Institute for Web Science and Technologies - WeST

Matching keyword based queries to schema level indices over Linked Open Data

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von
Tobias Hauck

Erstgutachter: Prof. Dr. Steffen Staab
Institute for Web Science and Technologies – WeST

Zweitgutachter: Dr. Thomas Gottron
Institute for Web Science and Technologies – WeST

Beginn der Arbeit: 22.05.2014
Abschluss der Arbeit: 21.11.2014

Erklärung

„Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet–Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung der Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Im Rahmen dieser Arbeit soll eine Methodik erarbeitet werden, die englische, keyword-basierte Anfragen in SPARQL übersetzt und bewertet. Aus allen generierten SPARQL-Queries sollen die relevantesten ermittelt und ein Favorit bestimmt werden. Das Ergebnis soll in einer Nutzerevaluation bewertet werden.

Inhaltsverzeichnis

1. Einleitung....1
1.1 Zielsetzung....1
1.2 Vorarbeiten....2
1.2.1 Übersetzung von natürlicher, englischer Sprache in Abfragesprachen....2
1.2.2 SchemEX....5
1.2.3 Stop Words....5
1.2.4 Lucene / Solr....6
2. Methodik....7
2.1 Datenbasis....7
2.2 Tokenisierung....8
2.3 Retrieval....10
2.4 Erstellen der SPARQL-Abfragen....12
2.5 Bewertung....17
3. Umsetzung....22
3.1 Aufbauen des Index....22
3.2 Datenbasis....23
3.3 Tokenisierung....25
3.4 Retrieval....25
3.5 Erstellen der SPARQL-Abfragen....26
3.6 Bewertung....28
4. Evaluation....29
4.1 Vorgehensweise....29
4.2 Auswertung....30
5. Fazit....37
Quellen....39
Anhang....41
Werkzeuge....41
Symfony2....41
PHP....41
A1: Erstellen von SPARQL-Queries mit Kantenlänge 1....42
A2: Implementation Tokenisierungsfunktion....43
A3: Implementation Retrievalfunktion....44
A4: Implementation Erstellung SPARQL-Pattern....45
A5: Implemenation Erstellung von Queries....46
A6: Implemenation Triple-Pattern zu SPARQL-Queries....47
A7: Implementation Lodatio-Request....48
A8: Implementation Bewertungsfunktion....49
A9: Nutzertest....50

1. Einleitung

Im Internet gibt es mit Linked Open Data frei verfügbare Daten, die direkt über HTTP per URI angesprochen werden können. Diese Daten können mit weiteren URIs verbunden sein und bilden einen Graphen. Die URIs werden durch Browsen erreicht. Alternativ können die URIs jedoch auch über das Suchen mit Keywords oder über das Suchen mit SPARQL-Queries abgefragt werden. Diese Arbeit fokussiert sich auf das Suchen mit Keywords.
Für allgemeine Suchmaschinen gilt, dass Nutzer es gewohnt sind, dass ihre Keyword-Abfragen nicht immer die gewünschten Resultate erzielen. Das liegt daran, dass die Eingaben falsch interpretiert werden. Je besser die Eingaben interpretiert werden können, desto höher ist die Wahrscheinlichkeit, dass die Antworten präzise ausfallen. Daher ist das richtige Interpretieren von Keyword-Abfragen ein wichtiger Meilenstein für die Beherrschbarkeit von Informationen.

1.1 Zielsetzung

Diese Bachelorarbeit soll einen Beitrag zur Forschung an der Interpretation solcher Keyword Abfragen leisten und deren Interpretation in der Abfragesprache SPARQL ausdrücken. Sie soll eine Methode entwickeln, die dem Nutzer ermöglicht, über Keyword-basierte, englische Anfragen entsprechende SPARQL-Abfragen zu generieren. Zunächst soll die Eingabe in einzelne Keywords zerlegt werden. Die Keywords sollen anschließend Schema-Elementen zugeordnet werden. Damit das möglich ist, sollen die Rohdaten von Linked Open Data für Lucene aufbereitet und parallel ein Schema-Index erzeugt werden. Mit Hilfe von Lucene sollen dann Schema-Elemente und deren URI erfragt werden, deren Daten Informationen über das Keyword enthalten. Aus den Ergebnissen von Lucene sollen SPARQL-Abfragen erstellt werden, die mit Hilfe von Lodatio[1] SchemEx[2] anfragen. Die möglichen Interpretationen der Anfrage als SPARQL sollen nach verschiedenen Ranking-Strategien priorisiert werden, zum Beispiel nach deren Ergebnismenge. Die Abfrage mit der höchsten Priorität soll ausgegeben werden. Alternativ sollen auch andere Abfragen angeboten werden, für den Fall, dass die höchst priorisierte Antwort nicht die richtige Antwort war.
Die Keywords sollen auf einer Webseite eingegeben und die Ergebnisse auf dieser ausgegeben werden. Die Qualität der Ergebnisse soll in einer Nutzerstudie evaluiert werden.

1.2 Vorarbeiten

Sowohl für die Übersetzung von natürlicher, englischer Sprache nach SPARQL als auch für das Anfragen von SPARQL-Queries an Linked Open Data gibt es Vorarbeiten.

1.2.1 Übersetzung von natürlicher, englischer Sprache in Abfragesprachen

Ein Forschungsschwerpunkt ist, die einzelnen Wörter der Abfrage zu zerteilen und anschließend mittels statistischer Methoden zu analysieren und gleichzeitig ein komplexes Wörterbuch nach den Wörtern zu durchsuchen, um herauszufinden, um welche Art von Wort (Subjekt, Prädikat, Objekt) es sich handelt und wie es mit der Frage in Zusammenhang stehen müsste [3]. Teilweise wird dabei auch “Query Segmentation“ [4] und “Query Cleaning” [5] angewandt. “Query Segmentation” ermittelt Wörter, die aus mehreren Wörtern bestehen und interpretiert sie dann als einen Wortblock, der nicht zerteilt werden darf. “Query Cleaning” versucht, relevante Wörter zu ermitteln und unwichtige Wörter zu entfernen. Der Ansatz erzielt für Abfragen, die Grundregeln der natürlichen Sprache beachten, hohe Erfolgsquoten, aber es fehlt die Übertragbarkeit auf Abfragen, deren elementare Sprachelemente fehlen, zum Beispiel ein Prädikat.

Ein anderer Forschungsschwerpunkt ist das Ranking [6, 7]. Beim Ranking werden die zu den Keywords passenden Elemente gesucht und miteinander in Beziehung gesetzt, um danach zu ermitteln, wie relevant ein Ergebnis ist. Die Relevanz berechnet sich nach vorgeschriebenen Mustern, zum Beispiel der Anzahl von Vorkommnissen. So könnte ein Übersetzungswerkzeug beispielsweise mit Hilfe des Rankings ermitteln, wie oft die zur Wahl stehenden möglichen Übersetzungen in der entsprechenden Zielsprache existieren und danach Aussagen darüber treffen, mit wie hoher Wahrscheinlichkeit eine bestimmte Übersetzung korrekt ist. Im Ranking spielen Wahrscheinlichkeitsverteilungen die zentrale Rolle.

Ein dritter Forschungsschwerpunkt ist die Suche nach möglichen Abfragekandidaten auf Schema- und Datenebene, um eine effiziente Suche auf in Graphen strukturierten Daten, zum Beispiel RDF, zu erreichen [8, 9]. Dabei werden die zugrunde liegenden Daten nach den einzelnen Keywords durchsucht und dann versucht über das Schema ihre Beziehung zu ermitteln. Die daraus resultierenden Teilbäume können über Algorithmen wie den STAR Algorithmus noch weiter vereinfacht werden [10]. STAR sucht in gegebenen Teilbäumen in einem Greedy-Verfahren unter Einbezug der Entity-Ebene nach den kürzesten Pfaden zwischen den Fixknoten und eliminiert anschließend alle Kanten, die Teil eines längeren Pfades mit dem selben Zielknoten sind. STAR bewertet die schrittweise vereinfachten Teilbäume jeweils über die Summe der Gewichte entlang der Kanten, wobei

eine niedrige Summe erreicht werden soll. Nach der Vereinfachung werden die Teilbäume auf ihre Pfadlänge, ihre relative Anzahl von Daten, die sie repräsentieren und die Anzahl der übereinstimmenden Keywords gewichtet. Je höher die Gewichtung, desto höher ist die Wahrscheinlichkeit, dass das Ergebnis relevant ist.
QUICK [11] ist ein konkreter Ansatz auf Schema-Ebene. Es wird versucht, aus einer Keyword-Abfrage konkrete SPARQL-Abfragen zu bilden. Der Unterschied zu dem Ansatz in dieser Arbeit ist, dass der Nutzer in den Erstellungsprozess mit einbezogen wird und somit in gewissem Grad eingeschränkt wird. Er wählt den richtigen Teilgraphen aus einer Menge von generierten Teilgraphen aus. Der Nachteil daran ist, dass der Nutzer ein grundlegendes Verständnis des semantischen Webs benötigt, um das QUICK User Interface zu bedienen.

Neben den rein schema-basierten Ansätzen gibt es auch eine Reihe von teilweise schema-basiert, teilweise daten-basierten Ansätzen. Daten-basiert bedeutet hierbei, dass zu einer Keyword-Abfrage nach konkreten Daten gesucht wird, die das Keyword enthalten.
Sqak[12] zum Beispiel ist ein hybrider Ansatz zwischen Schema- und Datenebene. Er sucht nach konkreten Daten zu einer Abfrage. Die Abfrage wird in einer eigenen Abfragesprache, rSQL, also "reduced SQL", gestellt. Dadurch ist es Sqak möglich, zu erkennen, welche Teile der Anfrage Schema-Elemente, zum Beispiel Datenbanktabellen, betreffen und welche Teile Eigenschaften konkreter Daten erfragen, zum Beispiel deren Namen. rSQL soll zudem das Bedienen vereinfachen und auch solchen Personen ermöglichen, die Abfragesprachen wie SPARQL oder SQL nicht verstehen. Die Funktionsaliase haben sprechendere Namen und fühlen sich natürlichsprachlich an. Der User kann jedoch nicht einfach wahllose Eingaben tätigen, sondern muss sich an die Syntax der Eingabesprache halten.
Sqak basiert auf älteren Arbeiten an Systemen wie BANKS[13], DISCOVER[14] und DBXplorer[15]. Diese Systeme bieten eine uneingeschränkte, keyword-basierte Suche auf reiner Datenebene an. Zu einer Eingabe werden alle zugrunde liegenden Daten mittels Algorithmen performant durchsucht und diejenigen Ergebnisse zurückgegeben, die die Keywords am meisten enthalten. Sqak versucht mit rSQL die Ausdrucksstärke zu erhöhen und büßt dabei die Freiheit und damit die Bedienbarkeit ein.
Ähnliche Ansätze wie Sqak, weitaus komplizierter, dafür aber ausdrucksstärker sind Lösungen wie FlexPath[16] oder die direkte Verwendung von SPARQL oder SQL.

Grundsätzlich steht jeder Ansatz immer vor dem Problem, dass bei steigender Bedienbarkeit die Ausdrucksstärke leidet. Das folgende Diagramm, das dem Paper von Sqak entnommen ist, zeigt diese Problematik und ordnet einige genannte Ansätze in dieses Diagramm ein.

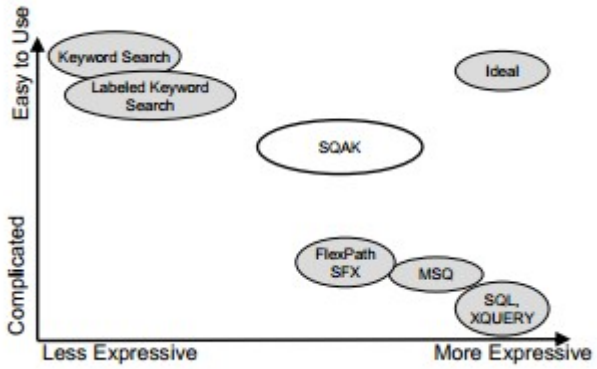


Abbildung 1: Vergleich verschiedener Ansätze: Bedienbarkeit vs. Ausdrucksstärke

Diese Bachelorarbeit soll den Ansatz der Suche nach möglichen Abfragekandidaten auf Schema- und Datenebene verfolgen. Die Keywords sollen ebenfalls durch Trennen an Leerzeichen ermittelt werden. Anschließend werden die Keywords mit Hilfe des Keyword-Indexes Lucene konkreten Schema-Elementen von Linked Open Data zugeordnet. Danach werden SPARQL-Abfragen aus den ermittelten Schema-Elementen erstellt, indem die erhaltenen Schema-Elemente über das zugrunde liegende Schema als SPARQL-Abfrage miteinander in Beziehung gesetzt werden. Die Ergebnisse sollen über eine Kostenfunktion bewertet werden. Diese Kostenfunktion soll Pfadlänge, relative Anzahl von Daten, die der Teilbaum repräsentiert und die Anzahl übereinstimmender Keywords gewichten und anschließend zu einer einzigen Bewertungsgröße verrechnen. Die drei Abfragen mit der höchsten Bewertung sollen über Lodatio auf den Index SchemEx nach ihrer Bewertung priorisiert ausgeführt werden. Die Abfrage mit der höchsten Bewertung wird ausgegeben, die beiden anderen hoch bewerteten Antworten werden als alternative Lösungen angeboten.

Die Bedienbarkeit des Ansatzes ist sehr leicht, die Ausdrucksstärke richtet sich nach dem zugrunde liegenden Index. Enthält der Index detaillierte

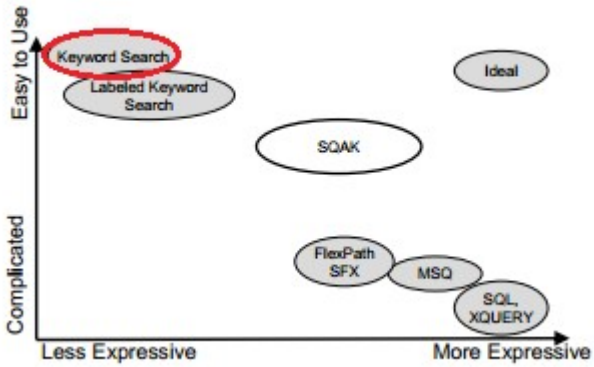


Abbildung 2: Einordnung des Ansatzes dieser Arbeit

Daten, kann er auch Anfragen nach Anzahlen, zum Beispiel "number of rooms of Hotel", beantworten. Insgesamt ist der Ansatz jedoch eher ausdrucksschwach und lässt sich daher in die Keyword Search einordnen.

1.2.2 SchemEX

SchemEX ist ein Index auf Linked Open Data [17]. Er dient dazu, verschiedene Datenquellen und Instanzen aus Linked Open Data für eine gegebene SPARQL-Anfrage zu lokalisieren. Er ermöglicht zum Beispiel das Finden von Instanzen eines bestimmten Typs oder von Typen, die über Properties in Relation zu einem anderen Typen stehen. Ermöglicht wird dies, indem die RDF-Instanzen von Linked Open Data auf ihr konzeptionelles Schema reduziert werden. Alle Schema-Elemente sind mit den Datenquellen und deren Instanzen verknüpft. Wird ein Schema-Element angesprochen, weiß der Index somit gleichzeitig, welche Daten dem Element zugeordnet sind. SchemEX ist damit in der Lage, zu einem konkreten SPARQL-Query alle relevanten Datenquellen in der Linked Open Data Cloud zu finden. Anders ausgedrückt: SchemEX ermöglicht eine graphbasierte Suche auf Instanzen der LOD Cloud.

Der SchemEX kann über die Plattform Lodatio angesprochen werden. Lodatio bietet sowohl ein Webinterface als auch eine Web-API an.
Das Webinterface erwartet als Eingabe einen SPARQL-Query und gibt als Ergebnis ein HTML-Dokument zurück. Das HTML-Dokument bietet alternative SPARQL-Queries an und gibt für den eingegebenen Query eine Auflistung von URIs oder Datenquellen, die Informationen zu der Anfrage enthalten. Zu jeder URI ist angegeben, wie viele Instanzen sie enthält. Zusätzlich werden unter der Ergebnisliste verwandte Queries aufgelistet. Das Webinterface wird in dieser Arbeit benötigt, um SPARQL-Queries unkompliziert auszuprobieren.
Die Lodatio API kann über HTTP GET angesprochen werden und benötigt als Parameter den Query *query*. Das Ergebnis der Anfrage erfolgt im JSON-Format und enthält alle relevanten URIs und deren Instanzmenge. Die API wird verwendet, um automatisch generierte SPARQL-Anfragen auf die Menge ihrer gefundenen Datenquellen und Instanzen auszuwerten.

1.2.3 Stop Words

Stop Words sind Wörter, die im täglichen Sprachgebrauch verwendet werden, aber keine Aussage über den eigentlichen Content machen. Anders als die bekannten Füllwörter umfassen Stop Words beispielweise auch Präpositionen und Artikel. Sie werden bei der Volltextindizierung nicht beachtet, weil sie "*keine Relevanz für die Erfassung des Dokumentinhalts besitzen*"[1].
Für das Entfernen von Stop Words aus Keyword-Abfragen gibt es Vorarbeiten. Einige verfolgen den Ansatz, Stop Word – Listen zu verwenden

1 http://de.wikipedia.org/wiki/Stoppwort

und die Eingabe auf Wörter in diesen Listen zu untersuchen. Grundsätzlich gilt dabei: Je größer die Stop Word – Liste, desto besser.
Neben den statischen Listen gibt es dynamische Ansätze für das Entfernen von Stop Words. Ein interessanter, statistischer Ansatz ist das Herausfinden von Stop Words über Verteilungen[18]. Der Ansatz setzt voraus, dass jede Sprache Stop Words besitzt und diese in einem gewissen prozentualen Verhältnis zur Gesamtmenge an Wörtern stehen. Hat ein Wort dieses Verhältnis, wird es als Stop Word erkannt und entfernt. Dieser Ansatz setzt voraus, dass man eine große Datenmenge ganzer Sätze einer Sprache besitzt, die grammatikalisch richtig sind. Über diese Daten kann der Stop-Word-Algorithmus angewandt werden und liefert eine Stop Word – Liste für die entsprechende Sprache, die man anschließend verwenden kann.

In dieser Arbeit wird eine Stop Word - Liste verwendet[2].

1.2.4 Lucene / Solr

"*Lucene ist eine Programmbibliothek zur Volltextsuche und ein Projekt der Apache Software Foundation*" [3].
Lucene ist in Java geschrieben und frei erhältlich. Da es sich bei Lucene lediglich um eine Programmbibliothek handelt, gibt es kein User-Interface. Lucene ist die Grundlage einiger weiterer Programmbibliotheken zur Volltextsuche, so zum Beispiel Solr, das in dieser Arbeit verwendet wird. Solr bietet ein User-Interface an und ist mit PHP leichter ansprechbar als Lucene. Solr ist speziell für PHP konzipiert.
Solr wird als Index zur Suche nach Schema-Elementen verwendet. Zu gegebenen Keywords werden entsprechende Einträge per Volltextsuche herausgesucht. Die Einträge enthalten Informationen über den Elementtypen und die URI. Ohne Volltextsuche wäre der Ansatz dieser Arbeit deutlich schwerer umsetzbar. Daher kann Lucene / Solr durchaus als Vorarbeit angesehen werden.

2 http://norm.al/2009/04/14/list-of-english-stop-words/
3 http://de.wikipedia.org/wiki/Apache_Lucene

2. Methodik

Dieses Kapitel beschreibt die Methodik, mit der aus einer Keyword-Anfrage SPARQL-Queries generiert und bewertet werden und aus allen Queries ein Favorit ausgewählt wird. Dazu wird zunächst die Datenbasis formal beschrieben. Anschließend wird gezeigt, wie die Eingabe tokenisiert wird. Im nächsten Abschnitt wird ausgeführt, wie den Tokens mit Hilfe der Retrievalfunktion Schema-Elemente zugeordnet werden. Aus der Menge an Schema-Elementen werden zunächst Triple-Pattern gebildet und aus den Triple-Pattern SPARQL-Queries. Diese werden dann schließlich bewertet und der relevanteste Query andhand der Bewertung ermittelt.

2.1 Datenbasis

Die abzufragenden Daten sind eine Menge von Dokumenten, die in einem Index gespeichert sind. Ein Dokument beschreibt genau ein Schema-Element und enthält neben der URI, also dem Schema-Element Bezeichner, noch einen Beschreibungstext sowie den Typen rdf:type oder rdf:property. Die Daten sind aus dem SchemEx-Datensatz entnommen. Es wurden alle URIs von Classes und Properties herausgefiltert und die durch die URIs identifizierten Ressourcen heruntergeladen. Eine Ressource enthält mehrere Beschreibungstexte und einen Typen rdf:type oder rdf:property. Da es verschiedene Beschreibungstexte gibt, kommen die URIs öfter als einmal im Index vor – jedoch immer mit einem anderen Beschreibungstext. Wir kürzen ein Dokument im Folgenden mit d ab.
Die Beschreibungstexte sind nicht zusammengefasst, da das Dokument d am Anfang der Bearbeitungszeit noch ein weiteres Attribut "Subtype" besaß. Das Attribut war für Beschreibungstexte verschieden. Weil nicht klar war, ob dieses Attribut für die Methodik interessant werden konnte, wurde es mit aufgenommen und damit dann auch die Aufteilung der Beschreibungstexte auf verschiedene Dokumente. Es stellte sich aber heraus, dass dieses Attribut nicht benötigt wurde. Die Dokumentstruktur wurde jedoch beibehalten, weil die Arbeit schon zu weit fortgeschritten war. Es ist durchaus möglich, die Beschreibungstexte zusammenzufassen.

Ein Dokument besteht aus einer ID, einer URI, einem Beschreibungstext und einem Typen. Der Typ ist entweder "Class" oder "Property".

$$d := (\underline{id}, URI, description, type)$$

Die Menge aller Dokumente d bildet die Gesamtmenge $D := \{d_1, d_2, \ldots d_n\}$.
Es gilt $d \in D$.

Da ein Dokumententyp entweder "Class" oder "Property" sein kann, klassifizieren wir zwei verschiedene Dokumentenmengen:

C ist die Menge von Dokumenten D, die von Typen "Class" sind, also $C(\underline{id}, URI, description, type = Class)$

P ist die Menge von Dokumenten D, die von Typen "Property" sind, also
$P(\underline{id}, URI, description, type = Property)$

C und P bilden gemeinsam die Menge $D = C \vee P$.

2.2 Tokenisierung

Die Eingabe i ist zunächst einmal lediglich ein String $i : String$.
i wird tokenisiert.
Die Tokenisierung erfolgt durch Trennen an Leerzeichen.

Die Tokenisierungsfunktion T produziert aus dem Eingabestring i eine Menge von Keywords W mit $w_1, w_1, \ldots, w_n \in W$, also $T(q): q \rightarrow P(W)$.

In Abbildung 3 sowie einigen weiteren Abbildungen wird die Notation x:y verwendet. x kann mit i oder w belegt sein. i steht für einen Eingabestring und w für ein Keyword. y stellt den Inhalt dieser Variablen dar.

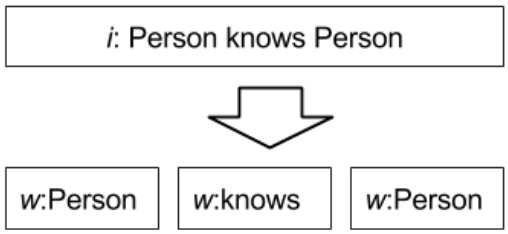


Abbildung 3: Tokenisierungsfunktion T

Zusammengesetzte Wörter, durch Leerzeichen getrennt

Nicht jedes Keyword w ist ein eigenständiger Begriff. Zusammengesetzte Wörter werden durch Leerzeichen getrennt aufgeschrieben, so z. B. "state of" oder "handball league". Das ist ein sprachliches Problem. Hinzu kommt, dass im Fall von "state of" das zweite Wort ein Stop Word ist, im Fall von "handball league" dagegen nicht. Insofern können zusammengesetzte Wörter Stop Words enthalten, müssen dies aber nicht. Die URIs, also die Bezeichner der Schema-Elemente, sind oft sprechend, enthalten aber keine Leerzeichen. URIs von Properties wie "state of" werden zusammen geschrieben, z. B. foaf:stateof. Ziel ist es, dass solche Wörter vor der

eigentlichen Verarbeitung entdeckt und durch eine Komposition von Wörtern ersetzt werden. Dabei sollen die Stop Words nicht entfernt werden, denn im Falle von "state of" zum Beispiel würde das Entfernen dazu führen, dass nach dem Type "state" gesucht wird und nicht nach der gewünschten Property "stateof". Damit würden Informationen entfernt werden, die noch benötigt werden. Trotzdem sollten zusammengesetzte Wörter vor der eigentlichen Verarbeitung des Queries erkannt und ersetzt werden, um die Performance zu verbessern und um zu verhindern, dass keine ähnlichen Schema-Elemente statt dem eigentlich richtigen Element ausgewählt werden.
Im besten Fall sollte aus einer Eingabe "country state of" unter der Annahme, dass es einen Type "Country" und eine Property "stateof" gibt, die Eingabe "country stateof" werden, die dann als "originaler Eingabestring" i weiter verwendet wird.

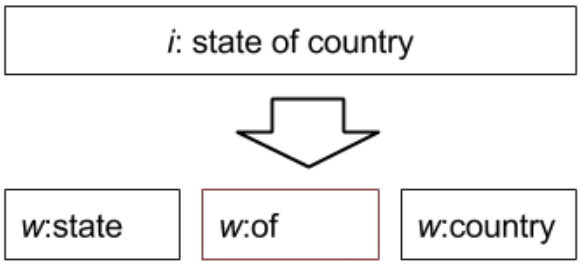


Abbildung 4: Tokenisierung: Problem Stop Word

Das Problem soll zunächst auf Schema-Ebene gelöst werden. Dazu bereitet die Tokenisierungsfunktion die Keywords so weit auf, dass sie nur noch solche Wörter enthält, die rein sprachlich einen einzigen Begriff darstellen. Begriffe, die danach immer noch sprachlich zusammen gehören, werden im weiteren Verlauf durch Scoring und andere Mechanismen eliminiert.

Das Verbinden von Wörtern erfolgt mit Hilfe der Funktion $T_C(W): W \to W_C | \{w_1 \, concat \, w_2, w_2 \, concat \, w_3, \ldots, w_{n-1} \, concat \, w_n\} \in W_C$.
Die Funktion *concat* verbindet zwei Strings zu einem String. Die Funktion T_C verbindet somit jeweils benachbarte Keywords miteinander und speichert sie in W_C . Danach wird nach möglichen Classes oder Properties gesucht, die Elemente von W_C beinhalten. Ist dies der Fall, werden die beiden einzelnen Keywords k_x und k_y durch das neue Keyword w_{neu} ersetzt, also $\{w_x, w_y\} \to w_z, w_z = w_x \, concat \, w_y$.

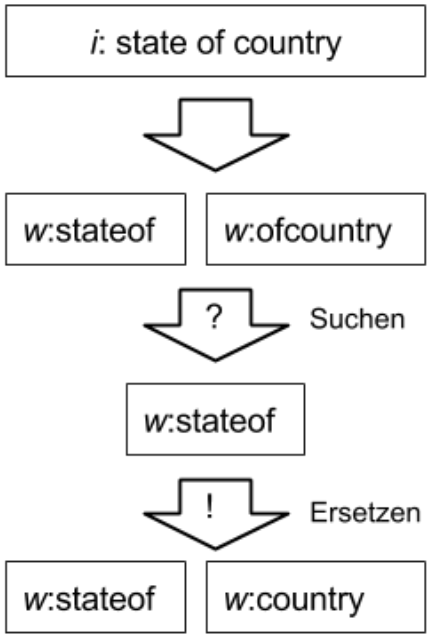


Abbildung 5: Präventivlösung Stop Words

$T(q) := \{w_1, w_2, ..., w_n\}$ gilt auch dann noch, wenn ein Keyword w durch ein kombiniertes Keyword w_C ersetzt wurde, da die vorher getrennten Keywords auch genau ein logisches Keyword bildeten.

Anschließend wird eine Stop Word – Liste eingesetzt, die alle restlichen Stop Words entfernen soll. Es werden alle Wörter gesucht, die in der Stop Word – Liste vorkommen und dann aus der Eingabe i entfernt.

2.3 Retrieval

Die Retrievalfunktion filtert eine Menge von Dokumenten durch Anwendung eines Queries. Die allgemeine Retrievalfunktion ist definiert als $S: Q \times D \rightarrow [0, \infty]$.

Die Retrievalfunktion R_q ist die Retrievalfunktion zu einem konkreten Query q . Das bedeutet, dass es zu jedem Query q eine Retrievalfunktion R_q gibt, die Dokumente D_q zurückgibt, die in der Antwortmenge von q enthalten sind.

Die Retrievalfunktion, die jeweils nur Dokumente vom Typ "Class" zurückgibt, ist
$C_q := \{d \mid d \in R_q \wedge d = (\underline{id}, URI, description, 'class')\}$

Die Retrievalfunktion, die jeweils nur Dokumente vom Typ "Property"

zurückgibt, ist
$P_q := \{d \mid d \in P_q \wedge d = (\underline{id}, URI, description, 'property')\}$.

Diese beiden Retrievalfunktionen dienen dazu, nur Properties oder nur Classes zu einem Query zu finden.

Der Query, der nur aus einem konkreten Keyword besteht, ist q_w .

Alle Retrievalfunktionen geben außer den gefilterten Dokumenten noch eine numerische Bewertung, im Folgenden als *Score* bezeichnet, zurück. Je höher die Score, desto eher beantwortet ein Dokument $d \in D_q$ den Query q oder anders formuliert desto höher ist die Relevanz des Dokuments.

Retrieval von Dokumenten, die ein bestimmtes Keyword enhalten

Zu jedem Keyword w werden die beiden Retrievalfunktion C_q und P_q ausgeführt.
Als Rückgabe erhält man alle $d \in D$, die das Keyword beinhalten.

Aus den einzelnen Dokumentmengen, die man mit Hilfe der Retrievalfunktionen C_q und P_q erhalten hat, sollen SPARQL-Queries erstellt werden, die möglichst viele Keywords abdecken.

Die Anzahl der Keywords zu einer Eingabe i ist $n_w = count(T(q))$.
Die Funktion *count* liefert die Anzahl an Results.

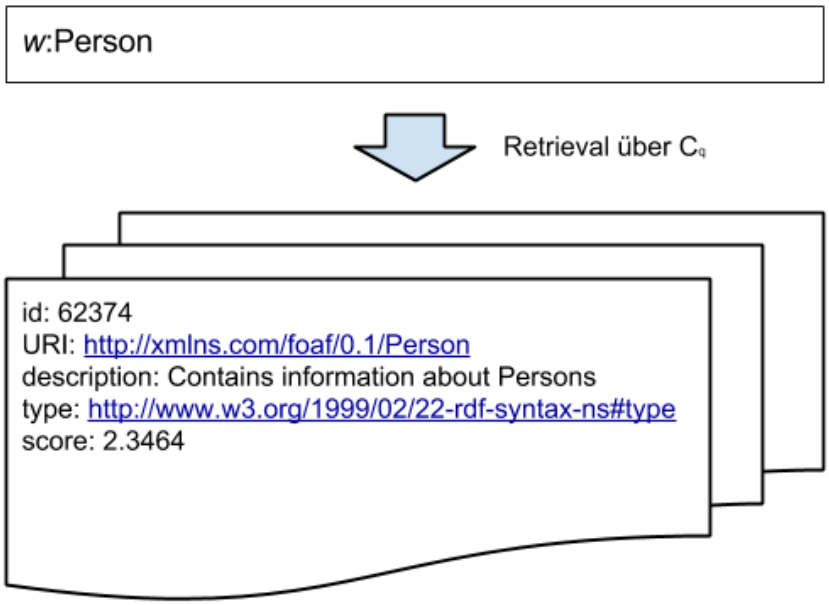


Abbildung 6: Retrievalfunktion Cq

2.4 Erstellen der SPARQL-Abfragen

Aus der Menge $D_q = C_q \wedge P_q$ für jedes q_w werden alle möglichen SPARQL-Abfragen erstellt.
Anders als bei QUICK [11] gibt es keine vordefinierten Query Templates, die zunächst verwendet werden. Die Query Templates suchen zu gegebenen Schema-Elementen vordefinierte Queries, von denen vermutet wird, dass sie zur Anfrage passen könnten. Die Templates würden die Bewertung verkomplizieren, da ihre Score sich variabel zu der Keyword-Anfrage anpassen müsste und sie könnten eventuell Keywords enthalten, die in der eigentliche Anfrage nicht enthalten sind. Der Einsatz von Query Templates ist jedoch hochinteressant und könnte in weiteren Arbeiten untersucht werden. Der gezielte Einsatz von Query Templates könnte auch einige der Performance-Probleme lösen, auf die im weiteren Verlauf gestoßen wird.

Eine SPARQL-Abfrage hat immer die selbe Struktur:

```
SELECT subject WHERE { subject predicate object . [ subject2 predicate2 object2 . ]* }
```

Die Kombination "subject predicate object ." wird als Triple-Pattern bezeichnet. Eine SPARQL-Abfrage kann unendlich viele Triple-Pattern besitzen. Jedes Triple-Pattern stellt genau eine Kante in einem Graphen dar. In SPARQL allgemein kann an jeder Stelle eine Variable oder eine URI stehen. In dieser Arbeit werden die Möglichkeiten jedoch eingeschränkt:

- Die Subjektposition soll immer von einer Variable besetzt sein. Das soll der Fall sein, damit die einzelnen SPARQL-Pattern über das mögliche Objekt des vorigen Patterns miteinander verbunden sind. Gibt es kein Objekt mit einer Variablenbelegung, ist die Belegung der Subjektposition mit einer Variablen immer noch sinnvoll. In diesem Fall ist das Subjekt immer mit "?x" belegt und beschreibt die zu suchende Variable.
- Die Prädikatposition soll nur von einer Property besetzt sein. Die Property ist entweder rdf:type oder die URI eines Schema-Elements vom Typ rdf:property. Damit dient die Prädikatposition entweder der Typisierung oder dem Verbinden von zwei Classes "?x" und "?y".
- Das Objekt soll nur von einer Class belegt werden. Das bedeutet, dass für jede Property die Objektposition immer eine Variable ist und für jede Class die URI der Class. Dadurch wird bewirkt, dass Properties im Schema-Graphen als Assoziationen zwischen zwei Classes dienen und Classes als Knoten vom Typ rdf:type.

Aus diesen Regeln ergibt sich für Properties das Pattern
$\forall c \in C : Pattern = ?x\ Predicate\ c$

und für Classes das Pattern
$\forall p \in P : Pattern = ?x\ p\ Object\ ,\ p = rdf:type$

Für jede Property wird daher das Triple-Pattern *?x Predicate ?y* eingesetzt und für jedes Objekt das Triple-Pattern *?x rdf:type Object.*

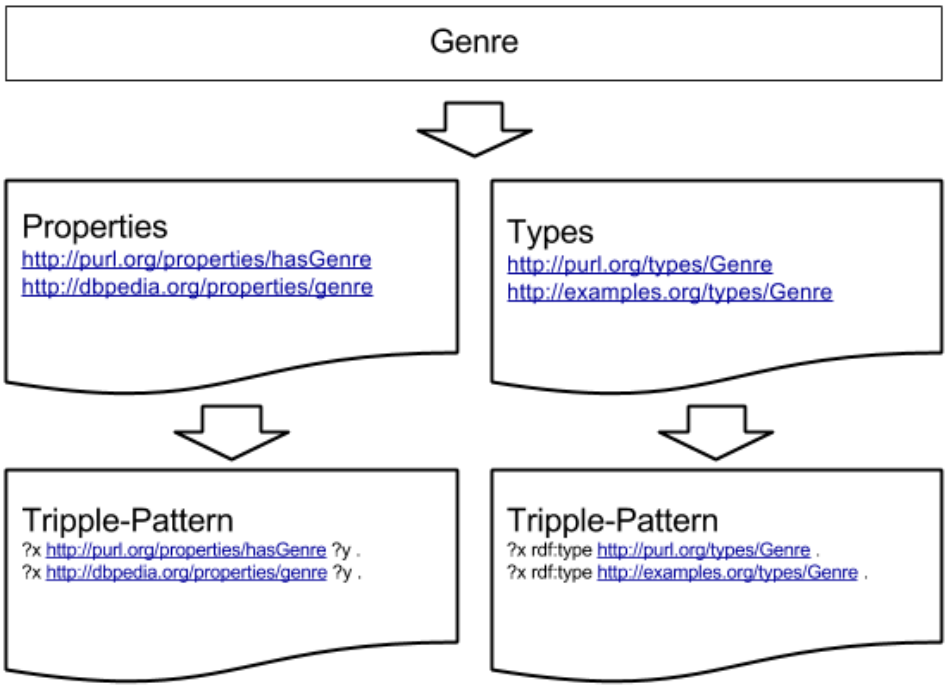


Abbildung 7: Erstellen von Triple-Pattern mit Beispiel "Genre"

Für jedes Keyword werden alls Triple-Pattern erstellt und können abgefragt werden.
Alle Triple-Pattern eines Keywords werden mit den anderen Triple-Pattern der anderen Keywords kombiniert.

Es wird also das kartesische Produkt über alle V_W , also die Vokabelmenge aller Keywords, gebildet.

$$S = C_q(w_1) \vee P_q(w_1)\, x\, C_q(w_2) \vee P_q(w_2)\, x \ldots x\, C_q(w_n) \vee P_q(w_n)$$

Dabei entsteht eine Menge von Schema-Teilgraphen. Die Schema-Teilgraphen sollen verschiedene Kantenlängen besitzen, um auch Abfragen über mehrere Kanten zu gewährleisten. Je nach Anzahl der Kanten eines Schema-Teilgraphen muss verschieden vorgegangen werden.

Suche auf Schema-Graphen mit Kantenlänge = 1

Ein Graph mit einer Kantenlänge von 1 ist ein Graph, in dem jeder Knoten den nächsten über genau eine Kante erreichen kann.

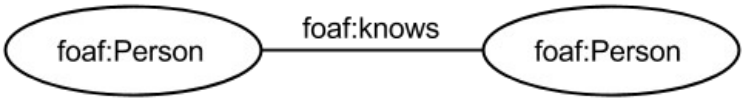


Abbildung 8: Verkürzte Darstellung eines Schema-Graphen mit Kantenlänge = 1

Für die Suche auf Schema-Graphen mit maximal einer Kantenlänge ergibt sich für eine SPARQL-Abfrage:

$$count(TP) = count(W) = count(N),$$
$$TP = Tripple - Pattern,$$
$$N = Kanten,$$
$$count = Zählt\ Elemente\ einer\ Menge$$

Es werden alle alle möglichen Kombinationen an Belegungen der Trippelpattern geliefert. Die Ergebnismenge S ist folglich die Menge aller erzeugten SPARQL-Abfragen über alle Kombinationen der Vokabeln aller Keywords. [4]

Für die Eingabe eines einzigen Keywords werden genau so viele S erzeugt wie es $count(D)$ gibt.
Für die Eingabe von zwei Keywords werden genau so viele S erzeugt wie es $count(D_{w1}) * count(D_{w2})$ gibt. w_1 Und w_2 stehen für die beiden Keywords.
Für die Suche auf Graphen mit der Kantenlänge 1 ist die Anzahl von Elementen in S also
$count(S) = count(D_{w0}) * count(D_{w1}) * (...) * count(D_{wn})$.

4 Siehe Anhang A1

Suche auf Schema-Graphen mit Kantenlänge > 1

Ein Graph mit einer Kantenlänge von mehr als 1 ist ein Graph, in dem sich die Knoten über beliebig viele Kanten erreichen können.

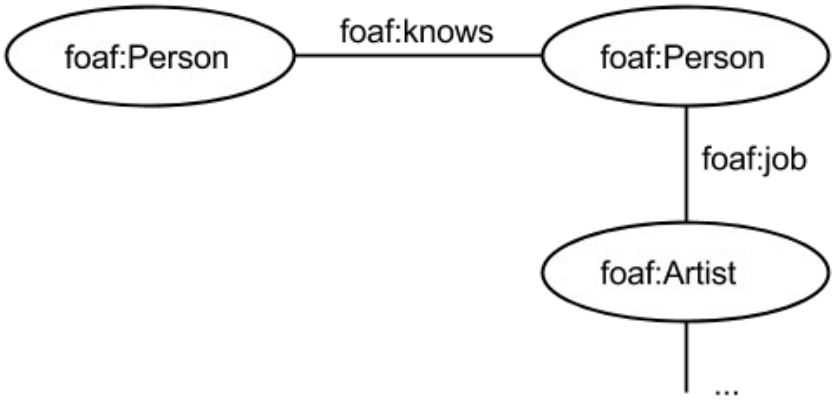


Abbildung 9: Verkürzte Darstellung eines Schema-Graphen mit Kantenlänge > 1

Die Suche über mehrere Kanten ist deutlich schwieriger. Für die Suche auf mehreren Kanten gilt:

$TP = s\ p\ o\ .\ s_1\ p_1\ o_1\ .\ (...)\ .\ s_n\ p_n\ o_n$, wobei

$s_0\ predicate\ o_0\ .\ s_1\ p_1\ object | s_1 = o_0$

Das bedeutet, dass jedes Subjekt eine neue Variable erhält und dass, falls das vorangegangene Triple-Pattern eine Property enthält, die neue Subjektvariable gleich der Objektvariablen des Property Triple-Patterns ist.

Für die Eingabe eines einzigen Keywords werden wieder genau so viele S erzeugt wie es $count(D)$ gibt.
Für die Eingabe von Keywords gilt allgemein:

$count(S) = count(D_{k0}) * 2 * count(D_{k1}) * 3 * (...) * n * count(D_{kn})$

Abgesehen davon, dass die Menge an SPARQL-Queries stärker wächst als bei Graphen mit genau einer Kantenlänge, wird die Suche auf den erstellten Teilbäumen mit jeder weiteren Kante komplexer. Vermutlich sind die durchschnittlichen Anfragen, die ein Nutzer stellt, einfache Anfragen, die sich nicht über mehr als zwei Kanten erstrecken. Also sind Queries, die darüber hinaus gehen, vermutlich nicht das, was der Nutzer wissen will.

Aufgrund dieser Vermutung werden auch nur solche Queries generiert. Das grenzt zudem die Menge an möglichen Queries ein.

Jeder SPARQL-Query erhält eine eigene Bewertung. Die Bewertung errechnet sich aus den Summen der jeweiligen Scores für einzelne $d \in D_k$.

$$Score_s = \sum_{i=0}^{n} score(d_k)$$

$score(d_k)$ berechnet der Index.

Zusammengesetzte Wörter - Problemlösung

Sollte ein Triple-Pattern schon einmal verwendet worden sein, wird es nicht wieder verwendet, sondern nur dessen Bewertung vervierfacht. Dieser Wert hat sich in einer vorläufigen Evaluation als günstig erwiesen. So wird das Problem, dass mehrere Wörter ein und dasselbe Wort bilden, umgangen, denn der kürzere SPARQL-Query, der ein Triple-Pattern enthält, das zwei oder mehr Keywords abbildet, wird höher bewertet.

Das Beispiel in Abbildung 15 zeigt, wie für eine Eingabe *i* "located near city" erkannt wird, dass "located near" sprachlich zusammen gehören. Da es keine URI *foaf:locatedNear* gibt, wird das zusammengesetzte Wort zunächst nicht erkannt. Die Eingabe *i* wird unbearbeitet übernommen. Es entstehen zwei verschiedene SPARQL-Statements. Für das Keyword "located" werden zwei Properties gefunden: *foaf:locatedNear* und *foaf:basedNear*. Für das Keyword "near" wird eine Property gefunden: *foaf:basedNear*. Da es nun für beide Wörter eine identische URI gibt, liegt der Schluss nahe, dass die beiden getrennten Keywords ein gemeinsames logisches Keyword bilden. Realisiert wird dies, indem die Scores der doppelten SPARQL-Patterns addiert werden und deren Summe anschließend verdoppelt wird (=vervierfacht). Anschließend wird das Duplikat im SPARQL-Statement entfernt. Dadurch erhält das nun aus zwei SPARQL-Pattern bestehende SPARQL-Statement eine höhere Score als das aus drei SPARQL-Pattern bestehende.

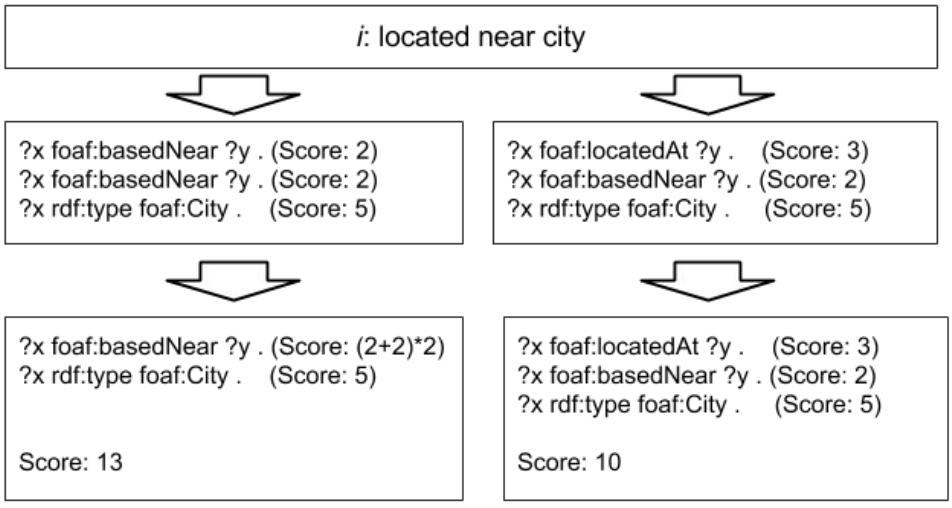


Abbildung 10: Zusammengesetzte Wörter: Erkennen durch Score

2.5 Bewertung

Alle $s \in S$ werden auf den Schemex angewandt und das Ergebnis bewertet.

Bewertungsgrößen sind:

(i) datasource(s): Anzahl der Datenquellen, die zu s relevant sind
(ii) instancecount(s): Anzahl Instanzen
(iii) score(s): Sei d(s): Menge der in s verwendeten Dokumente (Classes & Properties), dann gilt: $score(s) := \sum (S(q,d)), d \in d(s)$

Die Score ist als Bewertungsgröße ausgewählt worden, weil sie aufzeigt, wie sehr die URIs der verwendeten Schema-Elemente und deren textuelle Beschreibung zu den Keywords passen. Das ist ein Indikator dafür, ob ein Query relevant ist oder nicht.
Die Anzahl an Datasources ist als Bewertungsgröße ausgewählt worden, weil sie eine Aussage darüber trifft, ob es zu einem Query Ergebnisse gibt und in wie vielen verschiedenen Quellen das der Fall ist. Dieser Parameter ist ein Indikator dafür, ob ein Query überhaupt relevant sein kann. Eine Frage, die keine Antworten liefert, ist ein Indikator dafür, dass die Frage nicht gut formuliert wurde, also der Query eventuell nicht zielführend ist.
Eine ähnliche Begründung gibt es für die Instances. Dass zwischen Datasources und Instances differenziert wird, liegt daran, dass so noch einmal feiner bestimmt werden kann, ob sich die Bewertung nach der Anzahl an Datasources mit der nach der Anzahl an Instanzen deckt. Weichen die beiden Werte für verschiedene Queries stark voneinander ab, kann die Bewertung für einen Query mit weniger Datasources, aber mehr Instances eventuell noch dazu führen, dass dieser Query insgesamt relevanter wird.

Die Bewertungsgröße "Pfadlänge" ist nicht verwendet worden. Da die SPARQL-Queries sowieso nur Schema-Graphen mit maximal zwei Kanten abbilden, ist dieser Wert nicht relevant.

Gesamtbewertung

Bei der Gesamtbewertung soll derjenige SPARQL-Query ermittelt werden, der vermutlich richtig ist. Die Vermutung ist, dass dabei alle drei Größen eine Rolle spielen, also Bewertung, Instanzen und Datenquellen. Ob diese Signale helfen können, Relevanz oder Bedeutung von Queries zu erfassen, wird später in der Nutzerevaluation getestet.

Zunächst wird ihre Gewichtung festgelegt.
Damit die drei Größen score *s,* datasources *d*, instances *i* zueinander in Relation gesetzt werden können, benötigen sie die selbe Einheit und müssen einen Wert x vom Gesamtwert bilden. Die Einheit jeder Bewertungsgröße

wird in Prozent vom höchsten Wert angegeben. Es gilt:

$$relevance_x = value / max(value)$$

Für die Gesamtbewertung gilt zunächst:

$$R_{gesamt} = x * R_s + y * R_d + z * R_i$$

x, y und z sind die jeweiligen Relevanzkoeffizienten für die Größen R_s, R_d und R_i. R_s ist die Relevanz der Score, R_d die Relevanz der Menge an Datasources und R_i die Relevanz der Menge an Instances.

Ziel ist es, ein Verhältnis zwischen R_s, R_d und R_i zu ermitteln, um so zu einer Gesamtbewertung zu kommen, die für die meisten Queries stimmt. R_s, R_d und R_i sind variable Größen. x, y und z sind statische Werte der Zielfunktion R_{gesamt}, die es zu ermitteln gilt. Die Zielfunktion ist dann vollständig, wenn x, y und z einen festen Wert erhalten haben.

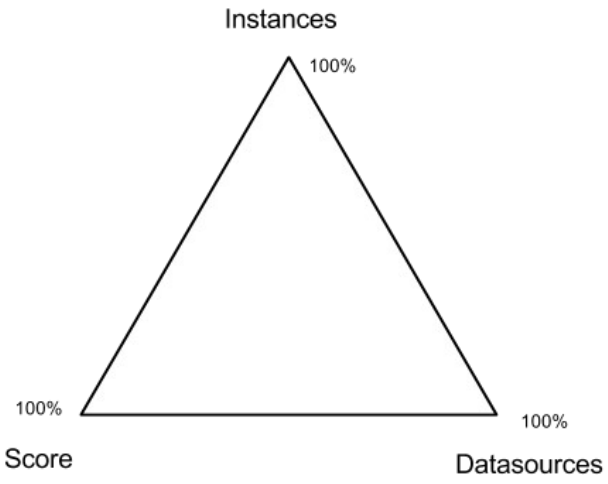


Abbildung 11: Ermittlung der Relevanzkoeffizienten im Dreiecksdiagramm

Die Relevanzkoeffizienten werden statistisch ermittelt. Das Verfahren beruht auf keiner Quelle, sondern ist für diesen Fall selbst erdacht. 10 Beispielanfragen werden hinsichtlich ihrer Relevanz untersucht, wenn man die Werte x, y und z der Funktion R_{gesamt} ändert.
Die Werte x, y und z bilden ein Dreiecksdiagramm, das zeigt, in welchem Bereich das relevanteste Resultat auch das erwartete Resultat ist und in welchem nicht.
Das Diagramm wird gebildet, indem einer der drei Werte auf 0 gesetzt wird und die anderen beiden verändert werden. Dabei setzt man zunächst einen der beiden Werte auf 1 und den anderen auf 0. Anschließend vermindert man den Wert um ein bestimmtes Interval und fügt den Wert des Intervals

zum anderen Wert hinzu. Solange das Resultat dem Erwartungswert entspricht, werden die Werte weiter verändert. Sobald das Resultat nicht mehr dem Erwartungswert entspricht, das heißt der Übergangswert ist überschritten, wird der Wert notiert und auf dem entsprechenden Schenkel des Dreiecks eingetragen. Wird zum Beispiel der Wert für die Score und der Wert für die Datasources verändert, wird der Übergangswert auf der horizontalen Achse des Dreiecks vermerkt – also dem Schenkel, der die Punkte 100% Score und 100% Datasources miteinander verbindet. Es ergeben sich durch diese Methode drei Punkte, die dann miteinander verbunden werden. Der Bereich, der von dem daraus gebildeten Dreieck umschlossen wird, ist der Wertebereich für x, y und z, der den Erwartungswert liefert. Die Prozentwerte für Punkte innerhalb des Dreiecks werden ermittelt, indem der Abstand der Ecken zu dem Punkt gemessen und durch die Schenkellänge geteilt wird.

Für die Keyword-Abfrage "Author" z. B. können Punkte in einem Erwartungsbereich von maimal 40% Datasources, 40% Instances und minimal 60% Score gewählt werden. Die Abbildungen sind lediglich Skizzen, dienen der Visualisierung und enthalten daher keine detaillierteren Werte.

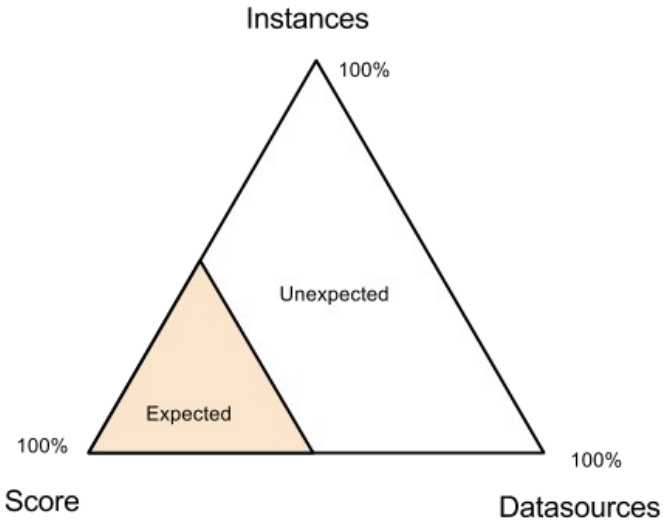


Abbildung 12: Resultat der Keywordeingabe "Author"

Für die Eingabe "Person" ist das Diagramm fast umgedreht.

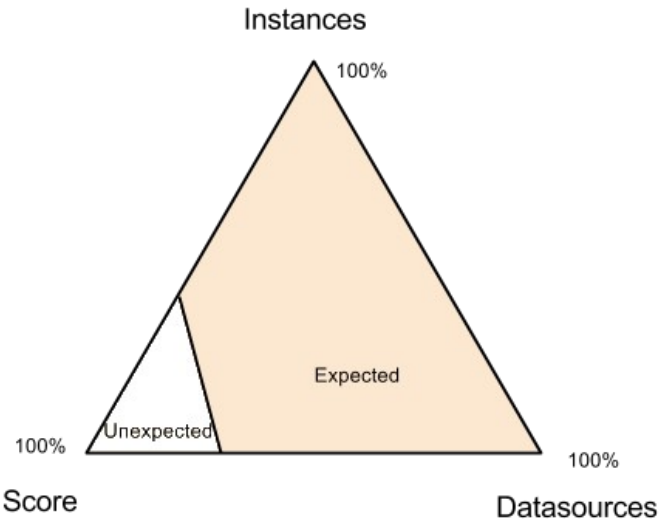


Abbildung 13: Resultat der Keywordeingabe "Person"

Aus allen zehn Dreiecksdiagrammen wird ein kummuliertes Diagramm gebildet, das die Werte *x*, *y* und *z* für R_{gesamt} bestimmen soll. Das geschieht über die Schnittmenge aller 10 Dreiecksdiagramme.

Abbildung 14: Kummuliertes Dreiecksdiagramm für alle zehn Testeingaben

Alle Positionen innerhalb dieses Bereichs ergeben für alle zehn Queries erwartete Resultate. Auf dieser Grundlage werden die Relevanzen wie folgt festgelegt:

x, der Anteil der Score an der Relevanz, wird mit 0,65 bestimmt.

y, der Anteil an Datasources an der Relevanz, wird mit 0,25 bestimmt.

z, der Anteil an Instanzen an der Relevanz, wird mit 0,1 bestimmt.

Dass y höher als z gewichtet wird, ist so zu interpretieren, dass die Anzahl an Datenquellen im Vergleich zu der Anzahl an Instanzen mehr Aussagekraft hat. Eine einzige Datenquelle könnte mehr Instanzen enthalten als die Summe von Instanzen vieler Datenquellen. Viele Datenquellen sprechen eher dafür, dass eine Abfrage relevant ist. Je mehr unabhängige Antworten es auf eine Abfrage gibt, desto eher macht die Abfrage als Solches Sinn, da verschiedene Quellen die Frage kennen, verstehen und beantworten können.

Es ergibt sich die Relevanzfunktion

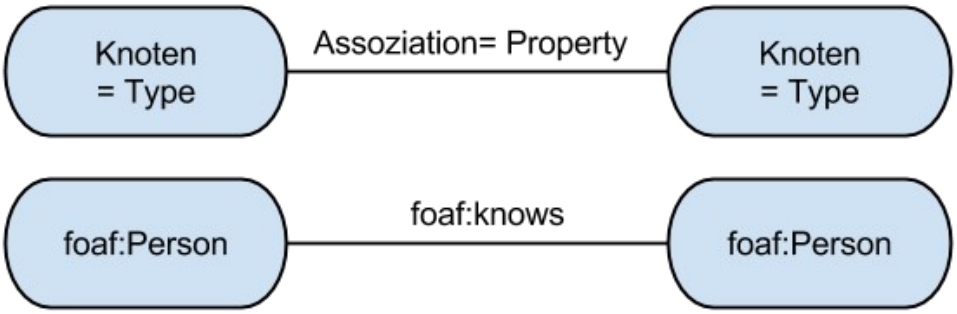
$$R_{gesamt} = 0{,}65 * R_s + 0{,}25 * R_d + 0{,}1 * R_i$$

3. Umsetzung

In diesem Kapitel wird die Implementation der Methodik beschrieben. Die Implementation erfolgt in PHP mit dem Framework Symfony2. Zunächst wird der Index aufgebaut und anschließend aus der tokenisierten Eingabe SPARQL-Abfragen erstellt, die an Lodatio gesendet und anschließend bewertet werden.
Die Codebeispiele werden zu Veranschaulichungszwecken abgeändert und auf das Wesentliche reduziert.

3.1 Aufbauen des Index

Der Index dient dazu, Schema-Daten vorzuhalten, sodass sie abgefragt werden können. Schema-Daten sind eine Menge von Schema-Elementen. Ein Schema-Element wird durch eine URI identifiziert. Die Ressource, die durch die URI identifiziert wird, enthält weitere Informationen, zum Beispiel einen Beschreibungstext und einen Typen. Der Typ definiert, welche Rolle eine URI innerhalb eines semantischen Graphen einnimmt. So steht der Typ "Property" zum Beispiel für eine Assoziation zwischen zwei Knoten. Der Typ "Class" widerum für einen Knoten. Der Index soll zu einem Keyword eine Liste von Schema-Elementen zurückgeben, die das Keyword entweder in der URI oder in dessen Beschreibungstext enthalten und aus denen ein Teilgraph aufgebaut werden kann. Somit sind für den Index nur solche URIs relevant, deren Typ entweder "Property" oder "Class" ist. Aus Properties werden im Graphen Assoziationen, aus Classes Knoten. Ziel ist es, dass aus den Rückgaben des Index ein Teilgraph aufgebaut werden kann, der jedes Keyword genau einmal entweder als Knoten oder Assoziation enthält. Der Teilgraph wird als SPARQL-Abfrage ausgedrückt.



Damit der Index dies leisten kann, muss jeder Eintrag eine URI, einen Beschreibungstext und einen Typ besitzen. Da viele URIs verschiedene Beschreibungstexte besitzen, enthält der Index mehrfach Einträge für URIs – jeder mit einem eigenen Beschreibungstext. Ein Index-Eintrag wird im Folgenden Dokument genannt und stellt eine URI-Beschreibung dar.

Folgende Struktur weisen daher alle Dokumente auf:

Name	SOLR-Name	SOLR-Typ	Beschreibung
Id	id	Int	Eine eindeutige ID des Eintrags
URI	uri_t	Text	Die URI des Dokuments
Type	type_t	Text	Der Typ des Dokuments. Entweder rdfs:Class oder rdfs:Property
Description	description_t	Text	Eine Beschreibung des Inhalts

Diese Felder werden im Folgenden Standard-Felder genannt.

3.2 Datenbasis

Zunächst werden die Daten, auf denen der Schemex basiert, heruntergeladen.[5]
Die Daten, auf denen SchemEX basiert, liegen im NQuad-Format vor. SchemeEX ist als RDF beschrieben. Die Daten gibt es zum Download im N3-Format [6]. Das N3 – Format besteht aus n Trippeln. Ein Triple besteht aus drei Zeichenketten, die durch Leerzeichen getrennt sind und enden mit einem Punkt. Zusammen bilden sie einen Satz – Subjekt, Prädikat, Objekt, Punkt. Das N3-Format ist ein Format zur Serialisierung von RDF.
Jedes Triple aus dem Schemex-Datensatz wird nach Properties und Types durchsucht und deren URIs gespeichert.
Die Ressourcen, die durch die URIs identifiziert werden, werden anschließend heruntergeladen und im Dateisystem abgelegt. Die meisten dieser Ressourcen sind Dateien im RDF-Format. Sie enthalten detaillierte Informationen wie ihren Typen und Beschreibungstexte.
Sie werden von ihrem Ausgangsformat in das JSON-LD-Format konvertert[7]. Das Konvertieren erfolgt mit Hilfe der API der URL http://rdf-translator.appspot.com. Das SOLR-Plugin für Symfony kann nur solche Dokumente indizieren, die im JSON-LD-Format vorliegen.
Nach dem Übersetzen der Ressourcen werden diese dem Index hinzugefügt. Das Hinzufügen erfolgt über ein Konsolenkommando, das für diesen Zweck implementiert wurde. Das Kommando liest jede Ressource nacheinander ein und fügt sie dem Index hinzu.

5 http://km.aifb.kit.edu/projects/btc-2010/000-CONTENTS
6 http://www.w3.org/TeamSubmission/n3/
7 http://json-ld.org/

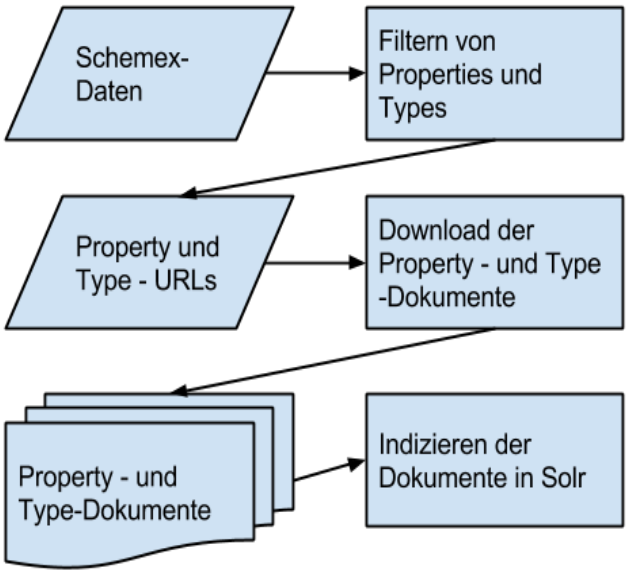


Abbildung 15: Vorgehensweise Indizieren der Schemex-Schema-Daten

Umsetzung

Das Dokument $d := (\underline{id}, URI, description, type)$ ist eine Klasse, deren private Attribute die Standard-Felder sind. Die Standard-Felder sind über Setter manipulierbar und über Getter auslesbar.

Alle Attribute erhalten die Annotation `@Solr\Field(type="{type}")`. Sie bewirkt, dass bei lesendem Zugriff auf die Klasse der Inhalt von Solr auf das entsprechende Attribut übertragen wird. Heißt eine Variable zum Beispiel "description", wird bei lesendem Zugriff auf die Klasse die Variable "description" mit dem Wert des Feldes "description" aus Solr für die entsprechende ID befüllt. Im schreibenden Fall wird das Feld "description" mit dem Inhalt der Variable "description" ersetzt. Wichtig ist dabei, dass der angegebene Datentyp mit dem Solr-Datentypen übereinstimmt. Somit ermöglicht die Annotation eine Objekt-Abbildung des Solr-Resultsets.
Objektabbildungen werden in objektorientierten Sprachen gerne in Verbindung mit relationalen Datenbanken gebraucht, da ihr Resultset ungetypt ist, aber für die weitere Verarbeitung ein Typ erforderlich ist. In diesem Zusammenhang spricht man von einer objektrelationalen Abbildung[8].

In der Implementation gibt es anders wie im theoretischen Teil nur den Typen "Dokument" D. Die Unterscheidung in D_P und D_C wird nicht durch Typisierung vorgenommen.

8 http://de.wikipedia.org/wiki/Objektrelationale_Abbildung

3.3 Tokenisierung

Die Tokenisierungsfunktion T [9] zerteilt den Eingabestring i zunächst an den Leerzeichen. Das geschieht mit der explode-Funktion von PHP. Sie zerteilt einen String an dem angegebenen Trennzeichen und gibt ein Array zurück.

Für das Verbinden von sprachlich zusammenhängenden Wörtern wird die Funktion so erweitert, dass sie jeweils benachbarte Keywords konkadiniert und den Index nach Treffern abfragt. Bei dem Abfragen nach Treffern werden lediglich URIs gesucht, die exakt so heißen wie das Wort. Wenn Beschreibungstexte auch berücksichtigt werden, werden Wörter verbunden, die sprachlich nicht zusammen hängen. Das Zusammenfassen an dieser Stelle funktioniert nur, wenn die zugrunde liegenden Dokument-URIs sprechend sind. Da das auf viele Dokument-URIs zutrifft, funktioniert der Ansatz.
Ist die Anzahl an Treffern für zwei konkadinierte Wörter größer als 0, werden die beiden einzelnen Wörter des Keyword-Strings durch ein einziges Wort ersetzt. So wird aus "state of" "stateof".
Wörter, die aus mehr als zwei einzelnen Wörtern bestehen, bleiben unberücksichtigt. Die Funktion ließe sich noch so erweitern, dass sie solche Wörter erkennt, die aus beliebig vielen Wörtern bestehen.

Die Funktion "retrieve", die im Anhang verwendet wurde, ist die Retrievalfunktion R_q . Sie wird im nächsten Kapitel erläutert.

3.4 Retrieval

Die Retrievalfunktionen C_q und P_q sollen zu einem Query jeweils die Dokumente zurückgeben, die nur Classes oder nur Properties enthalten.
Sie benötigt zwei Parameter: Den Dokumenttyp und den eigentlichen Query. Das Solr-Plugin wird mit dem Typen, der Anzahl an Ergebnissen und dem eigentlichen Query initialisiert und ausgeführt. Intern sendet das Solr-Plugin zunächst den Query über die Solr's API an Solr und bildet das Resultset mit Hilfe der Metadaten der Klasse auf die entsprechende Klasse und deren Attribute ab. Dabei werden die Datentypen berücksichtigt. Über die Funktion *getResult* erhält man somit ein vollständig befülltes Objekt vom Typ *$type*. Auch hier wird die Objekt-Abbildung angewandt.

In Verbindung mit Solr ist es eine Abbildung des Resultsets auf den entsprechenden Datentyp *Property* oder *Type*. Das Object Mapping ist an dieser Stelle wichtig, um eine Typisierung zu gewährleisten. Ohne das Mapping wäre das Resultset eine Hashmap und es wäre nicht klar, welche Werte wie herausgelesen werden können.
Es werden aufgrund von Performance-Gründen nur die wichtigsten 5 Ergebnisse zurückgegeben. Je mehr Daten im Resultset enthalten sind, desto

9 Siehe Anhang A2

mehr mögliche Queries werden erzeugt und desto schlechter wird die Performance.
Es wird nach Duplikaten gesucht und diese aus dem Ergebnis entfernt. Duplikate treten deshalb auf, weil die Dokumente im Index lediglich URI-Beschreibungen und keine URIs als Solches darstellen.

Retrieval von Dokumenten, die ein bestimmtes Keyword enthalten

Der konkrete Query zu einem Keyword unter Einbeziehung des jeweiligen Dokumenttyps ist:

```
"(url_t:$keyword OR (value_t:$keyword)^0.4) " . ' AND ' . $type
```

URIs, die das Keyword enthalten, werden mit 1 gewichtet, Beschreibungstexte, die das Keyword enthalten, werden mit 0.4 gewichtet. Das soll verhindern dass weniger relevante URIs zu hoch bewertet werden. Die Zahl 0.4 ist ein geschätzter Wert, der sich durch Beobachtungen bestätigt hat. Der Wert kann mit Sicherheit noch verfeinert werden. *$type* enthält Queries zum Typen.

3.5 Erstellen der SPARQL-Abfragen

Für das Erstellen von SPARQL-Pattern dient die Funktion createSparqlPattern[10]. Sie entscheidet anhand der Belegung von Subjekt, Prädikat und Objekt wie das SPARQL-Pattern aufgebaut wird.
Ist kein Prädikat gegeben, wird an der Prädikatposition der Typ des Objekts gesetzt, ansonsten wird das Prädikat mit der URI des Prädikats belegt.
Ist kein Objekt gesetzt, wird die Objektposition mit der Variablen ?y belegt, ansonsten mit der URI des Objekts.
Das SPARQL-Pattern wird anschließend in der Form "Subjekt Predikat Objekt ." zurückgegeben.

Subject	**Predicate**	**Object**	**Result**
?x	foaf:knows	null	?x foaf:knows ?y .
?x	null	foaf:Person	?x rdf:type foaf:Person .

Die Tabelle veranschaulicht die Funktionsweise der Funktion *createSparqlPattern* bei verschiedener Belegung der Variablen.

Da zum Erstellen von SPARQL-Pattern Properties und Types benötigt werden, und diese in der Keyword-Klasse zu finden sind, erhält die Keyword-Klasse die Funktion getQueries[11]. Sie generiert alle möglichen SPARQL-Pattern für alle Properties und Types des Keywords. Sie iteriert über alle Properties und Types und erstellt ein gültiges SPARQL-Pattern. Zu

10 Siehe Anhang A4
11 Siehe Anhang A5

den Pattern wird eine Bewertung hinzugefügt. Diese Bewertung setzt sich aus den Bewertungen des Index zusammen. Enthält ein Pattern nur genau ein Dokument, ist die Bewertung die selbe Bewertung wie die des Index. Enthält der Query sowohl eine Property als auch einen Type, werden beide Bewertungen addiert und auf ihr Zehntel reduziert. Das liegt daran, dass solche Queries in der Regel unbrauchbar sind, da sie sowohl auf Prädikatposition als auch auf Objektposition auf ein Dokument verweisen, das sie enthält. Das würde bedeuten, dass das Wort sowohl Verb als auch Nomen wäre. Solche Fälle sind sehr selten und werden daher gering bewertet. Allerdings sollten sie nicht einfach ausgeschlossen werden. Wenn es zu solchen Queries dennoch gute Ergebnisse seitens Lodatio gibt, kann sich ihre Gesamtbewertung auch durchaus noch so sehr steigern, dass sie höher gewichtet werden als solche mit einer hohen Grund-Score. Das wird im Kapitel Bewertung noch weiter ausgeführt.

Alle SPARQL-Pattern des Keywords müssen nun mit den SPARQL-Pattern der anderen Keywords über das kartesische Produkt konkadiniert werden. Dadurch wird aus einzelnen SPARQL-Pattern eine Aneinanderreihung von SPARQL-Pattern, die im Folgenden SPARQL-Statements genannt werden. Erreicht wird dies durch Konkadieren. Die SPARQL-Statements bilden die WHERE-Clause eines fertigen SPARQL-Queries. Sie erhalten eine Bewertung[12]. Die Bewertung ist die Summe der Bewertungen der einzelnen SPARQL-Pattern eines SPARQL-Statements. Es gibt dabei nur eine einzige Ausnahme, die dabei hilft, das Problem von zusammengesetzten Wörtern anzugehen.

Zusammengesetzte Wörter - Problemlösung
Entspricht ein SPARQL-Pattern exakt dem SPARQL-Pattern eines anderen Keywords, wird davon ausgegangen, dass es sich bei der Eingabe um ein kombiniertes Keyword handelt und dementsprechend wird die Score vervierfacht, damit der Query mehr Gewicht erhält.

Außer, dass alle Queries miteinander verbunden werden, werden aus der vorherigen Variable ?x die Variable ?y, aus der Variable ?y die Variable ?z . Das ermöglicht das Suchen über mehrere Kanten.

Anfragen an Lodatio
Lodatio wird mit den zehn relevantesten SPARQL-Queries angefragt[13]. Die Lodatio-API erwartet als GET-Parameter ein Datensatz-Limit und einen Query und gibt das Ergebnis in JSON zurück. Das Ergebnis wird deserialisiert und das Feld "Result" zurückgegeben. Sollte es zu einem Fehler gekommen sein, wird der Fehler abgefangen und ein leeres Ergebnis zurückgegeben. Das leere Ergebnis wird sowieso niedrig bewertet und entspricht daher einem Fehlerfall.

12 Siehe Anhang A6
13 Siehe Anhang A7

3.6 Bewertung

Die Bewertung nach der Score wird dadurch realisiert, dass die bereits vorhandene Score als Sortierargument verwendet wird und alle SPARQL-Queries absteigend nach ihrer Score sortiert werden.
Die Bewertung nach Datasources und Instances wird dadurch realisiert, dass die Anzahl aller Datasources bzw. Instances ermittelt wird[14]. Anschließend wird die Liste aller SPARQL-Queries sortiert absteigend nach Anzahl der Datasources bzw. absteigend nach der Anzahl der Instanzen.

Für die Gesamtbewertung werden alle drei Bewertungsquotienten addiert. Nur solche Quotienten, die Datasources und Instances enthalten, werden in das Resultset aufgenommen. Eine Abfrage, die keine Ergebnisse erzielt hat, wird kategorisch als relevant ausgeschlossen. Eine Abfrage ohne Antworten kann keine relevante Abfrage sein.

14 Siehe Anhang A8

4. Evaluation

In diesem Kapitel wird die Implementation evaluiert. An der Evaluation nehmen drei Personen teil. Alle Teilnehmer verstehen die Abfragesprache SPARQL.
Es wird zunächst untersucht, ob es Duplikate der Keyword-Abfragen gibt. Solche Duplikate sollen zusammengefasst werden.
Anschließend wird untersucht, wie hoch die maximale Bewertung einer Eingabe ist und wie sich die maximalen Bewertungen verteilen. Die maximale Bewertung ist die höchste Punktzahl eines Queries innerhalb einer Query-Liste einer Keyword-Abfrage.
Danach wird analysiert, inwiefern sich die Punktzahlen auf die Problemkategorien aufteilen und ob sich eine Aussage darüber treffen lässt, inwiefern Probleme besser oder schlechter gelöst werden können. Besondere Problemstellungen wie zusammenhängende Wörter oder Stop Words sollen im Einzelnen noch einmal analysiert werden. Es soll aufgezeigt werden, ob die angewandte Methode dazu führte, diese Probleme einzugrenzen oder nicht.
Im Folgenden werden die einzelnen Phasen der Bewertungsfunktion ausgewertet. Es gilt herauszufinden, ob die Bewertungsfunktion alle Parameter, also Score, Datasources und Instances, benötigt und ob sie diese im richtigen Maßstab miteinander zu einer Gesamtbewertung zusammenfasst.
Das Nutzerverhalten soll ebenfalls untersucht werden. Besonderheiten sollen aufgezeigt werden und es überprüft werden, welche Eingaben der Nutzer zu guten und zu weniger guten Ergebnissen geführt haben.
Zuletzt sollen noch solche Anfragen untersucht werden, die aus Sicht des Autors, also subjektiv, richtig sind und ausgewertet werden, wie viele davon ein positives Resultat erzeugen.

4.1 Vorgehensweise

Die Teilnehmer sollen aus 16 verschiedenen SPARQL-Queries englische, natürlichsprachliche Abfragen bilden. Die Abfragen werden dann in das Webinterface eingegeben. Die Antworten, also die erstellten SPARQL-Queries, werden unsortiert an eine unabhängige, dritte Person gegeben, dessen Aufgabe es ist, die einzelnen Queries zu bewerten.

Problemkategorien
Die 16 zu übersetzenden SPARQL-Abfragen sind in vier verschiedene Problemkategorien eingeteilt.
In der ersten Kategorie geht es um Abfragen auf genau ein Schema-Element. Dieses Schema-Element darf eine Class, aber auch eine Property sein. Ziel dieser Kategorie ist es, herauszufinden, wie gut zu einer Eingabe ein konkretes Element gefunden wird.
Die zweite Kategorie setzt zwei Schema-Elemente in Relation. Es soll

überprüft werden, ob bei der Eingabe der Keywords genau zwei Schema-Elemente gefunden werden und deren Beziehung richtig interpretiert wird.
In der dritten Kategorie werden Elemente verwendet, die entweder mehrfach oder in verschiedenen Kontexten verwendet werden können. Eine typische Anfrage, die ein Element mehrfach in ihrem SPARQL-Query benötigt, ist die Anfrage "Person knows Person". Dass ein Element in verschiedenen Kontexten verwendet werden kann, wird an dem Beispiel "Genre" klar. Genre kann sowohl als Class als auch als Property aufgefasst werden. Ziel ist es, herauszufinden, ob für eine gegebene Keyword-Abfrage das richtige Element verwendet wurde.
Die vierte Kategorie enthält Abfragen, deren Elemente im natürlichen Sprachgebrauch mit mehreren Wörter übersetzt werden, z. B. "handball league". Es gilt zu überprüfen, ob bei der Übersetzung nur genau ein Schema-Element herangezogen wird und "handball" und "league" nicht als einzelne Wörter aufgefasst werden.

Die Abfragen der Nutzerstudie sind im Anhang aufgelistet[15].

Auswertung der Nutzerstudie
Alle Keyword-Anfragen werden einzeln in das Webinterface eingegeben. Die Antworten, also eine Liste von SPARQL-Queries, sollen an eine Dritte, unabhängige Person, im Folgenden Prüfer genannt, gegeben werden. Er soll eine unabhängige Bewertung der Ergebnisse vornehmen. Um die Unbefangenheit des Prüfers zu garantieren, werden die einzelnen Ergebnislisten umsortiert und deren Score unkenntlich gemacht. Außerdem wird dem Prüfer nur mitgeteilt, um welchen ursprünglichen SPARQL-Query es geht. Er kennt also nicht die genaue Keyword-Eingabe. Der Prüfer weiß nur noch, welche SPARQL-Queries zu welchem ursprünglichen SPARQL-Query gehören.
Auf Grundlage dieser Informationen bewertet er jeden SPARQL-Query mit der Punktzahl 0,1 oder 3. Drei Punkte bedeutet, dass das Ergebnis als richtig angesehen wird, 0, dass es nicht im Entferntesten damit zu tun hat. Ein Punkt wird vergeben, wenn einige Elemente richtig erkannt wurden.

4.2 Auswertung

Alle 16 möglichen Keyword-Abfragen sind pro Teilnehmer vollkommen verschieden. Es gibt kein einziges Duplikat. Daher gibt es 48 verschiedene Keyword-Abfragen.

Verteilung nach maximaler Bewertung
Für 48 Keyword-Abfragen gibt es 21 Abfragen, die eine maximale Bewertung von 3 besitzen. Das sind 43,75 % aller Abfragen.
In weiteren 6 Fällen gibt es Abfragen, die mit maximal einem Punkt bewertet sind, also 12,5%.

15 Siehe Anhang A9

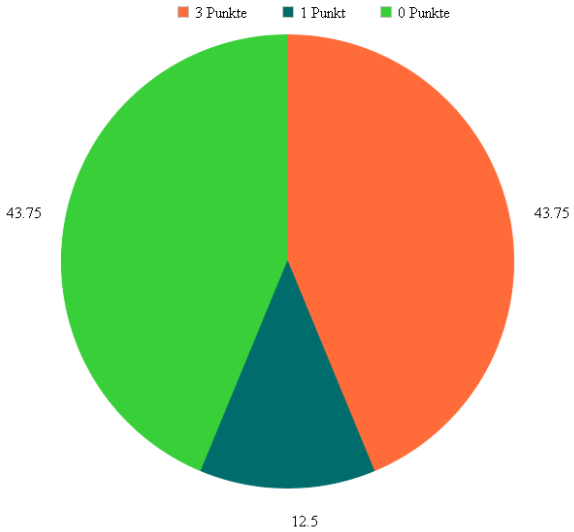


Abbildung 16: Verteilung Abfragen nach maximaler Bewertung

Die restlichen 21 Abfragen erzielen keine Punkte, was ebenfalls 43,75% entspricht.

Problemkategorien

Es gibt keine besonderen Auffälligkeiten für die Problemkategorien. Die positiven und negativen Ergebnisse sind auf alle 4 Problemkategorien gleichmäßig verteilt. Für Problemkategorie 1 gibt es 5 von 12 richtigen Ergebnissen, Problemkategorie 3 und 4 haben dieselben Werte. Problemkategorie 2 hat 6 von 12 richtigen Ergebnissen.
Daraus lässt sich ableiten, dass der Ansatz mit jeder Art von Problem die selbe Trefferquote erzielt.

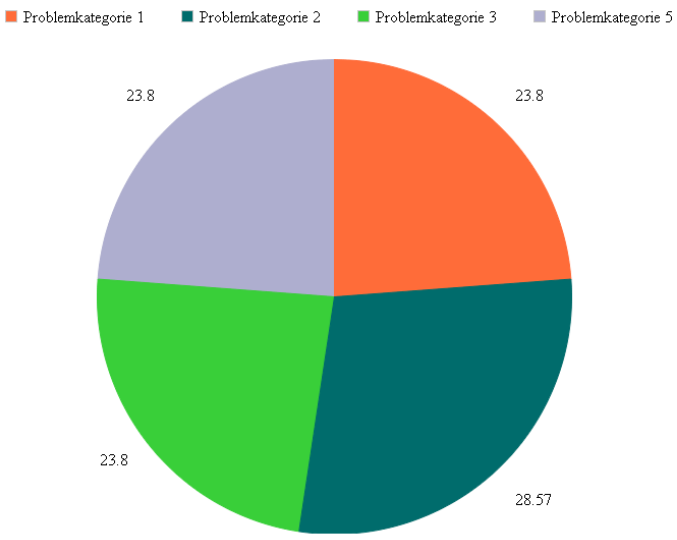


Abbildung 17: Verteilung Queries mit 3 Punkten auf Problemkategorien

In den Problemkategorien 3 und 4 geht es um zusammengesetzte Wörter. Zusammengesetzte Wörter werden mit der selben Häufigkeit erkannt wie einzelne Wörter. Daraus lässt sich schließen, dass die Problematik von zusammengesetzten Wörtern mit der erarbeiteten Methodik lösbar ist.

Gewichtung nach Score

16 von 21 Abfragen mit der Punktzahl 3 besitzen die höchste Score, was ca. 76% entspricht. Für 5 Abfragen, also ca. 24%, trifft dies nicht zu. Diese Abfragen befinden sich verteilt auf den Plätzen 2 bis 4. Das zeigt, dass die Score ein guter Indikator dafür ist, ob ein Query relevant ist oder nicht. Die weitere Untersuchung zeigt, dass dieser Wert allerdings Wert durch die Hinzunahme weiterer Bedingungen noch verbessert werden kann.

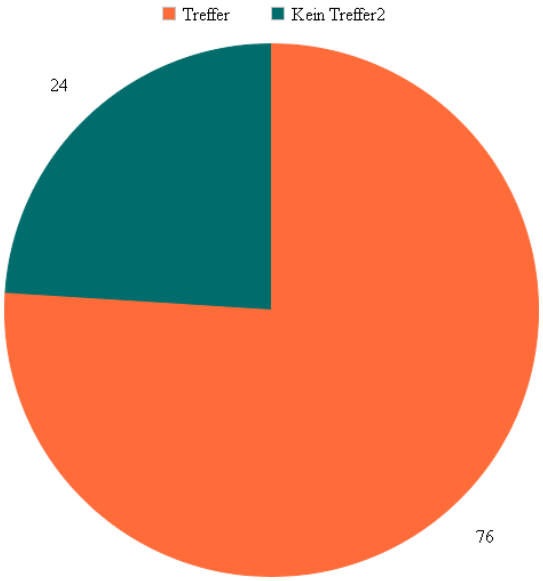


Abbildung 18: Gewichtung nach Score: Übereinstimmung Ranking

Gewichtung nach Datasources

10 von 21 Abfragen mit der Punktzahl 3, also ca. 48%, liefern für das richtige Ergebnis die höchste Menge an Datasources und wurden somit am höchsten gewichtet. Das zeigt, dass eine reine Gewichtung nach Datasources keine genaue Aussage über die Relevanz eines Ergebnisses trifft. Sie kann aber genutzt werden, um Relevanz-Bewertungen zu verfeinern.

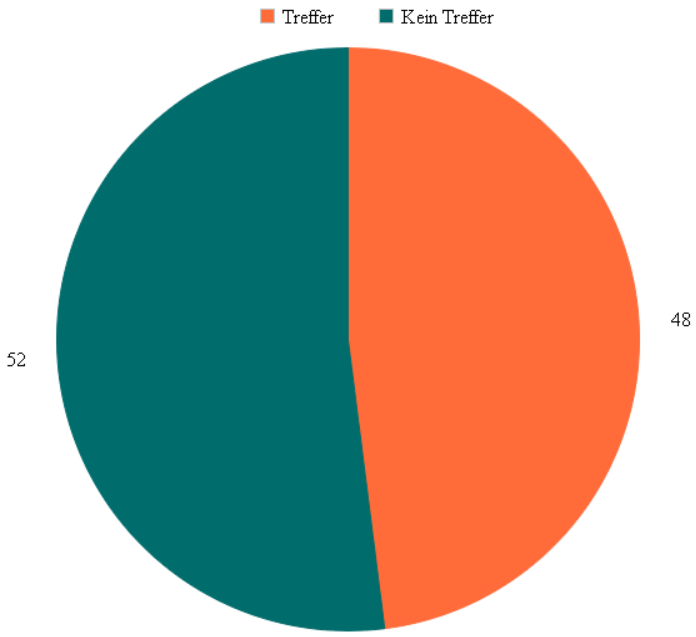


Abbildung 19: Gewichtung Datasources: Übereinstimmung Ranking

Gewichtung nach Instances

Nur 7 von 21 Abfragen mit der Punktzahl 3, also ein Drittel, liefern für das richtige Ergebnis die höchste Menge an Instances und wurden somit am höchsten gewichtet. Das zeigt, dass die reine Gewichtung nach Instances für eine Bewertung der Relevanz nicht ausreicht. Zur Verfeinerung von Bewertungen kann diese Größe aber dennoch gut genutzt werden.

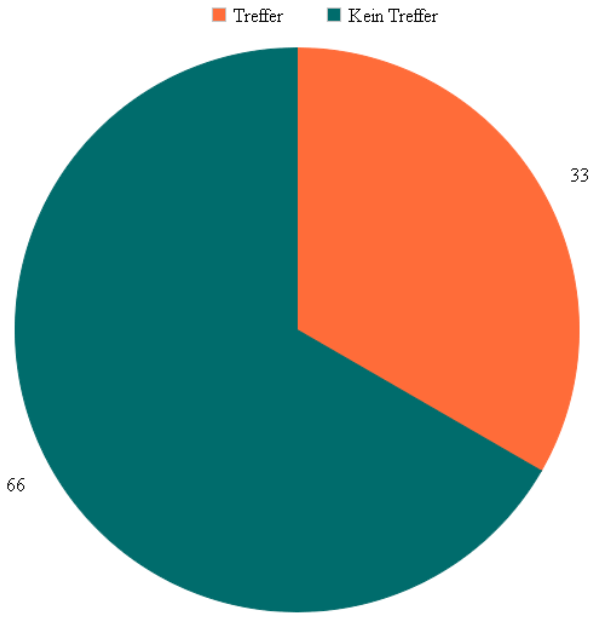


Abbildung 20: Gewichtung Instances: Übereinstimmung Ranking

Gesamtgewichtung
Alle 21 Abfragen, die mit drei Punkten bewertet wurden, sind auch die favorisierten Abfragen.
Die Abfragen mit einer Bewertung von 1 befinden sich immer auf den nachfolgenden Plätzen oder auf Platz 1 und Folgende, wenn es keinen Query mit der Punktzahl 3 gibt.
Die Queries mit der Punktzahl 0 sind immer die zuletzt aufgeführten. Es gibt keine Ausnahmen.
Das spricht dafür, dass die Bewertungsfunktion mit den zugrunde liegenden Informationen über Score, Datasources und Instances in der Lage ist, zu erkennen, ob ein Query relevant ist oder nicht.

Stop Words
Durch das Filtern von Stop Words erzielen auch solche Eingaben 3 Punkte, die auf den ersten Blick zu komplex aussehen. Die Eingabe "Film whose Director is an Actor" erzielt ein genau so gutes Ergebnis wie die im besten Fall zu erwartende Eingabe "Film Director Actor". Einer der Teilnehmer hat besonders viele Stop Words genutzt, um seine Abfrage sprachlich genau zu formulieren. Alle diese Abfragen erzielten 3 Punkte.

Alternative Abfragen
Von den maximal 10 möglichen Queries, die aus einer Eingabe resultieren, sind im Schnitt 8 unbrauchbar. Die 2 brauchbaren Queries sind – wie schon erwähnt – auch immer diejenigen Queries, die an den ersten Stellen positioniert sind. Das bedeutet, man bräuchte gar nicht so viele Anfragen an Lodatio schicken, sondern könnte sich viel mehr auf die Score verlassen und stattdessen nur die Top 3 weiter auswerten. Das beweist auch die 3-Punkte Wertung auf alle Queries. Diese liegt bei lediglich 10%.

Nutzerverhalten
Es ist zu beobachten, dass Nutzer ihre Abfragen sehr verschieden ausdrücken. Das ist eine Erkenntnis, die man schon in drei Testfällen beobachten kann. Während ein Nutzer die Frage nach "Gib mir alle Personen" mit "persons" ausdrückt, beschreibt sie ein anderer mit "persons alive dead list".
Da im konzeptionellen Teil davon ausgegangen wird dass ein Nutzer präzise und aussagekräftige Keywords für die Suche verwendet, erzielen Nutzer, die ein solches Eingabeverhalten widerspiegeln, entsprechend bessere Ergebnisse als solche, die aus diesem Muster ausbrechen. Ein nach dem Nutzertest vollzogener Test, in dem nur exakt die Keywords eingegeben wurden, die für die Suche relevant sind, bestätigt diese Vermutung. So ist die 3-Punkte-Quote in diesem Test bei exakt 100%, also jeder Query hat in seiner Query-Liste mindestens einen Query, der 3 Punkte erzielt und auch der relevanteste ist.
Das strikte Arbeiten auf der Schema-Ebene basiert zudem auf der Annahme, dass Nutzer nur solche Wörter eingeben, die direkt zu ihrer Frage gehören und keine Wörter, die die Antwortmenge weiter eingrenzen sollen und auch keine Stop Words, zum Beispiel "alive dead list". Die Eingabe solcher

Wörter führt dazu, dass nach Elementen gesucht wird, die *alive, dead und list* ausdrücken. Sowohl "alive" als auch "dead" ist als Eigenschaft einer Person möglich, "list" dagegen nicht. Daher kann für so eine Eingabe kein positives Ergebnis erzielt werden. Die uneingeschränkte Eingabemöglichkeit, die in dieser Arbeit bewusst möglich sein soll, führt gerade durch die verschiedenen Verhaltensweisen zu schlechten oder gar keinen Ergebnissen. Das Eingabeverhalten von Personen, die relativ schlechte Ergebnisse erzielen sollte weiter untersucht werden, um unter Umständen Muster zu erkennen und aus diesen Mustern weitere Wege zu finden, das Übersetzen von natürlicher Sprache auf SPARQL weiter zu verbessern.

Keyword-Eingaben, die der subjektiven Korrektheit entsprechen
Interessant ist die Auswertung nach solchen Eingaben, die der subjektiven Korrektheit entsprechen. Das sind Keyword-Eingaben, die vom Autor als richtig angesehen werden – unabhängig davon, ob das Ergebnis richtig ist.
Von den 48 Keyword-Abfragen entsprechen 27 Abfragen der subjektiven Korrektheit. Von den 27 Abfragen haben 20 eine Punktzahl von 3, also 74%. Die anderen 7 Abfragen haben eine Punktzahl von 1, also 26%. Keine Abfrage erzielt 0 Punkte.
Das legt den Schluss nahe, dass bei einer ausschließlichen Nutzung der relevanten Keywords die Ergebnisse gut bis sehr gut sind.

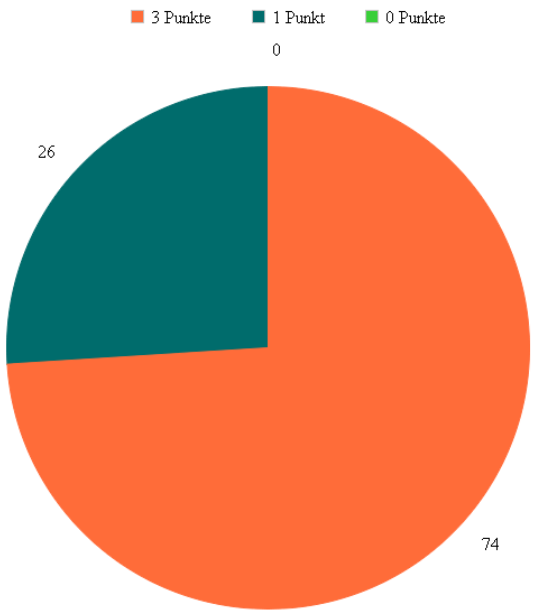


Abbildung 21: Verteilung Abfragen subjektiver Korrektheit nach maximaler Bewertung

Reaktionszeit
43 von 48 Anfragen benötigen im Schnitt weniger als 4 Sekunden bis zur

Anzeige einer Antwort. Das ist grundsätzlich positiv, jedoch gibt es einzelne Abfragen, die aufgrund der Menge zutreffender Schema-Elemente oder der zugrunde liegenden Daten länger als 120 Sekunden benötigten. Bei der Eingabe von mehr als 5 Wörtern, die keine Stop Words sind, benötigt die Suche mindestens 60 Sekunden. Das ist zu viel. Gründe hierfür sind zum Einen die Lodatio-API und zum Anderen die Vorgehensweise zum Erstellen der SPARQL-Queries. Durch das Konkadinieren von SPARQL-Pattern zu SPARQL-Queries entsteht bei Keywords mit vielen dazu passenden Schema-Elementen eine extrem lange Liste von Queries und das wirkt sich negativ auf die Performance aus.

5. Fazit

Diese Arbeit hat gezeigt, dass das Interpretieren von keyword-basierten Anfragen über das Schema funktioniert. Es erzielt zudem relativ gute Ergebnisse. So konnten Probleme wie das Erkennen von zusammengesetzten Wörtern auf Schema-Ebene behandelt und durch Einsetzen einer Stop Word – Liste verfeinert werden.

Die größten Herausforderungen dieser Vorgehensweise sind

- die Performance
- das Nutzerverhalten
- ein "vollständiger" Index

Der Flaschenhals der Performance ist das wiederholte Anfragen des Index und das Anfragen des Schemex über Lodatio. Das Erstellen von SPARQL-Queries dauert bei einer großen Menge von Schema-Elementen zu lange und kann ebenfalls verbessert werden. Allerdings ist das vernachlässigbar im Vergleich zu den beiden erst genannten Punkten.
Die Performance ließe sich durch eine Optimierung des Index verbessern. Je präziser der Index Types und Properties findet, desto weniger Anfragen müssen an Lodatio gesendet werden. Außerdem könnte man ein Score-Limit für den Index einführen. Nur solche Einträge, die mehr als eine Score x erzielen, werden zurückgegeben.
Der Einsatz von Query Templates wurde aufgezeigt. Auch dieser könnte die Performance verbessern. Wenn statt riesiger kartesischer Produkte über eine große Menge von SPARQL-Queries Templates angefragt werden, sinkt der Aufwand zum Erstellen der Abfragen. Außerdem kann man sich bei Query Templates sicher sein, dass sie valide sind und ein Resultat erzielen.
Viel komplexer wäre der Ansatz, Anfragen zu cachen und abzuspeichern. Im Cache könnten zusätzliche Informationen wie die Score oder die Relevanz abgespeichert werden. So müssten gleiche Anfragen nicht immer und wieder an Lodatio gesendet werden, sondern zunächst untersucht werden, ob das Resultat der Anfrage im Cache vorhanden ist.

Das Nutzerverhalten kann man nicht verbessern. Man kann das Nutzerverhalten nur untersuchen, Muster erkennen und Problemlösungen für diese Muster ausarbeiten. Besondere Probleme machen Umschreibungen der eigentlichen Frage. Hier ist der Ansatz meist ratlos und erzielt keine relevanten Ergebnisse. Das könnte eventuell durch einen größeren Index oder gezieltere Suchtexte verbessert werden.

Eine Erweiterung des Index ist immer möglich. Sie könnte durch das Suchen weiterer Vokabeln erfolgen oder durch das Erstellen eines eigenen umfangreichen Vokabelsets. Der Index ließe sich dadurch optimieren, dass man Einträge gezielt mit Informationen wie Synonymen oder verwandten Wörtern optimiert und lange Beschreibungstexte mit irreführenden Inhalten

entfernt. Die Dokumente hätten dann keine echten Beschreibungstexte mehr, sondern vielmehr eine Aneinanderreihung von Tags.

Quellen

[1] Thomas Gottron, Ansgar Scherp, Bastian Krayer, Arne Peters. *LODatio: A Schema-Based Retrieval System for Linked Open Data at Web-scale, S. 1-5,* 2013.

[2] Mathias Konrath, Thomas Gottron, and Ansgar Scherp. *SchemEX—Web-Scale Indexed Schema Extractionof Linked Open Data*, S. 1-8, 2012.

[3] Giovanni Tummarello, Richard Cyganiak, Michele Catasta, Szymon Danielczyk, Renaud Delbru, Stefan Decker*, Sig.ma: live views on the Web of Data,* S. 9-17, 2010.

[4] Saeedeh Shekarpour, Axel-Cyrille Ngonga Ngomo, Soren Auer. *Query Segmentation and Resource Disambiguation Leveraging Background Knowledge*. Department of Computer Science, University of Leipzig, S. 1 – 15, 2013.

[5] K. Q. Pu and X. Yu. Keyword query cleaning. PVLDB, 1(1), S. 909-920, 2008.

[6] Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, Gerhard Weikum. *Language-model-based Ranking for Queries on RDF-Graphs*, 2009, S. 1-12, 2012.

[7] Veli Bicer, Thanh Tran, Radoslav Nedkov. *Ranking Support for Keyword Search on Structured Data using Relevance Models,* S. 1669 – 1678, 2011.

[8] Thanh Tran, Haofen Wang, Sebastian Rudolph, Philipp Cimiano. *Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data,* S.1-12 2009.

[9] Gideon Zenz, Xuan Zhou, Enrico Minack, Wolf Siberski, Wolfgang Nejdl. *From Keywords to Semantic Queries - Incremental Query Construction on the Semantic Web*, S. 1- 19, 2009.

[10] Gjergji Kasneci, Maya Ramanath, Mauro Sozio, Fabian M. Suchanek, Gerhard Weikum. *STAR: Steiner-Tree Approximation in Relationship Graphs,* S.1-12, 2010.

[11] Gideon Zenz, Xuan Zhou, Enrico Minack, Wolf Siberski, Wolfgang Nejdl. *From Keywords to Semantic Queries — Incremental Query Construction on the Semantic Web,* S. 2-3, 2009

[12] Sandeep Tata, Guy M. Lohman, *SQAK: Doing More with Keywords,* S. 889 – 900, 2008.

[13] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, S. Sudarshan. *Keyword Searching and Browsing in Databases using BANKS,* S. 1 - 10, 2002.

[14] V. Hristidis and Y. Papakonstantinou. *DISCOVER: Keyword search in relational databases,* S. 1 – 12, 2002.

[15] S. Agrawal, S. Chaudhuri, G. Das. *DBXplorer: A System for Keyword-Based Search over Relational Databases,* S. 5–16, 2002.

[16] S. Amer-Yahia, E. Curtmola, A. Deutsch. *Flexible and efficient XML search with complex full-text predicates,* S. 575–586, 2006.

[17] Mathias Konrath, Thomas Gottron, Steffen Staab, Ansgar Scherp. *SchemEX – Efficient Construction of a Data Catalogue by Stream-based Indexing of Linked Data,* S. 1, 2012.

[18] W. John Wilbur and Karl Sirotkin. *The automatic identification of stop words*, S. 45-55. 1991.

Anhang

Werkzeuge

Symfony2

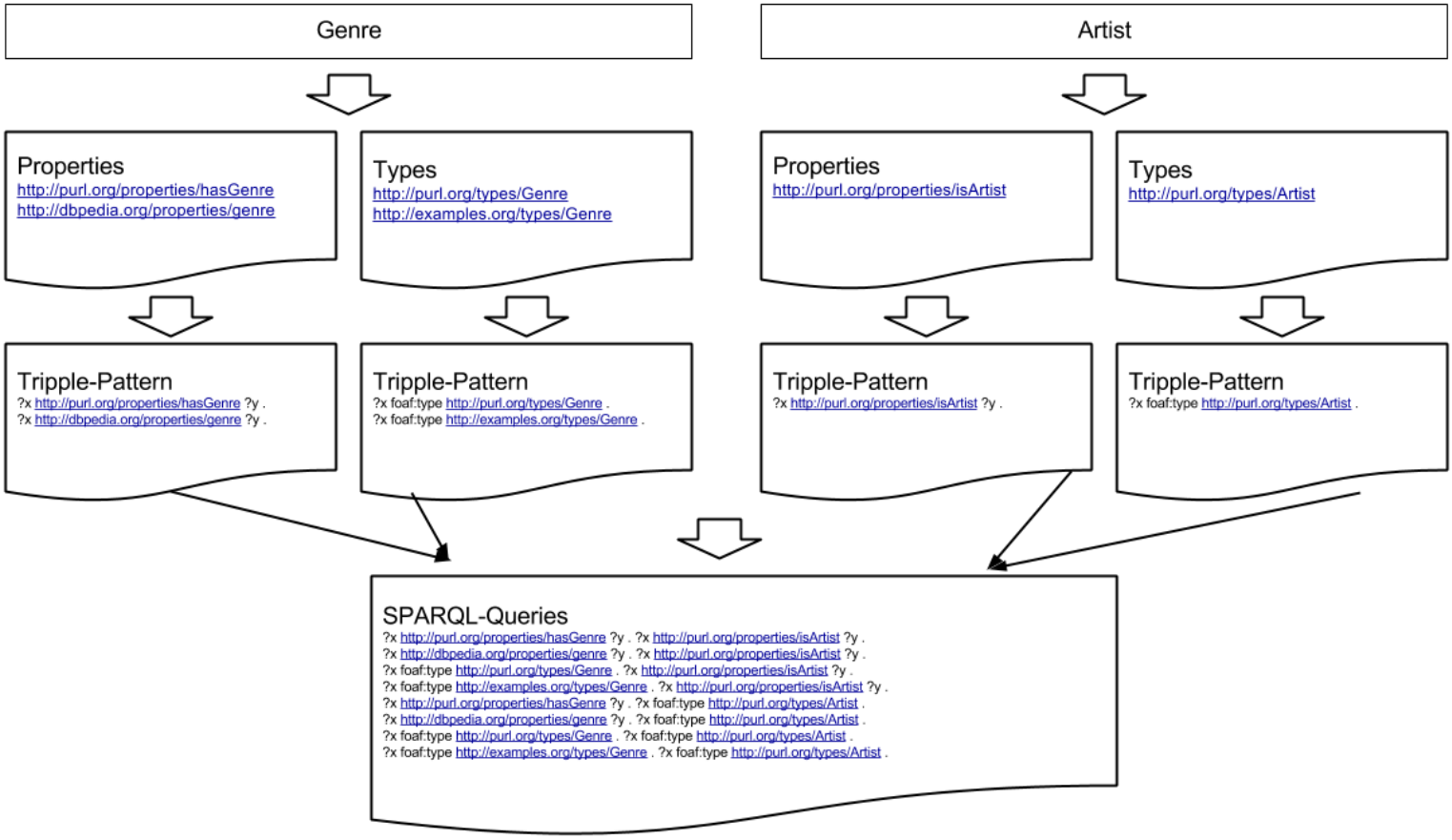
Für die Implementierung wird das PHP Framework Symfony2 verwendet[16]. Symfony2 enthält eine durchdachte Request-Response Architektur und ermöglicht eine saubere Implementierung für Webservices. Da diese Bachelorarbeit am Ende eine Weboberfläche mit Suchfunktion anbietet, bietet es sich an, direkt ein Framework und eine Sprache zu verwenden, die genau darauf zugeschnitten sind. Außerdem gibt es für Symfony2 einige Solr-Plugins, die SPARQL-Abfragen direkt auf Objekte mappen. Das vereinfacht das Aufbereiten der Sparql-Resultsets.

PHP

Da Symfony in PHP implementiert ist, werden alle Code-Beispiele in PHP dargestellt.

16 http://symfony.com/

A1: Erstellen von SPARQL-Queries mit Kantenlänge 1



A2: Implementation Tokenisierungsfunktion

```
private function tokenize($keywordString) {
	$keywords = explode(' ', $keywordString);
	if (count($keywords) === 1) return $keywordString;
	for ($i=0;$i<count($keywords)-1;$i++) {
		$result = $this->retrieve("url_t:$keywords[$i]
$keywords[$i+1]");
		if ($result->count() > 0) {
			$keywordString = str_replace(
				$keywords[$i] . ' ' . $keywords[$i+1],
				$keywords[$i] . $keywords[$i+1],
				$keywordString);
	}
	return $keywordString;
}
```

A3: Implementation Retrievalfunktion

```
private function retrieve($type, $query) {
	$solrQuery = $this->solrClient
			->createQuery($type)
			->setRows(5);
	$query->setCustomQuery($query);

	$result = $query->getResult();
	$return = new ArrayCollection();
	foreach ($result as $res) {
		if ($return->get($res->getUrl()) != null
			|| !$this->isUrl($res->getUrl()))
				continue;

		$return->set($res->getUrl(), $res);
	}
	return $return;
}
```

A4: Implementation Erstellung SPARQL-Pattern

```
private function createSparqlPattern($subject, $predicate,
$object) {
	$predicate = empty($predicate) ? $object->getType() :
"<$predicate>";
	$object = empty($object) ? '?y' : "<$object>";
	return "$subject $predicate $object .";
}
```

A5: Implemenation Erstellung von Queries

```
public function getQueries() {
	$queries = new ArrayCollection();
	foreach ($this->properties as $property) {
		foreach ($this->types as $type) {
			$query = array(
				'score' => ($type->getScore() +
$property->getScore()) * 0.1,
				'query' => $this->createSparqlPattern('?
x', $property
                                                        ->getUrl(),
$type->getUrl()));
                    if (!$queries->contains($query)) $queries-
>add($query);
                }
                $query = array(
                    'score' => $property->getScore(),
                    'query' => $this->createSparqlPattern('?x',
$property->getUrl(), ''));
                if (!$queries->contains($query)) $queries-
>add($query);
	}
	// same with types as first loop and properties as second
one
	return $queries;
}
```

A6: Implemenation Triple-Pattern zu SPARQL-Queries

```
private function concatQueryLists($queryList, $queryList2) {
	if ($queryList2->isEmpty()) return $queryList;
	if ($queryList->isEmpty())  return $queryList2;

	$newQueryList = new ArrayCollection();
	foreach ($queryList as $query) {
		$scoreBase = $query['score'];
		$query = $query['query'];
		foreach ($queryList2 as $query2) {
			$score = $scoreBase + $query2['score'];
			$query2 = $query2['query'];
			$sameQuery = preg_match('/'.preg_quote($query,
'/').'/', $query2);
			$newQuery = $sameQuery ? $query2 : ($query2 . '
' . $query);
			$newQuery = array(
		'score' => $sameQuery ? $score*2*($query2 === $query ?
5 : 1) : $score,
		'query' => $newQuery
			);
			if (!$newQueryList->contains($newQuery))
				$newQueryList->add($newQuery);

			$query3 = str_replace('?x', '?y',
str_replace('?y', '?z', $query));
			$stillSameQuery =
preg_match('/'.preg_quote($query3,'/').'/',

		 $query2);
			$newQuery = $stillSameQuery ? $query2 :
($query2 . ' ' . $query3);
			$newQuery = array('score' => $sameQuery ?
$score*10 : $score,

'query' => $newQuery);
			if (!$newQueryList->contains($newQuery))
				$newQueryList->add($newQuery);
		}
	}
	return $newQueryList;
}
```

A7: Implementation Lodatio-Request

```
public function send($query) {
	$url = 'http://lodatio.west.uni-
koblenz.de:8080/MysqlLookup/CombiLookup?limit=5000&query=' .
urlencode($query);
	$result = json_decode($this->curl->setUrl($url)->execute());

	if (!empty($result->Status) && $result->Status === 'ERROR')
{
		$result->Result = array();
	}

	return $result->Result;
}
```

A8: Implementation Bewertungsfunktion

```
$res['totalScore'] = ($res['score'] * $scoreCoefficient /
$highestScore)
                + ($res['datasources'] *
$datasourcesCoefficient / $mostDatasources)
                + ($res['instances']   * $instancesCoefficient   /
$mostInstances);

if ($res['datasources'] > 0 && $res['instances'] > 0)
      array_push($resultSet, $res);
}
```

A9: Nutzertest

SPARQL	Keyword Query

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x
WHERE {
?x rdf:type foaf:Person .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT ?x
WHERE {
?x rdf:type dbpedia:Actor .
}	

PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT ?x
WHERE {
?x dbpedia:capital ?y .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT ?x
WHERE {
?x rdf:type dbpedia:Airport .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT ?x
WHERE {
?x dbpedia:capital ?y .
?y rdf:type dbpedia:Country .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT ?x
WHERE {
?x rdf:type dbpedia:Country .
?x dbpedia:governmentType ?y .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x
WHERE {
?x rdf:type foaf:Person .
?x foaf:skypeID ?y .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>	

PREFIX purl: <http://purl.org/ontology/wo/>
SELECT ?x
WHERE {
?x rdf:type purl:Family .
?x purl:genus ?y .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x
WHERE {
?x rdf:type foaf:Person .
?x foaf:knows ?y .
?y rdf:type foaf:Person .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX purl: <http://purl.org/ontology/po/>
SELECT ?x
WHERE {
?x rdf:type purl:Series .
?x purl:genre ?y .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT ?x
WHERE {
?x rdf:type dbpedia:Actor .
?x dbpedia:spouse ?y.
?y rdf:type dbpedia:Actor .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ontology: <http://dbpedia.org/ontology/>
PREFIX property: <http://dbpedia.org/property/>
SELECT ?x
WHERE {
?x rdf:type ontology:Film .
?x property:director ?y .
?y rdf:type ontology:Actor .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia: <http://dbpedia.org/ontology/>
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX dbpyago: <http://dbpedia.org/class/yago/>
SELECT ?x
WHERE {
?x rdf:type dbpedia:Film .
?x dbpprop:starring ?y .
?y rdf:type dbpyago:20th-centuryActors
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>	

PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT ?x
WHERE {
?x rdf:type dbpedia:Hotel .
?x dbpedia:numberOfRooms ?y .
}	

PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT ?x
WHERE {
?x dbpedia:railwayLineUsingTunnel ?y .
}	

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT ?x
WHERE {
?x rdf:type dbpedia:Building .
?x dbpedia:rebuildingDate ?y .
}	