allerdings nicht möglich alle Formatierungen zu realisieren, die es auch in der HTML-Ansicht des WAB gibt. Insofern war es naheliegend den von WAB nun in HTML bereitgestellten Korpus als neue Inputdaten für die Darstellung der normalisierten Form der Suchergebnisse in WiTTFind zu verwenden.

5.1. Extraktion der relevanten HTML-Tags

Für die Darstellung der normalisierten Form der Sucherergebnisse ist es essentiell, die in den Inputdateien bereitgestellten relevanten Inhalte zu extrahieren. Als relevant wird jeder HTML-Block gesehen, der mit einem gewissen Kommentar beginnt und endet. Dieser Kommentar besteht aus dem Schriftzug „ab“ und einem Siglum. Der Schriftzug „ab“ ist in (früheren) XML-Inputdateien noch ein auswertbarer Tag und steht für anonymen Block. Hier ein Ausschnitt aus einem solchen XML-Inputfile:

```
<ab n="Ms-101,25r[3]"
    ana="abnr:111"
    date="pub:W-NB_date:19140921"
    date_norm="1914-09-21">
   <seg type="publ">
      <s n="Ms-101,25r[3]_1" ana="facs:Ms-101,25r abnr:111 satznr:605">
         <seg type="notation">φa, ψb <space xmlns:w2c="http://data.ub.uib.no/wab2cis/"
   </seg>
</ab>
<ab n="Ms-101,25r[4]"
    ana="abnr:112"
    date="pub:W-NB_date:19140921"
    date_norm="1914-09-21">
   <seg type="publ">
      <s n="Ms-101,25r[4]_1" ana="facs:Ms-101,25r abnr:112 satznr:606">Wenn ich sage:
   </seg>
</ab>
<ab n="Ms-101,25r[5]"
    ana="abnr:113"
    date="pub:W-NB_date:19140921"
    date_norm="1914-09-21">
   <seg type="publ">
      <s n="Ms-101,25r[5]_1" ana="facs:Ms-101,25r abnr:113 satznr:607">Es scheint mir
```

Abbildung 5.1.: Beispiel für einen ab-Block im XML-Format

In HTML-Inputdateien jedoch leitet „ab“ nur einen Kommentarblock ein und schließt es an passender Stelle wieder.

```
<!--ab n="Ts-213_Ir.2" ana="abnr:2"-->
<table border="0" cellpadding="0" cellspacing="0" class="mainAB"
   <tr>
      <td colspan="8">
         <div width="100%" style="background-color:gray;">Ts-213_
      </td>
   </tr>
   <tr>
      <td width="75px" valign="top"> </td>
      <td width="20px" valign="top" text-align="right" style="fon
      <td class="sm-left" width="20px" valign="top" text-align="c
            <div style="text-align: center;"></div></span></td>
      <td class="sn-left" width="20px" valign="top" text-align="c
      <td width="5px"> </td>
      <td style=" text-align: left;" valign="top" width="700px">
         <div class="ab-style" style="margin-bottom: 10px;width:
            <div class="text-override" style=""><a href="#segTs-2
                  1)

                  <!--s n="Ts-213_Ir.2" ana="facs:Ts-213,Ir abnr:
                  <!--/s-->


                  <!--s n="Ts-213_Ir.2" ana="facs:Ts-213,Ir abnr:
                  <!--/s--> </a><BR><BR></div><br> </div>
      </td>
      <td class="sn-right" width="20px" valign="top" text-align="
      <td class="sm-right" width="20px" valign="top" text-align="
   </tr>
</table>
<!--/ab-->
```

Abbildung 5.2.: Beispiel für einen ab-Block im HTML-Format

Es gilt nun innerhalb eines solchen Kommentarblocks das Siglum als Key und den relevanten darzustellenden HTML-Block als Value zu extrahieren. Ein derart extrahiertes Key-Value-Paar ist sozusagen ein Snippet, das als Dokument, bzw. als Key-Value-Paar in

JSON-Format in der Datenbank gespeichert wird.

Um die Extraktionsaufgabe bestmöglich lösen zu können, hat sich die Python-Bibliothek LXML-HTML am effektivsten erwiesen. Der Grundgedanke beim Einsatz von LXML-HTML ist es, jede einzulesende Datei mit der LXML-HTML-Methode fromstring() zu parsen:

```
59      htmlString = dateiobjekt.read()
60      html = lxml.html.fromstring(htmlString)
61      mainab_Tags=html.find_class("mainAB")
```

Abbildung 5.3.: LXML-HTML-Methode fromstring

Mit der Methode find_class() wird anschließend jeder Block gefunden, in dem das Attribut „class“ mit „mainAB“ bezeichnet ist:

```
<!--ab n="Ms-101_IIr.2" ana="abnr:2"-->
<table border="0" cellpadding="0" cellspacing="0" class="mainAB" width="880px">
   <tr>
      <td colspan="8">
         <div width="100%" style="...">Ms-101_IIr.2 (date: 19140809)</div>
      </td>
   </tr>
   <tr>
      <td width="75px" valign="top"> </td>
      <td width="20px" valign="top" text-align="right" style="..."><span style="..."> </span></td>
      <td class="sm-left" width="20px" valign="top" text-align="center"><span style="...">
            <div style="..."></div></span></td>
```

Abbildung 5.4.: table-Attribut class

Somit kann über alle Blöcke geparst und das Siglum als Key und die relevanten HTML-Tags für die Darstellung in normalisierter Form als Value ermittelt werden. Beispielhaft soll hier die Ermittlung der Siglen veranschaulicht werden.

Zunächst wird in einer Schleife über alle Blöcke iteriert und die Speicherorte der HTML-Elemente in der Liste „tabellenspalten“ gespeichert:

```
for current_main_AB_tag in mainab_Tags:
    tabellenspalten = current_main_AB_tag.getchildren()
    siglenspalte = tabellenspalten[0]
```

Abbildung 5.5.: Iteration über alle Blöcke mit dem Attibut class=mainAB

Das erste Element in der Liste „tabellenspalten“ (siehe oben Code) ist der Speicherort des ersten <tr> Blocks:

Abbildung 5.6.: Im ersten <tr> Block ist das Siglum

Als nächstes wird über den Speicherplatz aller Tags iteriert und mit der Methode text_content() ausgelesen:

```
siglum_div_tag=next(siglenspalte.iter('div'))
roh_siglum_aus_html = siglum_div_tag.text_content()
```

Abbildung 5.7.: Iteration über Tags

Die ausgelesene Information enthält einerseits das Siglum und andererseits eine Datumsangabe.

Abbildung 5.8.: Siglum mit Datumskennzeichen

Die ausgelesene Information wird schließlich mit Regex- und split Methoden um die Datumsinformation bereinigt, so dass am Ende das reine Siglum als Key für die Speicherung in einer JSON-Datei zur Verfügung steht:

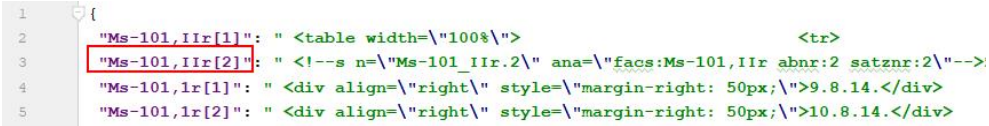

Abbildung 5.9.: Siglum ohne Datumskennzeichen

5.2. Speicherung der relevanten HTML-Tags in MongoDB mittels Perl

Für die Speicherung der in JSON-Files strukturierten Siglum-HTML-Block-Paare, bzw. Key-Value-Paare in die Datenbank kommt die Programmiersprache Perl zum Einsatz. Das Perl-Importskript ist im Repository des Projekts „quadroreader-data“ eingecheckt, da es momentan nur im Quadroreader die Schnittstelle zur MongoDB-Datenbank gibt.

Zunächst wird eine config.json Konfigurationsdatei eingelesen, in der die wichtigsten Datenbankverbindungsparameter gelistet sind:

```
19  {
20      # Read the config file
21      my $dataDir = shift();
22      my $configFile = catfile($dataDir, "config.json");
23      my $config = do { local (@ARGV, $/) = $configFile; decode_json(<>); };
```

Abbildung 5.10.: Einlesen des Config-Files

Anschließend wird eine Verbindung zur MongoDB aufgebaut:

```
25  # Open the database connection
26  my $client = MongoDB::MongoClient->new(host => $config->{"MONGO_HOST"}, port => $config->{"MONGO_PORT"});
27  my $database = $client->get_database($config->{"DATABASE"});
28
```

Abbildung 5.11.: Database-Connection zur MongoDB

In einem weiteren Schritt wird ein Logfile spezifiziert:

```
29  # Open the log file
30  my $logFileDir = catdir($dataDir, "logs");
31  mkdir($logFileDir);
32  open(LOGFILE, ">>", catfile($logFileDir, "transcription-import.log"));
33  say LOGFILE "##############################";
34  say LOGFILE "Log for importHtmlTranscriptionToDB.perl | time: ".strftime('%Y-%m-%d %H:%M',localtime);
35
```

Abbildung 5.12.: Spezifikation des Logfiles

Als nächstes wird für jede eingelesene JSON-Datei die Transcription ermittelt. Die Transkription enthält Felder wie z. B. NORM.xml, tagged-annotated.xml, usw. Das Neue an der Transcription sind die in HTML-Form vorliegenden Datensätze.

Schließlich wird überprüft, ob es den Eintrag in der Datenbank schon gibt. Wenn nicht, wird ein INSERT ausgeführt. Wenn es den Eintrag bereits gibt, wird ein UPDATE ausgeführt. Beim UPDATE wird überprüft, ob der einzufügende Eintrag identisch zum bestehenden Datensatz ist. Wenn sie identisch sind, wird nichts gemacht (Siehe Zeile 67),

ansonsten wird geupdated.

```
foreach(sort keys %{$htmlSnippets}){
  # Check if there is already an entry for the current siglum
  $entry = $collection->find_one({"siglum" => $_});
  if(!$entry){
    $collection->insert_one({
      "siglum" => $_, "html" => $htmlSnippets->{$_}, "remark-number" => $counter++
    });
    say LOGFILE "INSERT new siglum ".$_;
  }else{
    if(!Compare($htmlSnippets->{$_},$entry->{"html"})){
      $collection->update_one({"siglum" => $_},{'$set' => {"html" => $htmlSnippets->{$_}}});
      say LOGFILE "UPDATE siglum ".$_;
    }else{
      say LOGFILE "NOTHING TO BE DONE for siglum ".$_;

    }
  }
}
}
```

Abbildung 5.13: Logik bezüglich Datenbankimport

5.3. Interne Verarbeitung

Eine Suchanfrage in WiTTFind stößt eine ganze Reihe von Prozessen an. Diese Prozesse sind im Hintergrund auf mehreren in Gitlab organisierten Projekten verteilt. Die drei wichtigsten heißen „Wittfind-web“, „WF“ und „Quadroreader“. Die Interaktion dieser drei Plattformen ist für die Funktionsfähigkeit von WiTTFind unabdingbar.

Die Suchanfrage wird zunächst im Rahmen von Wittfind-web mit der in der Datei search.js definierten Routine WiTTFind_search() verarbeitet. Dabei wird das entgegengenommene search_pattern mit einem Post-Request an den Webserver wfa geschickt. Die Suchanfrage ist bezüglich des erwarteten Ergebnisses auf maximal 200 limitiert. Hier ein Ausschnitt aus der Funktion WiTTFind_search():

```
function wittfind_search(search_pattern)
{
  start_loading_animation();
  var wfa_host = c_wfa["host"];
  var wfa_port = c_wfa["port"];

  $.post("http://" + wfa_host + ":" + wfa_port,
         JSON.stringify( value: { "query" : search_pattern, "max": 200 }),
         function() {}, "json")
  .done(function(data, matches) {


    var wfa_results = data;


    var hit_array = [];
    var hit_siglum_array = [];
```

Abbildung 5.13.: Ausschnitt der Funktion WiTTFind_search()

Der Webserver wfa ist eine Komponente aus dem Projekt WF. Im Projekt WF ist die

eigentliche – in der Programmiersprache C++ entwickelte – search engine von WiTTFind. Der Webserver wfa nimmt also die Anfrage von search.js und leitet diese an den WiTTFind-Server. Der WiTTFind-Server durchsucht alle zur Verfügung stehenden WiTTFind-Dokumente und bildet zu jedem gefundenen Treffer einen Hit. Da es üblicherweise mehr als einen Treffer gibt, wird eine ganze Hitliste an den Webserver wfa zurückgeschickt, der es wiederrum an search.js weiterleitet, wo der Suchbegriff zum ersten Mal entgegengenommen und verarbeitet wurde. Das Rückgabeergebnis von der search engine über wfa an search.js befindet sich in der Variable „data“. Mit dem Inhalt dieser Variable ist nun die Grundlage geschaffen, um über entsprechende Funktionen des Skripts search.js und weiteren Skripten des Projekts Quadroreader (dazu später mehr in Kapitel 4.5) an die Snippets zu gelangen, die vorher in einem Vorverarbeitungsprozess aus den HTML-Inputdateien extrahiert und in der MongoDB-Datenbank abgelegt wurden.

Der Inhalt der Variable „data“ besteht nämlich aus einem Array, dessen Elemente selber wiederrum Key-Value-Paare in JSON-Form sind. Als Key wird lediglich die Kennzeichnung „n“ geführt. Auf der Value-Seite sind die wichtigen Siglen. Das heißt, der WF-Server findet zu jedem in WiTTFind eingegebenen Suchbegriff zwar die passenden Textstellen, gibt allerdings nur die dazugehörigen Siglen wider. Hier ein Beispiel für ein solches Hit-Array:

hitArray: [{"n":"Ms-115,39[3]_1"},{"n":"Ms-115,39[3]_2"},{"n":"Ms-115,206[2]et207[1]et208[1]_16"},{"n":"Ms-135,82r[3]et82v[1]_2"},{"n":"Ms-135,82r[3]et82v[1]_3"},{"n":"Ms-135,86v[2]et87r[1]_3"},{"n":"Ms-136,47b[4]_3.1"},{"n":"Ms-137,71a[2]_11"},{"n":"Ts-213,199v[3]_1"},{"n":"Ts-213,199v[3]_2"},{"n":"Ts-228,55[4]_1"},{"n":"Ts-228,55[4]_2"},{"n":"Ts-230a,7[4]_1"},{"n":"Ts-230a,7[4]_2"},{"n":"Ts-230b,7[4]_1"},{"n":"Ts-230b,7[4]_2"},{"n":"Ts-230c,7[4]_1"},{"n":"Ts-230c,7[4]_2"},{"n":"Ts-232,604[3]_2"},{"n":"Ts-232,604[3]_3"},{"n":"Ts-233b,29[6]_1"},{"n":"Ts-233b,29[6]_2"}]

Abbildung 5.14.: Beispiel für Hit-Array

Warum werden hier nun die Siglen zu einem bestimmten im Text vorkommenden Begriff ausgegeben und nicht die Textstellen selbst? Der Grund dafür ist, dass die Inputdateien, die die Originaltexte enthalten, computerlinguistischen Verfahren unterzogen werden. Beispielsweise wird aus den Inputdateien eine xml-oa Version hergestellt, in der die Texte in xml-Form vorliegen:

xml-oa: <ab ana="field:PhilosophyOfLanguage_date:19330319?-19330415?" emph="blbef_1" n="Ts-213,199v[3]" xml:id="Ts-213_199v.3" xml:lang="de"> <add rend="ipp_h"> <seg type="wabmarks-secml">v√/</seg> <s type="es">Wir würden kaum fragen, ob das Krokodil etwas <lb/> damit <emph rend="us1_h">meint</emph> wenn es mit offenem Rachen auf <lb/> <choice type="dsl"> <orig type="alt1"> <del type="d_h">uns</del> </orig> <orig type="alt2"> <add rend="i_h">einen Menschen</add> </orig> </choice> zukommt. </s> <s type="es">Und wir w <corr type="trsn"> <orig type="trsn1">u</orig> <reg type="trsn2">ü</reg> </corr>rden erklären, das Kro <lb rend="shyphen"/>kodil könne nicht denken & darum sei eigentlich hier <lb/> von einem Meinen keine Rede. </s> </add> </ab>

Abbildung 5.15.: Beispiel für xml-oa

Oder sie werden in eine pos-getaggte-Form gebracht, in der unterschiedliche Auswertungen

wir Tokenanzahl, Zeilennummerierung, uvm. registriert werden:

xml-pos-tagged: <ab ana="abnr:1019" date="field:PhilosophyOfLanguage_date:19330319?-19330415?" date_norm="1933-03-19?-1933-04-15?" n="Ts-213,199v[3]"> <s ana="facs:Ts-213,199v abnr:1019 satznr:3238" n="Ts-213,199v[3]_1"> <w ana="pagenr:303 linenr:23 tokennr:4" l="wir" t="PPER">Wir</w> <sp/> <w ana="pagenr:303 linenr:23 tokennr:5" l="werden" t="VAFIN">würden</w> <sp/> <w ana="pagenr:303 linenr:23 tokennr:6" l="kaum" t="ADV">kaum</w> <sp/> <w ana="pagenr:303 linenr:23 tokennr:7" l="fragen" t="VV... tokennr:19" l="hier" t="ADV">hier</w> <lb/> <sp/> <w ana="pagenr:303 linenr:26 tokennr:1" l="von" t="APPR">von</w> <sp/> <w ana="pagenr:303 linenr:26 tokennr:2" l="eine" t="ART">einem</w> <sp/> <w ana="pagenr:303 linenr:26 tokennr:3" l="Meinen" t="NN">Meinen</w> <sp/> <w ana="pagenr:303 linenr:26 tokennr:4" l="keine" t="PIAT">keine</w> <sp/> <w ana="pagenr:303 linenr:26 tokennr:5" l="Rede" t="NN">Rede</w> <w ana="pagenr:303 linenr:26 tokennr:6" l="." t="$.">.</w> </s>undefined</ab>

Abbildung 5.16.: Beispiel für xml-pos-tagged

Es gibt also nicht *die* Texstellen, sondern je nachdem, welche Art von Text dargestellt werden soll, ist zu überlegen, wie der gewünschte Text produziert werden kann. Und hier ist die einfachste Lösung auch die plausibelste: Mit den Siglen. Denn alle Texte, ob html-, xml- oder pos-getaggte Texte, sie alle haben die Siglen als gemeinsamen Nenner. Um die Texte für die normalisierte Darstellung zu ermitteln ist also von den Siglen als Ausgangspunkt auszugehen. In unserem Fall schließlich: jeden Value-Bereich in der MongoDB zu finden, dessen Key einem Siglum aus dem – vom wfa-Webserver zurückgelieferten – hitArray entspricht.

5.4. Darstellung der Suchergebnisse

Für die Darstellung der normalisierten Texte ist es zunächst wichtig eine Verbindung zum Quadroreader herzustellen und in einem Get-Request die passenden HTML-Elemente zu einem Siglum zu ermitteln. Die Quadroreader API bietet unter anderem folgende Schnittstelle: **/transcription/:siglum**.

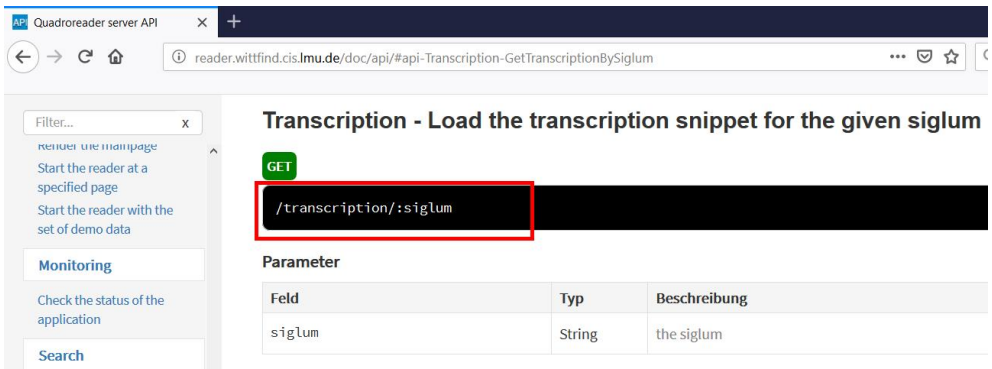

Abbildung 5.17.: Quadroreader API

Die zentralen Dateien im Projekt Quadroreader sind die quadroreader-server.js und die transcription.js. Bei der quadroreader-server.js. wird u. a. die Verbindung zur MongoDB aufgebaut. Die transcription.js liest die Daten aus der MongoDB aus. Um den Get-Request

an den Quadroreader ein wenig plastischer zu veranschaulichen, gäbe man beispielweise den Hostnamen gefolgt von „/transcription/“ gefolgt von einem Siglum in einen Browser ein:

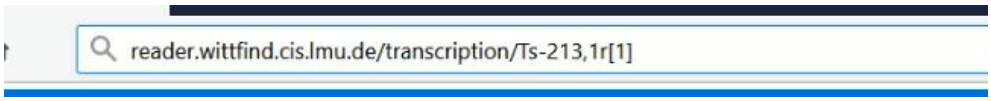

Abbildung 5.18.: Beispieleingabe, um Datenbank mit konkretem Siglum auszulesen

Als Ergebnis erhielte man folgende transcription:

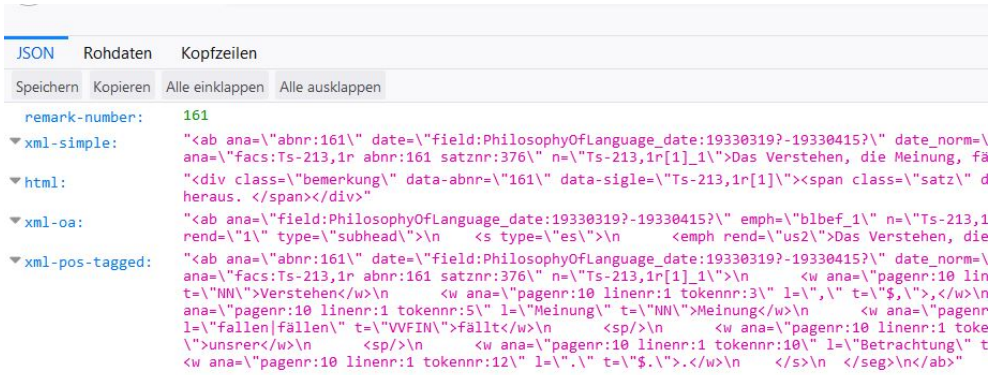

Abbildung 5.19.: Beispielausgabe zu einem Siglum

Der Get-Request als Quellcode-Anweisung ist in der im Projekt Wittfind-web integrierten Datei getHtmlAndDoHighlightWithSiglum.js implementiert:

```
1
2   // Diese Funktion würde in search.js in der Zeile 287 aufgerufen werden.
3   function getHtmlAndDoHighlight(siglum) {
4
5       var qr_host = c_quadroreader["host"];
6       var qr_port = c_quadroreader["port"];
7
8
9       $.get("http://" + qr_host + ":" + qr_port + "/transcription/" + siglum,
10
11          function (htmlContent,search_pattern) {
12
13              //  beginHighlight und endHighlight Tags definieren das Highlighting
14
15              beginHighlight = "<b>";
16              endHighlight = "</b>";
17
18
```

Abbildung 5.20.: Beispiel für ein Get-Request

Der Aufruf der Funktion getHtmlAndDoHighlight() erfolgt ebenfalls in der im Projekt Wittfind-web integrierten Datei search.js.

5.4.1. search.js

Bei der Javascript-Datei search.js handelt sich um die gravierendste Datei im WiTTFind-Ökosystem, da diese eine Art Scharnier zwischen den drei Projekten Wittfind-web, WF und Quadroreader bildet. Die search.js hat über 1600 LOC (Lines of code) und trägt dazu bei diverse computerlinguistische Arbeiten zu realisieren, wie z. B. Pos-Tagging, Frequenzlisten, uvm. Anhand von Frequenzlisten können sogar Informationen gewonnen werden, die ein Bild, bzw. eine Facsimile besser beschreiben und somit einen behindertengerechten Zugang zu den Suchergebnissen von WiTTFind eröffnen können. (Hinweis zu Ivanas Arbeit)

Die search.js schickt mit in ihr implementierten Funktionen Daten sowohl an WF als auch an Quadroreader und empfängt wiederum von diesen beiden Projekten Daten. Es ist das Zusammenspiel dieser drei Projekte, die die Suchmaschine WiTTFind zum Laufen bringen. Beispielsweise schickt die Funktion „WiTTFind_search()“ den vom User ausgewählten Suchbegriff an WF (Siehe Kapitel 4.4) und erhält eine Hitliste von Siglen in Form eines Arrays, welches wiederrum mit der Funktion send_hits_to_quadroreader():

```
1599  function send_hits_to_quadroreader(hit_siglum_array) {
1600    var qr_host = c_quadroreader["host"];
1601    var qr_port = c_quadroreader["port"];
1602
1603    var qr_server_data = {};
1604    qr_server_data.hitArray = JSON.stringify(hit_siglum_array);
1605
1606    var data_id;
1607    var error = null;
1608    $.post("http://" + qr_host + ":" + qr_port + "/data", qr_server_data, function(data) {
```

Abbildung 5.21.: Ausschnitt der Funktion send_hits_to_quadroreader()

... zum Quadroreader geschickt wird:

```
373          hit_array.push(current_hit);
374          hit_siglum_array.push({ 'n': current_hit.matches[0].name });
375        });
376      });
377
378
379      send_hits_to_quadroreader(hit_siglum_array);
380
381      sorted_hit_array = [];
382
```

Abbildung 5.22.: Aufruf der Funktion send_hits_to_quadroreader() in search.js

Die zurückgelieferten Daten vom Quadroreader werden in search.js weiterverarbeitet, wie z. B. in der Funktion getHtmlAndDoHighlight(), in der die gefundenen Treffer zu dem Suchbegriff gehighlighted und schließlich auf der Website in normalisierter Form dargestellt werden.

```
search.js
286  getHtmlAndDoHighlight(siglum,search_pattern).then(function(sentence) {
287    $("#treffer" + index).html(sentence);
288  });
```

Abbildung 5.23.: Aufruf der Funktion getHtmlAnDoHighlight in search.js

5.4.2. Highlighting von Suchergebnissen

Die Funktion zum Highlighten von Suchergebnissen getHtmlAndDoHightlight() ist als Javascript-Datei mit dem Namen getHtmlAndDoHighlightWithSiglum.js im Projekt Wittfindweb gespeichert und wird schließlich in der Datei search.js aufgerufen. Das Hightlighten von Suchergebnissen war ein Schwerpunkt der vorliegenden Bachelorarbeit. Insofern sollen die wichtigsten Passagen dieses Skripts an dieser Stelle näher betrachtet werden.

Als Funktionsparameter erhält die Funktion ein Siglum und den Suchbegriff (seach_pattern). Mit einem Get-Request wird der HTML-Inhalt zum Siglum ermittelt:

```
4   function getHtmlAndDoHighlight(siglum, search_pattern) {
5
6       var qr_host = c_quadroreader["host"];
7       var qr_port = c_quadroreader["port"];
8
9
10      $.get("http://" + qr_host + ":" + qr_port + "/transcription/" + siglum,
11
```

Abbildung 5.24.: Ausschnitt der Funktionsdefinition von getHtmlAndDoHighlight()

Anschließend werden in einer Callback-Funktion die Highlight-Tags definiert, hier das bold-Tag. Sowohl das search_pattern als auch der HTML-Content werden in die Kleinschreibweise konvertiert, damit die Entitäten nicht case-sensitiv sind.

```
11      function (htmlContent,search_pattern) {
12
13          //  beginHighlight und endHighlight Tags definieren das Highlighting
14
15          beginHighlight = "<b>";
16          endHighlight = "</b>";
17
18
19          var highlightHtmLContent = "";
20          var i = -1;
21          var lowerCaseSearchPattern = search_pattern.toLowerCase();
22          var lowerCaseHtmlContent = htmlContent.toLowerCase();
```

Abbildung 5.25.: Definition des Bold-Tags als Highlight-Tag

Schließlich wird in einer While-Schleife der HTML-Content solange nach dem Suchbegriff durchforstet und Fundstellen mit dem bold-Tag versehen, bis es kein HTML-Content mehr gibt. (Siehe Anhang C)

6. Fazit

Die Herausforderung für die zugrundeliegende Arbeit bestand einerseits darin, aus den HTML-Dateien adäquate Snippets zu extrahieren und in eine MongoDB-Datenbank zu importieren und andererseits, die Komplexität der Interaktion von mehreren Projekten (Wittfind-web – WF – Quadroreader) zu durchdringen und ihre gegenseitigen Wechselwirkungen im Hightlighting von Suchergebnissen zu berücksichtigen.

Als Ergebnis kann festgestellt werden, dass das Highlighten von Suchergebnissen in WiTTFind durchaus auch mit der neuen HTML-Datenquelle möglich ist. Der Wittgenstein-Nachlass in HTML-Form bietet somit die Gelegenheit, die Suchergebnisse in normalisierten Textabschnitten so darzustellen, wie es auch in HTML-Darstellungen des WAB geliefert wird. Somit können alle Unzulänglichkeiten überwunden werden, die in der bisherigen Verfahrensweise noch auftreten, wenn z. B. XML-Inputdateien mit der Sprache XSLT in HMTL transformiert werden.

Weiterführende Arbeiten könnten beim Highlighten von konjugierten, bzw. deklinierten Formen des Suchbegriffs ansetzen. Voraussetzung wäre, dass der (in Python geschriebene) wfa-Webserver ein Array von search_patterns zurückgibt. Momentan ist der Workflow noch so, dass Suchergebnisse über den anchored-Server zurückgegeben werden:

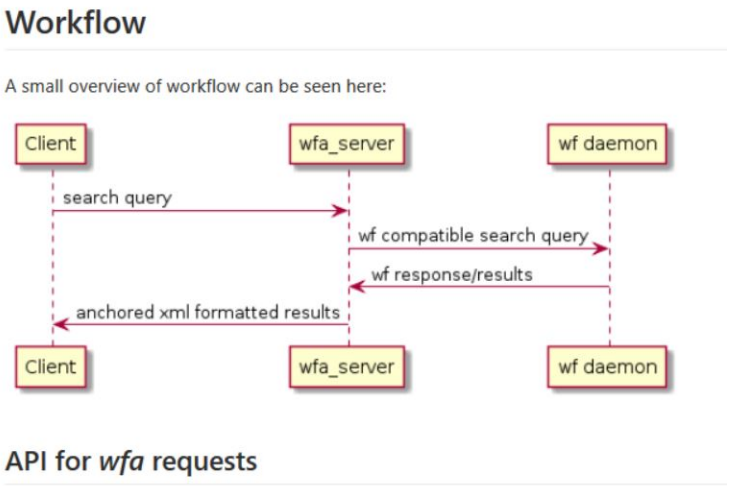

Abbildung 6.1.: Abbildung 6.1: WiTTFind workflow

In einem solchen Array würden, beispielweise zu dem Suchbegriff „Denken“, alle morphologisch abgewandelten Formen wie „denken“ oder „gedacht“ enthalten sein. Darüber hinaus wären Überlegungen angebracht, inwiefern ein solches Array direkt zum Client

durchgereicht und somit der anchored-Server bezüglich der Rückgabe eines solchen Arrays umgangen werden kann.

Literaturverzeichnis

[1] *Artikel über Ludwig Wittgenstein,*
`https://de.wikipedia.org/wiki/Ludwig_Wittgenstein`

[2] Matthias Lindinger, *Highlighting von Treffern des Suchmaschinentools WiTTFind im zugehörigen Faksimile*, Bachelor-Thesis, 2013.

[3] `http://www.wittgensteinsource.org`

[4] Tobias Trelle, *MongoDB - Der praktische Einstieg*, dpunkt.Verlag, 2014.

[5] Helmut Vonhoegen, *Einstieg in XML, Grundlagen, Praxis, Referenz*, Rheinwerk, 2015.

[6] `https://lxml.de/lxmlhtml.html`

[7] Thomas Hughes-Croucher & Mike Wilson, *Einführung in node.js*, O'Rheilly Verlag, 2012.

Abbildungsverzeichnis

Listings

Inhalt der beigelegten CD

- Elektronische Version der Arbeit (LaTeX)
- Elektronische Version der Arbeit (PDF)
- HTML_Extractor.py
- html_snippets.make (im Witt-data-Repository)
- importHtmlTranscriptionToDB.perl
- database.make (im quadroreader-data-Repository)
- getHtmlAndDoHighlightWithSiglum.js (im Wittfind-Repository)

A. HTML_Extractor.py

Listing A.1: HTML_Extractor.py

```
1  import lxml.html
2  import re
3  import logging
4  import json
5  from collections import OrderedDict
6  import sys
7
8  # die Abarbeitung von Commandozeilenargumenten wird mit [1:] variabel gehalten
9  for file in sys.argv[1:]:
10     #file ist der absolute Pfad der jeweiligen html-Datei
11
12     def check_if_page_break_table_tag(tag):
13         if tag.tag=="table" and tag.get("width") == "100%":
14             return True
15         else:
16             return False
17
18     dateiobjekt = open(file, 'rb')
19
20     htmlString = dateiobjekt.read() # Hier wird alles in einen String umgewandelt
21
22     html = lxml.html.fromstring(htmlString)#Somit kann lxml die HTML optimal mit der eigenen Methode fromstring parsen
23
24     #print(lxml.html.tostring(html, pretty_print=True))
25
26     # Liste aus den AB-Tags. Header oder Metainhalte, usw. sind nicht mehr vorhanden. Nur AB-Tags!
27     mainab_Tags=html.find_class("mainAB")
28
29     #print(mainab_Tags)
30
31     ausgabe_dict=OrderedDict()
32
33     # In der Schleife Abarbeitung der einzelnen AB-Tags
34     for current_main_AB_tag in mainab_Tags:
```

```
35
36          # hier wird der Speicherplatz eines jeden Childelements von mainAB
37          # in der Liste tabellenspalten gespeichert.
38          tabellenspalten=current_main_AB_tag.getchildren()
39
40          #tabellenspalten[0] ist der Speicherort des ersten <tr> </tr> Blocks,
41          #wo auch das zu extrahierende Siglum ist
42          siglenspalte = tabellenspalten[0]
43
44          #Hier wird  ber  den Speicherplatz eines jedes Elements iteriert, das ein div
45                  #ist und dessen Speicherplatz
46          #wird in der Variable siglum_div_tag gespeichert
47          siglum_div_tag=next(siglenspalte.iter('div'))
48
49          # Variable siglum_div_tag hat in jeder Iteration immer nur einen Wert, so dass der Inhalt dieses Speicherorts
50          # mit der Methode text_content() ausgelesen werden kann. Es ist ein Siglum und die date-Angabe
51          roh_siglum_aus_html = siglum_div_tag.text_content()
52
53          # Hier wird das reine Siglum mittels Regex ermittelt
54          roh_siglum_ohne_datum = re.sub(r"\s\(date:␣[0-9]{0,9}\?{0,1}-{0,1}[0-9]{0,9}\?{0,1}\)\s{0,1}\({0,1}[A-Za-z0-9_A-Za-z0-9_\s:]{0,100}\){0,1}", "",
55                                         roh_siglum_aus_html) # Sucht den usdruck und ersetzt den Treffer mit dem zweiten Parameter
56
57
58          inhaltsspalte = tabellenspalten [1]
59
60          dokumentname=roh_siglum_ohne_datum.split("_")[0]
61          roheinzelsiglenstring=roh_siglum_ohne_datum.split("_")[1]
62          roheinzelsiglen=roheinzelsiglenstring.split("et")
63
64          fertige_einzelsiglen=[]
65
66          # Baut die Einzelsiglen
67          for roheinzelsiglum in roheinzelsiglen:
68              teilketten=roheinzelsiglum.split('.')
69              #print(len(teilketten))
70              if(len(teilketten) == 1):
71                  roheinzelsiglum = teilketten[0]
72              else:
73                  roheinzelsiglum=teilketten[0]+"["+teilketten[1]+"]"
74              roheinzelsiglum=dokumentname+","+roheinzelsiglum
75              #print(roheinzelsiglum)
76              fertige_einzelsiglen.append(roheinzelsiglum)
77
78          inhaltstag=current_main_AB_tag.getchildren()[1].getchildren()[5].getchildren()[0].getchildren()[0]
79
80          relevante_tags=inhaltstag.getchildren()
81
82          if len(relevante_tags)==0:
83              logging.warning("Achtung␣keine␣relevanten␣Tags␣gefunden:␣"+ aktuelles_Siglum)
```

```
84              continue
85
86          aktuelles_Siglum=fertige_einzelsiglen[0]
87          counter=0
88          max_counter=len(fertige_einzelsiglen)-1
89          aktuelle_Seite=aktuelles_Siglum.split("[")[0]
90
91
92          ausgabe_dict[aktuelles_Siglum]=""
93
94          for relevanter_tag in relevante_tags:
95              if check_if_page_break_table_tag(relevanter_tag):
96                  #print("Achtung!")
97                  a_tag = next(relevanter_tag.iter('a'))
98                  href=a_tag.get("href")
99                  seite = re.sub(r"http://www.wittgensteinsource.org/", "", href)
100                 seite = re.sub(r"_f", "", seite)
101                 #print(seite)
102                 aktuelle_Seite=seite
103                 if counter < max_counter:
104                     counter+=1
105                     aktuelles_Siglum=fertige_einzelsiglen[counter]
106                     ausgabe_dict[aktuelles_Siglum]=""
107             stringens=str(lxml.html.tostring(relevanter_tag, method="html", encoding="unicode"))
108             stringens=re.sub(r"\n", "", stringens)
109             ausgabe_dict[aktuelles_Siglum]=ausgabe_dict[aktuelles_Siglum]+"_"+ stringens
110
111     ausgabepfad=file + "_snippets.json"
112
113     print("Schreibe_Ausgabe_nach:_"+ausgabepfad)
114
115
116     with open(ausgabepfad, 'w', encoding="utf8") as outfile:
117         json.dump(ausgabe_dict,  outfile, indent=1 , ensure_ascii=False,  )
118     outfile.close()
```

B. importHtmlTranscriptionToDB.perl

Listing B.1: importHtmlTranscriptionToDB.perl

```
1  #!/usr/bin/env perl
2  ## Aufruf: perl importHtmlTranscriptionToDB.perl <DATA_DIR> <LIST OF HTML TRANSCRIPTION JSON-FILES>
3  ## Autor: Behzat Cinar
4
5  use strict;
6  use warnings;
7  use 5.014;
8  use utf8;
9  use open qw(:encoding(utf8) :std);
10
11 use Data::Compare;
12 use Encode;
13 use File::Basename;
14 use File::Spec::Functions;
15 use JSON::PP;
16 use MongoDB;
17 use POSIX qw/strftime/;
18
19 {
20   # Read the config file
21   my $dataDir = shift();
22   #my $configFile = catfile($dataDir, "config.json");
23   my $config = do { local (@ARGV, $/) = $dataDir; decode_json(<>); };
24
25   # Open the database connection
26   my $client = MongoDB::MongoClient->new(host => $config->{"MONGO_HOST"}, port => $config->{"MONGO_PORT"});
27   my $database = $client->get_database($config->{"DATABASE"});
28
29   # Open the log file
30   my $logFileDir = catdir($dataDir, "logs");
31   mkdir($logFileDir);
32   open(LOGFILE, ">>", catfile($logFileDir, "transcription-import.log"));
33   say LOGFILE "################################";
34   say LOGFILE "Log for importHtmlTranscriptionToDB.perl | time: ".strftime('%Y-%m-%d %H:%M',localtime);
```

```
35
36    my $documentId;
37    my $collection;
38    my $entry;
39    my $counter = 0;
40
41    foreach(@ARGV){
42      # Extract the document id and the complete path from the filename
43      ($documentId) = basename($_) =~ /^$config->{"DATA_PREFIX"}(.+?)_/;
44      my $htmlSnippets = do { local (@ARGV, $/) = $_; decode_json(encode_utf8(<>)) };
45
46      $collection = $database->get_collection($documentId."-transcription");
47
48      say LOGFILE "[".$documentId."]_Step_for_importing_the_html_snippets:";
49
50      foreach(sort keys %{$htmlSnippets}){
51        # Check if there is already an entry for the current siglum
52        $entry = $collection->find_one({"siglum" => $_});
53        if(!$entry){
54          $collection->insert_one({
55            "siglum" => $_, "html" => $htmlSnippets->{$_}, "remark-number" => $counter++
56          });
57          say LOGFILE "INSERT_new_siglum_".$_;
58        }else{
59          if(!Compare($htmlSnippets->{$_},$entry->{"html"})){
60            $collection->update_one({"siglum" => $_},{'$set' => {"html" => $htmlSnippets->{$_}}});
61            say LOGFILE "UPDATE_siglum_".$_;
62          }else{
63            say LOGFILE "NOTHING_TO_BE_DONE_for_siglum_".$_;
64
65          }
66        }
67      }
68    }
69
70    # Close the log file
71    close(LOGFILE);
72  }
```

C. getHtmlAndDoHighlightWithSiglum.js

Listing C.1: getHtmlAndDoHighlightWithSiglum.js

```
1
2  // Ein Siglum wird  bergeben  , das dazugeh rige HTML vom Quadroreader gelesen
3  // Diese Funktion w rde in search.js in der Zeile 287 aufgerufen werden.
4  function getHtmlAndDoHighlight(siglum, search_pattern) {
5
6      var qr_host = c_quadroreader["host"];
7      var qr_port = c_quadroreader["port"];
8
9
10     $.get("http://" + qr_host + ":" + qr_port + "/transcription/" + siglum,
11
12         function (htmlContent, search_pattern) {
13
14             //   beginHighlight und endHighlight Tags definieren das Highlighting
15
16         beginHighlight = "<b>";
17         endHighlight = "</b>";
18
19
20         var highlightHtmLContent = "";
21         var i = -1;
22         var lowerCaseSearchPattern = search_pattern.toLowerCase();
23         var lowerCaseHtmlContent = htmlContent.toLowerCase();
24
25         while (htmlContent.length > 0) {
26
27             //i ist-gleich an der 0. Stelle des Suchstrings und die x-te Position im Text
28             i = lowerCaseHtmlContent.indexOf(lowerCaseSearchPattern, i+1);
29
30             if (i < 0) {
31                 //wenn alle search-patterns abgearbeitet wurden:
32                 highlightHtmLContent += htmlContent;
33                 htmlContent = "";
34             } else {
```

```
35                      // Solange das letzte Tag im Dokument nicht erreicht ist:
36                      if (htmlContent.lastIndexOf(">", i) >= htmlContent.lastIndexOf("<", i)) {
37
38                          // script-Block Inhalte werden ignoriert
39                          if (lowerCaseHtmlContent.lastIndexOf("/script>", i) >= lowerCaseHtmlContent.lastIndexOf("<script", i)) {
40                              //hier wird in einem Schleifendurchlauf der Suchbegriff von der 0. Stelle bis zur letzten Stelle des Suchstrings farblich markiert:
41                              highlightHtmLContent += htmlContent.substring(0, i) + beginHighlight + htmlContent.substr(i, search_pattern.length) + endHighlight;
42
43                              htmlContent = htmlContent.substr(i + search_pattern.length);
44                              lowerCaseHtmlContent = htmlContent.toLowerCase();
45                              i = -1; // f r die n chsten Vorkommen des Suchbegriffs wird die Z hlvariable wieder zur ckgesetzt.
46                          }
47                      }
48                  }
49              }
50
51              return highlightHtmLContent;
52          });
53
54
55  }
```