Projekt Big Data: Semantische Suche mit Apache Solr

Eike Nils Knopp , Minh Hieu Nguyen

29. März 2018

Universität Hamburg - MIN Fakultät - Fachbereich Informatik

Betreuer: Andrej Fast, Heinrich Widmann, Julian Kunkel

Teilnehmer 1

Eike Nils Knopp - 5knopp@informatik.uni-hamburg.de - 6809069

Teilnehmer 2

Minh Hieu Nguyen - 5nguyen@informatik.uni-hamburg.de - 6632126

Inhaltsverzeichnis

1. Einleitung

Das Projekt “Big Data” an der Universität Hamburg (in Zusammenarbeit mit dem Deutschen Klimarechenzentrum) beschäftigt sich mit verschiedenen Aspekten des Themenbereichs Big Data. Dabei wird der Focus insbesondere auf drei Gebiete gelegt: die Analyse von bereits bestehenden Daten unter der Verwendung von maschinellem Lernen, die Weiterentwicklung von Werkzeugen aus dem Bereich Data Engineering oder die Leistungsanalyse dieser Werkzeuge und die Umsetzung von Analyseverfahren, Algorithmen oder Visualisierungstechniken. Das Thema “Semantische Suche mit Apache Solr” lässt sich dabei dem zweiten Gebiet zuordnen.

2. Aufgabenstellung

Die Aufgabe unserer Gruppe lautete: Semantische Suche mit Apache Solr – inwieweit kann Apache Solr für die semantische Suche genutzt werden.

Eine semantische Suche beschränkt sich nicht nur auf den Suchbegriff, sondern bezieht auch die Semantik, also die Bedeutung des Begriffs, mit ein. So sind bei Suchanfragen nicht nur Ergebnisse interessant, die den Suchbegriff enthalten, sondern auch Ergebnisse, die Synonyme des Suchbegriffs oder auch andere Varianten des Wortstamms des Begriffs enthalten. (Also Ergebnisse die semantisch Equivalent sind.)

Die Aufgabenstellung gliedert sich dabei in folgende Punkte:

- Erstellung einer Webseite zur Interaktion mit der Search Engine Apache Solr
- Erstellung eines eigenen Solr Servers und Indizierung der zur Verfügung gestellten Daten
- Exploration der Möglichkeiten eine semantische Suche in Solr zu implementieren
- Implementierung und Erweiterung von sinnvollen Features für eine semantische Suche

Als finales Ergebnis wollten wir also eine Website erstellen, mit der es möglich ist eine semantische Suche auf unserem Solr Server durchzuführen.

3. Realisierung

3.1. Synonyme

Als Synonyme bezeichnet man Wörter, die trotz ihrer Unterschiede die gleiche Bedeutung besitzen. Beispielsweise Computer und Rechner. Zwei verschiedene Wörter, die dennoch das gleiche beschreiben. Sie sind also semantisch gleichwertig.

Eine Suche nach sowohl dem Suchbegriff als auch nach Synonymen des Suchbegriffs wird von Solr tatsächlich von Haus aus unterstützt, genannt `solr.SynonymFilterFactory`.

solr.SynonymFilterFactory

Eine Suche nach Synonymen mit Hilfe der SynonymFilterFactory basiert auf einem Dictionary, der `synonyms.txt`. In dieser Datei werden Synonyme auf zwei verschieden Arten definiert:

1. Ersetzen eines Wortes, um zum Beispiel Rechtschreibfehler zu korrigieren:

   ```
   Color    => Colour
   nähmlich => nämlich
   ```

2. Erweitern des Begriffs der Query zu weiteren möglichen Begriffen:

   ```
   GB, Gigabyte, GigaByte, gb, giB
   TV, Fernseher, Röhre
   ```

Dieses Synonyme können sowohl zur Indexzeit als auch zur Queryzeit angewendet werden, wobei der empfohlene Ansatz allerdings die Anwendung zur Indexzeit ist. Die Anwendung zur Queryzeit kann mehrere Probleme erzeugen. Zum einen können Multi-Word Queries nicht als Phrasen verwendet werden, also exakte Suchen, bei denen die Suchbegriffe mit Anführungszeichen umschlossen sind.

Zum anderen werden seltene Synonyme durch das Inverse DocumentFrequency (IDF) Scoring mit einer höheren Gewichtung versehen, wodurch sie in der Ergebnisliste wesentlich weiter vorne stehen als man eigentlich erwarten würde. Das führt zu unintuitiven Ergebnissen. Diese Probleme lassen sich zwar umgehen, wenn man die Synonyme zur Indexzeit anwendet, dies führt allerdings zu einem unnötig aufgeblasenen Suchindex und die Daten müssen jedes Mal neu indiziert werden, sobald die `synonyms.txt` Datei verändert wurde.

Die Möglichkeit eine semantische Suche mit Synonymen zu implementieren ist also durchaus gegeben, es gibt allerdings noch ein weiteres Problem, was diese Implementie-

rung für unseren Anwendungsfall nahe zu unbrauchbar macht: Das Dictionary für die Synonyme muss manuell erstellt werden. Möchte man nur einige wenige Wörter oder Phrasen abdecken, so lässt sich das noch manuell durchführen, hat man aber bereits einen Datensatz mit mehreren Tausend Daten und Millionen von Wörtern, so ist eine manuelle Implementierung des Dictionary für Synonyme nicht mehr realisierbar.

Wir konnten dennoch einen nützlichen Anwendungsfall für Synonyme finden: zur Vereinheitlichung des Sprachfilters. In dem b2find Datensatz sind gleiche Sprachen häufig auf unterschiedliche Weisen codiert. So ist beispielsweise Englisch als `en`, `Eng`, `en_us` oder auch `english` codiert. Dadurch wird der Sprachfilter nicht nur sehr unübersichtlich, da gleiche Sprachen in verschiedenen Versionen angezeigt wird, es führt auch zu Problemen beim tatsächlichen Filtern. Aktiviert man beispielsweise den Filter für `english`, so verliert man alle Ergebnisse, die zwar auch in englischer Sprache sind, aber eine der vielen anderen Codierungen von `english` als Tag besitzen. Und hier kommen die Synonyme zum Einsatz: wir definierten für jede Sprache in den Daten einen entsprechenden Eintrag in unserem Dictionary. Für 15 Sprachen und ca. 3-7 unterschiedliche Codierungen pro Sprache hält sich der Aufwand in Grenzen. So werden bei der Indizierung der Daten alle unterschiedlichen Versionen einer Sprachcodierung auf eine einheitliche Codierung gemappt:

```
anglais,eng,en,en_us,en_uk => english
```

Wobei es allerdings bei Codierungen mit mehreren Wörtern wie zum Beispiel “Modern Greek” wieder zu den bereits erwähnten Problemen der Multi-Word Synonyme kommen kann. So wird nicht nur der Sprachfilter übersichtlicher und einfacher in der Anwendung, auch die tatsächliche Datenbasis wird sauberer, da die Sprachcodierung tatsächlich ersetzt werden und alle Daten einheitlich codiert sind.

Möchte man Synonyme auf die gesamten Inhalte der Datenbasis erweitern, so gibt es dennoch eine Möglichkeit ohne das gesamte Dictionary selber zu verfassen. Öffentliche Thesaurus von Bibliotheken oder auch aus OpenOffice lassen sich auch verwenden, um große Dictionary für Synonyme zu implementieren.

3.2. Word-Stemming

Als Stemming bezeichnet man ein Verfahren, bei dem man verschiedene morphologische (Struktur eines Wortes) Variationen eines Wortes auf einen gemeinsamen Wortstamm zurückführt und so auch von einem Wort auf andere, morphologisch verwandte Wörter schließen kann.

So lassen sich die Worte `fishing` oder `fished` auf den Wortstamm `fish` zurückführen und so auch auf Worte wie zum Beispiel `fisher` schließen. So können mit einer Query nicht nur Singular und Plural eines Suchbegriffs abgedeckt werden, sondern auch Wörter, die den gleichen Wortstamm besitzen und so meistens eine ähnliche Semantik besitzen.

Word-Stemming kann in Solr auf zwei verschiedene Arten implementiert werden. Entweder Dictionary based oder Algorithm based.

Dictionary based:

Eine Variante eines Dictionary-based-Stemming Filters ist der Hunspell-Stemming Filter (solr.HunspellStemFilterFactory). Dieser Filter erzeugt das Stemming eines Begriffs anhand eines Dictionary und definierter Regeln für die entsprechende Sprache. Solch ein Dictionary und die dazugehörigen Regeln werden allerdings nicht von Solr bereitgestellt, sie müssen entweder selbst erstellt werden oder von Drittanbietern erworben werden. Dabei ist die Qualität des Stemmings auch natürlich stark beeinflusst von der Qualität und Größe des Dictionary und der Regeln. Nur entsprechend umfassende Dictionarys und Rule Files führen zu akzeptablen Ergebnissen.

Müssen das Dictionary und die Rules manuell erstellt werden, weil sich kein zufriedenstellendes Material online findet, so ist der Aufwand für eine Implementierung nicht zu vertreten und sprengt für entsprechend große Datenmengen absolut den Rahmen. Für die Datenmengen, mit denen wir uns im Rahmen dieses Projekts beschäftigen, ist das manuelle Erstellen solcher Files keine Option und somit ist der Hunspell-Stemming Filter keine geeignete Option für uns.

Algorithm-based:

Eine Alternative zu Dictionary basierten Stemming Filtern sind Filter, die auf Algorithmen basieren. Diese sind zwar (je nach Dictionary und Rules) meist weniger präzise als auf Dictionary basierende Filter, jedoch sind sie mit erheblich weniger Arbeits- und Wartungsaufwand verbunden.

solr.SnowballPorterFilterFactory:

Der wohl am weitesten verbreitete Stemming Algorithmus ist der Snowball Porter Algorithmus. Er unterstützt gleich mehrere Sprachen, darunter auch English und Deutsch, aber auch Russisch, Spanisch und noch viele weitere. Bei diesem Algorithmus wird das Stemming über Regeln erzeugt, die meist auf die Endungen des zu stemmenden Wortes angewendet werden. Beispielsweise werden die Endungen “e” und “en” abgeschnitten, um aus dem Plural eines Wortes den Singular zu bilden:

```
Tische => tisch
Betten => bett
```

Dies führt allerdings auch dazu, dass auch Wörter, die im Singular auf beispielsweise “e” enden, modifiziert werden:

```
Lampe => lamp
```

Dies geschieht dadurch, dass der Algorithmus das Wort an sich überhaupt nicht in Betracht zieht, sondern sich lediglich auf die Endung beschränkt und die definierten Regeln auf diese anwendet.

Im Kontext unserer Suchmaschine ist dies allerdings gar nicht mal so schlimm, da so, unabhängig von der grammatikalischen Korrektheit des Stemmings, verwandte Worte auf einen gleichen Wortstamm zurückgeführt werden:

```
Lampe -> lamp
Lampen -> lamp
Rechner -> rechn
Rechnen -> rechn
```

Wir erreichen also, trotz grammatikalisch inkorrektem Stemming, alle Wörter des gleichen Wortstammes in unseren Daten. Begriffe, die sich im Plural verändern wie zum Beispiel “Wort” => “Wörter” werden zwar auch korrekt auf den Wortstamm “wort” zurückgeführt, allerdings wird diese Regel auch zum Beispiel auf das Wort “Möwe” => “mow” angewendet. Eskann so also in seltenen Fällen zu unerwünschten Suchergebnissen kommen, wenn zwei semantische völlig unterschiedliche Begriffe so fälschlicherweise den gleichen Wortstamm erhalten.

solr.KStemFilterFactory:

Der KStem Filter ist eine etwas weniger aggressive Variante des Porter Stem Filters und funktioniert auf eine ähnliche Weise, wenn auch mit etwas abgeschwächten Regeln. So erreicht man zwar nicht ganz so viele Daten wiemit dem Snowball Porter Filter, vermeiden allerdings auch falsche Ergebnisse durch inkorrekt gestemmte Begriffe. Dieser Stemming Filter ist allerdings nur für Daten in englisher Sprache verfügbar.

3.3. Result-Clustering mit Carrot2

Ein weiterer interessanter Punkt der semantischen Suche ist das sogenannte Result-Clustering. Dabei werden ähnliche Ergebnisse unter einem gemeinsamen Label gruppiert.

Es ergeben sich thematische Cluster, also Gruppen von Ergebnissen, die sich semantisch ähneln. Das Label für diese Gruppen wird dabei aus den Daten der Ergebnisse erzeugt.

Das Result-Clustering mittels Carrot2 ist bereits in Solr integriert und lässt sich mittels einer Search-Component und einem Request-Handler in der solrconfig aktivieren. Wird nun eine Query gegen den Request-Handler geschaltet, so erhält man zusätzlich zu den Suchergebnissen auch eine Liste mit Labels, der Gewichtung und den zugehörigen Ergebnissen. Anhand dieser Rückgabe lassen sich die Suchergebnisse visuell aufbereiten, beispielsweise in einem Foam-Tree.

Diesen kann man entweder selber implementieren oder man verwendet Implementierung von Carrotsearch (https://carrotsearch.com/foamtree/). Auch wenn diese Option nicht kostenfrei ist, bietet Carrotsearcheine kostenlose Testversion an, mit der wir zurzeit Arbeiten.

Wir haben das Result-Clustering zuerst auf unserem Solr Core mit den Daten des DKRZ getestet, mussten allerdings schnell feststellen, dass die Qualität der Ergebnisse sehr stark von der Art der Daten abhängt. Da die Daten des DKRZ äußerst wissenschaftlich sind, kommt es häufig (je nach Suchbegriff) vor, dass der Carrot2 Algorithmus nur geringfügig geeignete Labels findet. Da die Labels aus den Daten generiert werden, kann beispielswiese aus einem Datensatz mit Zahlen nur ein Label aus Zahlen generiert werden. Inwiefern solche abstrakten Labels für den Anwender hilfreich sind, hängt dann vom Anwender ab.

Als nächstes haben wir den DKRZ Core gegen unseren b2find Core ausgetauscht, um zu sehen, wie sich Carrot2 mit anderen Datenbasen verhält. Obwohl auch diese Datenbasis viele wissenschaftliche Daten enthält, kann Carrot2 hier häufig wesentlich lesbare Labels erzeugen.

3.4. Intuitives Suchfeld

Um ein besseres Sucherlebnis zu ermöglichen, haben wir ein intuitives Suchfeld für die Webseite implementiert. Zusätzlich zu der traditionellen Texteingabe bietet das Suchfeld die Möglichkeit, einfach nach einem bestimmten Feld zu suchen.

Während der Eingabe des Suchbegriffs wird dem Nutzer Vorschläge angezeigt, die nach Kategorien bzw. Feldern zusammengefasst sind (siehe Abbildung 2). Dafür haben wir das von Solr zur Verfügung gestellte Terme-Komponent (engl. “Terms Component”) genutzt. Das Terme-Komponent ermöglicht den Zugriff auf die Terme der Dokumente, die bestimmten Kriterien erfüllen. In unserem Fall werden die Terme angezeigt, die

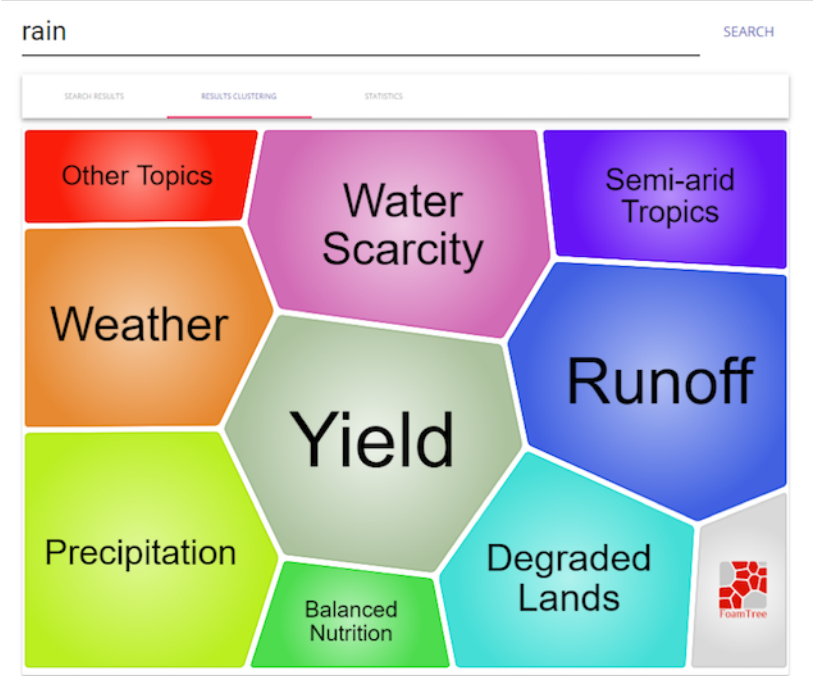

Abbildung 1: Result Clustering mit Carrot2

häufig vorkommen und den aktuellen Suchbegriff als Präfix haben. Es lässt sich ebenfalls andere Kriterien konfigurieren, wie z. B. die Terme die den Suchbegriff enthalten. Diese Konfiguration führt jedoch zu langen Antwortzeiten, da Solr damit den Suchbegriff mit viel mehr Termen abgleichen muss.

Wenn eine Suchklausel nicht ausreichend sein sollte, kann man verschiedene Klauseln miteinander kombinieren, indem man die aktuelle Klausel zu der kombinierten Klausel mittels des + Knopfs neben der Texteingabe hinzufügt. Man kann selbstverständlich die hinzugefügten Klauseln auch wieder entfernen. Wenn man damit fertig ist, kann man die zusammengesetzte Suchklausel wie gewohnt abschicken und die verfeinerten Ergebnisse werden in der Ergebnisliste angezeigt (siehe Abbildung 3).

3.5. Entwicklung der Webseite

Damit Nutzer auf eine geeignete Art und Weise mit Solr interagieren können, benötigen wir eine passende Benutzerschnittstelle. Die Webseite ist dafür entwickelt worden.

Für die Entwicklung der Webseite haben wir entschieden, ReactJS als das Framework einzusetzen, da es eine einfache Strukturierung der Anwendung und ein effizientes UI-Rendering ermöglicht. Außerdem haben wir uns dafür entschieden, dass wir keinen

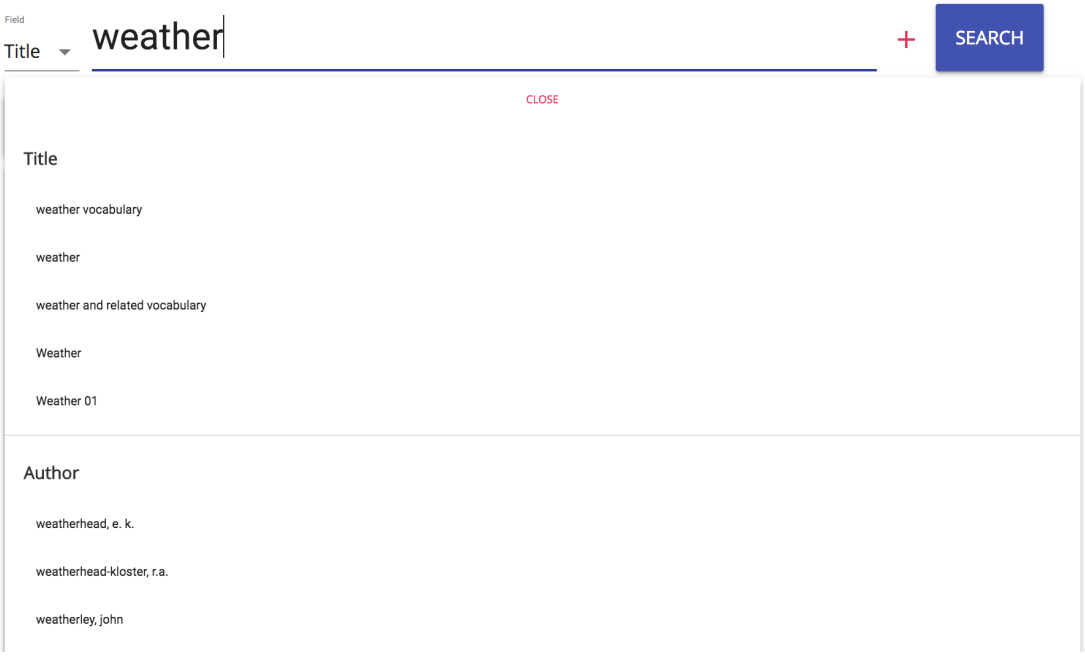

Abbildung 2: Auto-Suggestion

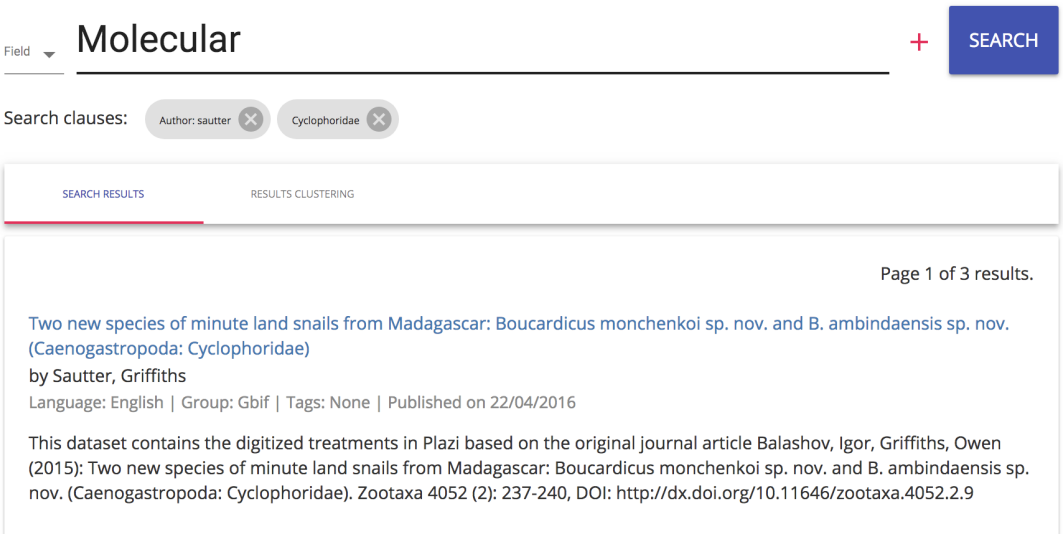

Abbildung 3: Zusammengesete Suchterme

Backend brauchen und arbeiten ausschließlich mit ReactJS. Daher läuft die Webseite auf dem Browser des Nutzers als eine eigenständige Anwendung. Die Entscheidung lässt sich auf die Überlegung zurückführen, dass die Webseite hauptsächlich mit dem Solr Server arbeitet und wir zunächst keine Informationen persistieren oder komplexe Berechnungen durchführen müssen.

Die Anwendung besteht aus folgenden Hauptkomponenten: App, SearchForm, SearchResult, Filter und ResultClustering. Da die Funktionalität der Webseite relativ simpel und übersichtlich sind, werden die Domain-Logik und UI-Logik mehr oder weniger gemischt implementiert.

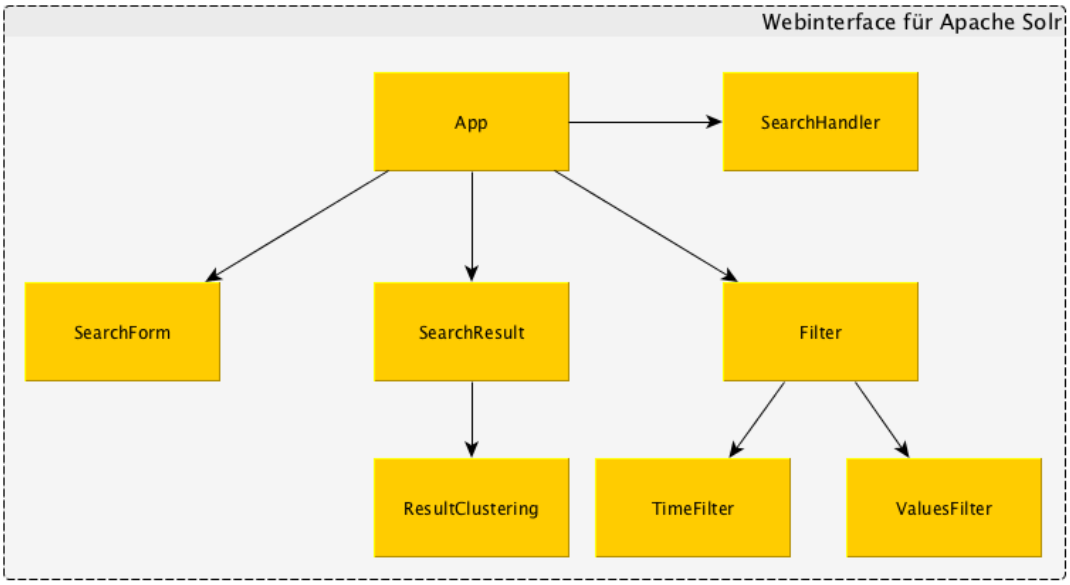

Abbildung 4: Komponente Diagramm

App Komponente

App ist die übergeordnete Komponente, die alle anderen Komponenten enthält und die Kommunikation zwischen diesen koordiniert.

SearchHandler Komponente

Die SearchHandler Komponente ist die zentrale Komponente und auch die einzige nicht-UI Komponente in unserer Anwendung. Sie ist dafür zuständig, mit der REST-Schnittstelle des Solr-Servers zu interagieren bzw. Suchanfragen an den Server zu schicken und die Ergebnisse (also Dokumente und Faceten) vom Server anderen Komponenten zur Verfügung zu stellen.

Andere Komponente können auf Ereignisse reagieren, indem sie einen Callback bei der SearchHandler Komponente registrieren. Wenn z. B. die Ergebnisse vom Server verfügbar sind, wird ein Ereignis ausgelöst und andere Komponente werden darüber informiert und bekommen die entsprechenden Informationen mitgeteilt.

SearchForm Komponente

Die SearchForm Komponente ist dafür zuständig, dem Nutzer ein erweitertes Suchfeld mit Funktionalitäten wie Autosuggestion und zusammengesetzte Suchklauseln anzubieten. Ein passender Ausdruck wird nach einer vorgegebenen Syntax konstruiert und an den SearchHandler übergeben (siehe Abbildung 2 und 3).

SearchResult Komponente

Die SearchResult Komponente übernimmt folgende Aufgaben:

- Suchergebnisse in passender Format anzeigen
- Pagination der Suchergebnisse

Siehe Abbildung 5.

Filter Komponente

Die Filter Komponente ist auch eine wichtige Komponente der Anwendung. Dabei werden die Faceten bei jeder Suchanfrage auf der linken Seite der Webseite aktualisiert und angezeigt (Siehe Abbildung fig:filtercomp). Außerdem ermöglicht diese Komponente die Verfeinerung der Ergebnisse. Bei Änderungen der Filter wird entsprechend ein Ausdruck nach der von Solr vorgegebenen Syntax konstruiert und an dem SearchResult Komponent übergeben.

Bei der Implementierung beschränken wir uns auf zwei Arten von Filtern, Zeit-Filter und Wert-Filter, für die jeweils wiederverwendbare Komponenten implementiert sind. Es geht bei dem Zeit-Filter darum, die Zeitangabe eines bestimmten Feldes zu verfeinern. Exemplarisch haben wir den Filter für das Veröffentlichungsdatum in unserer Webseite implementiert. Werte-Filter ermöglicht die Filterung durch die Auswahl eines konkreten Wertes für Felder, die mehrere mögliche Werte haben. In unserer Webseite sind Filter für Autor, Sprache, Forschungsgruppe und Tag implementiert.

ResultClustering Komponente

Für das Ausprobieren der Result-Clustering-Funktion von Solr haben wir die ResultClustering Komponente gebaut. Zur Visualisierung der Cluster, die von Carrot2 ermittelt

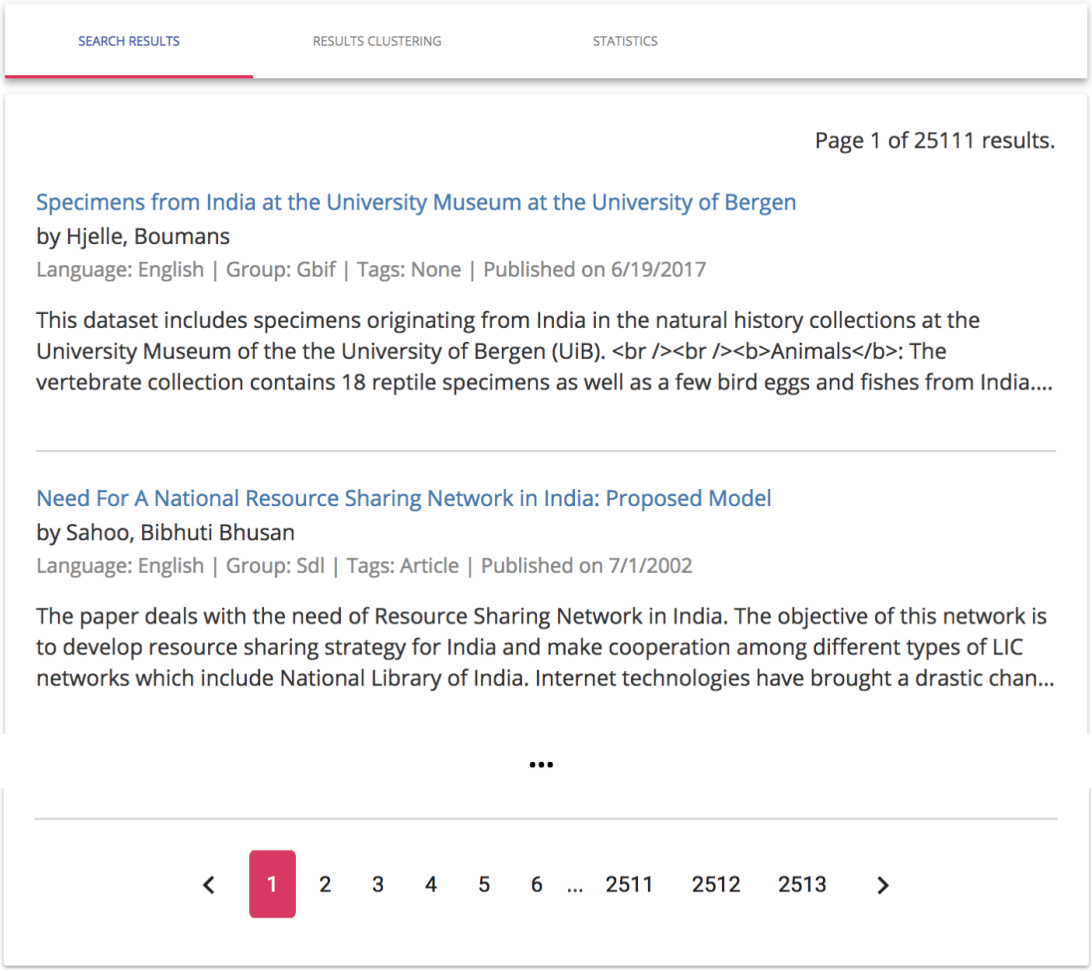

Abbildung 5: Suchergebnisse

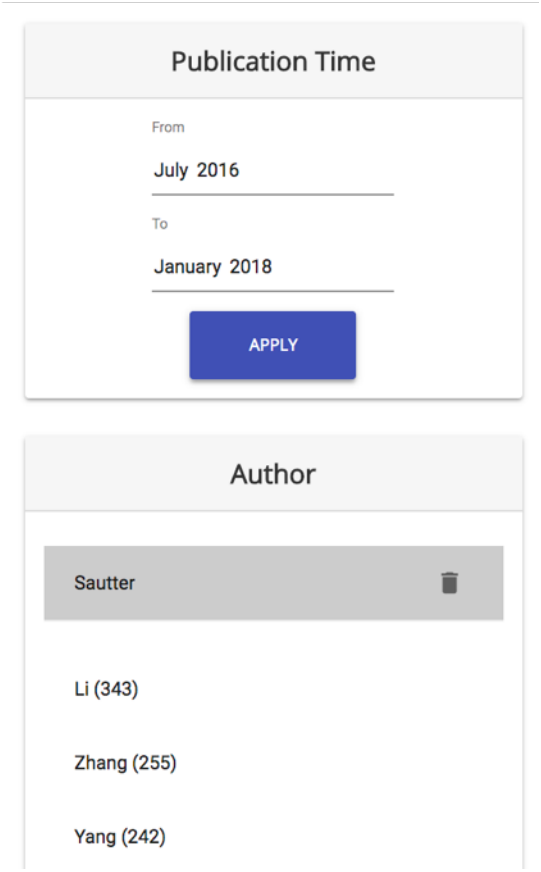

Abbildung 6: Filter Komponente

werden, wird das Foamtree-Plugin eingesetzt (Siehe Abbildung 1). Mehr über Result-Clustering mit Carrot2 findet man in Abschnitt 3.2.

3.6. Probleme bei der Entwicklung

Catch-All-Feld

Eines der ersten Probleme, auf das wir gestoßen sind, war eine Fehlkonfiguration unseres Solr-Schema. Obwohl die Daten korrekt indiziert wurden und auch eine Query mit dem String "*:*" korrekte Ergebnisse lieferte (korrekte Anzahl an gefundenen Dokumenten etc.), wurden für jede andere Query keine Ergebnisse gefunden.

Die Query-Syntax `Feld:Begriff` durchsucht das angegebene Feld nach dem angegebenen Begriff. Kombiniert man das mit der Wildcard * durchsucht die Query so also alle Felder nach allem und liefert so alle Dokumente der Datenbasis zurück. Führen wir nun aber eine normale Query mit nur einem Suchbegriff durch, so geben wir kein spezielles Feld an, in dem nach dem Suchbegriff gesucht werden soll. In solch einem Fall fällt Solr auf ein Default-Search-Field zurück. Dieses hatten wir zu diesem Zeitpunkt noch nicht definiert und die Query lief in die Leere. Man kann allerdings auch nicht einfach eines der bestehenden Felder als Default-Search-Field definieren. Wird so ein Feld als Default-Search-Field definiert, so wird nur innerhalb dieses Feldes nach dem Begriff gesucht. Ist die gesuchte Information in einem anderen Feld, so wird das Dokument nicht gefunden.

Die Lösung für dieses Problem ist ein Catch-All Field: Zur Indexzeit wird der Inhalt jedes Feldes eines Dokuments in ein weiters einzelnes Feld kopiert. Dieses Feld eines Dokuments enthält also die Informationen aller anderen Felder. Dieses Catch-All Field definieren wir nun als Default-Search-Field. Für einfache Queries wird also trotz nicht definiertem Feld alle Felder der Dokumente durchsucht und Solr kann die Ergebnisse entsprechend zurückliefern.

Result-Clustering und FoamTree Plugin

Wie oben schon erwähnt, verwenden wir bei der Visualisierung von Clustern mit Carrot2 das Foamtree Plugin. Jedoch ist die API von Foamtree nicht mit ReactJS kompatibel und kann daher nicht direkt in den React-Code implementiert werden. Wir haben an dieser Stelle ein Trick angewendet, indem wir die Visualisierung mit Foamtree in einer weiteren HTML-Seite mit reinem Javascript-Code implementieren und nun diese Seite mit Hilfe eines iFrames in die React-Komponente einbetten. Die Suchergebnisse werden dann serialisiert und als URL-Parameter der HTML-Seite übergeben, wo sie dann von der API von Foamtree zu einem visuellen Cluster verarbeitet werden.

Felder mit Semikolon-getrennten Werten

Einige Felder enthalten Listen, die als Semikolon-getrennte Werte dargestellt werden, z. B. `author_ss`, `extra_Language`, `tags_ss`. Damit sie in Solr verarbeitet werden können, muss ein neuer Datentyp dafür definiert werden. In dem Schema, das uns als Exemplar zur Verfügung gestellt wurde, wird der `semicolonDelimited` für solche Felder definiert. Jedoch konnten wir nach Anwendung des Datentyps keinen Unterschied bei den Dokumenten sehen. Sie wurden nach wie vor als einzelner, Semikolon-separierter String angezeigt und nicht wie erwartet als Array, in dem die einzelnen Werte abgespeichert sind. Dies stellte uns vor erhebnliche Probleme beim Filtern der Ergebnisse, beispielsweise beim Filtern der Autoren. Mit diesem Problem haben wir uns eine Weile ohne Erfolg beschäftigt. Letzendlich stellte sich heraus, dass die Werte intern tatsächlich anders behandelt werden aber bei einer Anfrage im Original zurückgeliefert werden. Wir mussten also tatsächlich gar nichts tun, um das "Problem" zu lösen. Auch wenn wir einen String von Semikolon-separierten Autoren angezeigt bekamen, konnten wir den als Array behandeln. Das war eigentlich eher eine Erkenntnis als ein Fehler, mit der wir aber einige Zeit verbraucht haben.

Unregelmäßige Sprachen innerhalb eines Dokuments

Ursprünglich wollten wir zur vereinheitlichung der Sprachtags ein Tool zur automatischen Erkennung der Sprache verwenden. So müssten wir nicht mehr auf die unsauberen Tags der Dokumente zurückgreifen, sondern könnten diekt aus dem Inhalt der Dokumente die jeweilige Sprache ermitteln. Wir mussten allerdings feststellen, das auch innerhalb eines Dokumentes die Sprache von Feld zu Feld variieren kann und es so nicht möglich ist, ein einheitliches Feld zur Analyse der Sprache des Dokumentes festzulegen, was für das Spracherkennungstool notwendig gewesen wäre. Wir mussten also diesen Ansatz vrwerfen und verwendeten stattdessen den in 3.1 bereits beschriebenen Ansatz, die unsauberen Sprachtags mithilfe von Synonymen auf saubere, einheitliche Tags zu mappen.

4. Setup-Anleitung

Daten in Solr importieren

1. Ein neuen Core mit dem Namen `b2find` durch die Solr-Webapp oder mit dem CLI erstellen:

   ```
   bin/solr create -c dkrz
   ```

   Ein Verzeichnis für den Solr Core wird anschließend unter `<SOLR_INSTALLATION>/server/solr/b2find`

angelegt, wo der Index sowie die Konfigurationen Datein abgelegt sind.

2. Die `solrconfig.xml`-Datei aus dem Projekt (`<PROJEKT_ORDNER>/b2find-conf/solrconfig.xml`) zu `<SOLR_INSTALLATION>/server/solr/dkrz/conf/solrconfig.xml` kopieren oder verlinken.
3. SQLite JDBC Dependency `sqlite-jdbc-3.21.0.jar` zu `/dist` hinzufügen (siehe `solrconfig.xml`).
4. Config-Datei für Data-Import Handler `<PROJEKT_ORDNER>/b2find-conf/DIHconfigfile.xml` zu `solr-7.1.0/server/solr/dkrz/conf` kopieren oder verlinken.
5. Den Pfad zur Datenbank-Dump-Datei (in unserem Fall `b2finddump.db`) in `DIHconfigfile.xml` anpassen.
6. Die Solr Instanz neustarten.
7. Die Daten im Solr-Webapp indizieren bzw. importieren lassen.

Anmerkung:

- Während der Entwicklung verwendeten wir die Solr Version 7.1.0.
- Die Solr-Instanz solle auf dem Port 8983 laufen.

Webseite starten

1. Zum Hauptverzeichnis der Anwendung unter `<PROJEKT_ORDNER>/app` navigieren
2. `npm start` ausführen, um die Anwendung zu starten
3. Die Webseite unter `http://localhost:3000` aufrufen.

Literaturangaben

1. Potter, Grainger (2014): Solr in Action, Manning
2. Apache Solr Reference Guide 7.1 - https://lucene.apache.org/solr/guide/7_1/index.html
3. Stemming and Lemmatisation with Apache Solr - http://thinknook.com/keyword-stemming-and-lemmatisation-with-apache-solr-2013-08-02

4. Result Clustering - https://lucene.apache.org/solr/guide/7_1/result-clustering.html
5. ReactJS Documentation - https://reactjs.org/docs/introducing-jsx.html

Genutzte Software / Hardware während des Projekts

Konfiguration von Solr & Entwicklung der Webseite

- Betriebssystem: macOS, Linux
- Web Browser: Google Chrome
- Entwicklungstool: Terminal, Vim-Editor
- Software & Bibliotheken: Apache Solr, ReactJS, Material-UI

Erstellung des Berichts

- Microsoft Word zur Bearbeitung des Berichts
- Latex zur Fertingsstellung des Berichts