Ein System zur Präsentation und Visualisierung von Linked Open Data in Webseiten mittels Semantic Data Widgets

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft
der Universität Duisburg-Essen
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von
Timo Stegemann
aus
Oberhausen

1. Gutachter: Prof. Dr.-Ing. Jürgen Ziegler
2. Gutachter: Prof. Dr. rer. nat. Gregor Schiele

Datum der mündlichen Prüfung: 29.06.2022

Zusammenfassung

Die Menge der öffentlich verfügbaren verlinkten Daten wächst seit den Anfängen des Semantic Webs zu Beginn des 21. Jahrhunderts kontinuierlich an. Die Linked Open Data Cloud, in welcher verschiedene Datensätze zusammengetragen werden, enthält mit über 200 Milliarden Relationen einen unfassbaren Wissensschatz, auf den jede Internetnutzerin und jeder Internetnutzer frei zugreifen kann. Doch auch wenn diese Daten »öffentlich« und »frei« sind, bedeutet dies nicht, dass sie von allen Menschen genutzt werden können. Die enorme Komplexität der zur Nutzung benötigten Techniken, Anwendungen und Formate verhindert, dass ein größerer Teil der Internetnutzerinnen und -nutzer an diesem Wissensschatz teilhaben kann.

Diese Dissertation setzt sich mit dem genannten Problem auseinander und beschreibt die Semwidg-Werkzeug-Suite. Deren Ziel ist es, zumindest Nutzern mit Kenntnissen aus dem Bereich der Web-Technologien, aber ohne Fachkenntnisse aus dem Linked-Data-Bereich, einen einfachen Zugang zu diesen Daten zu ermöglichen und so eine Mehrfachverwendung (*Content-Syndication*) von Linked-Data-Inhalten zu fördern. Semwidg ist ein System zur Präsentation und Visualisierung von Linked Open Data in Webseiten mittels Semantic Data Widgets. Diese Widgets kapseln Datenabfragen und präsentationsrelevante Informationen, die dazu benötigt werden, Daten aus der Linked Open Data Cloud abzurufen, sie zu verarbeiten und anschließend innerhalb von Webseiten anzuzeigen. Semwidg besteht aus den drei Teilkomponenten SemwidgQL, SemwidgJS und SemwidgED.

SemwidgQL ist eine leicht zu erlernende Pfadabfragesprache, die sich hinsichtlich ihrer Syntax und Semantik an den Vorkenntnissen von Webentwicklern und Programmieren orientiert und sich clientseitig in SPARQL – die De-Facto-Standardabfragesprache von Linked-Data-Inhalten – transkompilieren lässt. Anhand zweier Studien konnte gezeigt werden, dass SemwidgQL leichter als SPARQL zu erlernen ist und effektiv und effizient genutzt werden kann.

SemwidgJS ist eine JavaScript-Widget-Bibliothek zur Präsentation und Visualisierung von Linked Data in Webseiten. Durch eine Reihe vordefinierter Widgets-Klassen werden die üblichen Web-UI-Elemente für Aus- und Eingabe sowie Diagramme und Karten zur Verfügung gestellt. Die Widgets können untereinander Daten austauschen, wodurch sich auch interaktive Webseiten erstellen lassen. Ebenso ist SemwidgJS ein Framework, welches die Definition eigener Widget-Klassen mit wenigen Zeilen JavaScript-Code erlaubt.

SemwidgED ist ein Online-WYSIWYG-Editor, welcher die Nutzer bei der Einbindung von SemwidgJS-Widgets in Webseiten und der Formulierung von SemwidgQL-Abfragen unter-

stützt. Die Gebrauchstauglichkeit von SemwidgED wurde in einer empirischen Nutzerstudie hinsichtlich Effektivität, Effizient und Nutzerzufriedenheit untersucht. Hierbei erreichte SemwidgED bei der relevanten Zielgruppe sehr gute Ergebnisse.

Abstract

The amount of publicly available linked data has been growing steadily since the beginning of the Semantic Web at the start of the 21st century. With over 200 billion relations, the Linked Open Data Cloud, in which various data sets are gleaned, comprises an inconceivable treasure trove of knowledge that can be freely accessed by every Internet user. However, even though this data is "public" and "free", this does not mean that it can also be used by everyone. The enormous complexity of the technologies, applications and formats required for using this data, prevents the majority of Internet users from participating in this knowledge.

This dissertation addresses the aforementioned problem and describes the Semwidg tool suite. Its goal is to provide easy access to this data, at least for users with knowledge of web technologies but without expertise in Linked Data, and thus to promote multiple use (*Content-Syndication*) of Linked Data content. Semwidg is a system for presenting and visualizing Linked Open Data in web pages using Semantic Data Widgets. These widgets encapsulate database queries and presentation-relevant information needed to retrieve data from the Linked Open Data Cloud, process it, and then display it within web pages. Semwidg consists of the three subcomponents SemwidgQL, SemwidgJS and SemwidgED.

SemwidgQL is an easy-to-learn path query language that is adapted to the background knowledge of web developers and programmers in terms of its syntax and semantics. It can be transcompiled on the client-side into SPARQL, the de facto standard query language of Linked Data content. Based on two studies, it has been shown that SemwidgQL is easier to learn than SPARQL and can be used effectively and efficiently.

SemwidgJS is a JavaScript widget library for presenting and visualizing Linked Data in web pages. It provides the usual web UI elements for output and input, as well as charts and maps, through a set of predefined widget classes. The widgets can exchange data with each other, allowing interactive web pages to be created. Furthermore, SemwidgJS is a framework that allows to define custom widget classes with just a few lines of JavaScript code.

SemwidgED is an online WYSIWYG editor that supports users in embedding SemwidgJS widgets in web pages and formulating SemwidgQL queries. The usability of SemwidgED was evaluated in an empirical user study with respect to effectiveness, efficiency and user satisfaction. SemwidgED achieved very good results within the relevant target group.

Danksagung

An dieser Stelle möchte ich mich ganz herzlich bei all jenen bedanken, die mich bei meiner Arbeit unterstützt haben.

Mein besonderer Dank gilt Jürgen Ziegler, der mir für viele Jahre ermöglichte, Teil seines Teams zu sein. Ich konnte in dieser Zeit sehr viel lernen und stünde ohne meine dort gemachten Erfahrungen und mein dort gesammeltes Wissen nicht da, wo ich heute stehe.

Weiterhin möchte ich mich bei Gregor Schiele dafür bedanken, dass er die Zweitbegutachtung meiner Arbeit übernommen hat.

Ein weiterer Dank gilt meinen (ehemaligen) Kollegen und gleichzeitig auch Freunden, auf deren Rat und Unterstützung ich mich immer verlassen konnte. Speziell möchte ich mich hier bei Werner Gaulke (als thematisch nächstgelegener Kollege) und Benedikt Loepp (als räumlich nächstgelegener Kollege) bedanken, die für viele Jahre meine ersten Ansprechpartner waren. Ebenso möchte ich mich bei Steffen Lohmann und Philipp Heim bedanken, die mich während meiner Zeit als studentische Hilfskraft anleiteten, mit denen ich meine ersten wissenschaftlichen Publikationen verfasst habe und auf deren Vorarbeiten die hier vorliegende Arbeit in Teilen aufbaut.

Ganz besonders bedanken möchte ich mich bei Katja Herrmanny für ihre Unterstützung bei der Konzeption und statistischen Auswertung der im Rahmen dieser Arbeit durchgeführten Nutzerstudien sowie für ihr ausgiebiges Korrekturlesen.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Seit den Anfängen des Semantic Webs (vgl. Berners-Lee et al., 2001) wächst die Menge frei verfügbarer, in Graph-Form verlinkter Daten kontinuierlich an (vgl. Mika & Potter, 2012; Ermilov et al., 2013; Schmachtenberg et al., 2014). Ein Teil dieser Daten findet sich in der Linked Open Data Cloud (LOD-Cloud)[1] wieder, in der RDF-Datensätze[2] aus verschiedenen Domänen zusammengetragen werden. Während die LOD-Cloud in ihrer ersten Version aus dem Jahr 2007 aus lediglich 12 miteinander verknüpften Datensätzen bestand, wuchs sie auf mittlerweile 1300 Datensätze im Jahr 2021 an. Die Datensätze liegen als herunterladbare RDF-Dateien vor oder können über eine einheitliche Web-Schnittstelle mittels der RDF-Abfragesprache SPARQL[3] abgefragt werden. Diese Schnittstelle ist bei etwa der Hälfte der Datensätze verfügbar und ermöglicht den direkten Zugriff auf über 160 Milliarden der insgesamt über 200 Milliarden Subjekt-Prädikat-Objekt-Relationen beziehungsweise RDF-Tripel der LOD-Cloud.

Die Menge der nicht frei zugänglichen verlinkten Daten dürfte noch um einiges größer ausfallen. So verfügen IT-Konzerne wie etwa Google, Amazon, Apple, Microsoft und Facebook über enorme Datenbestände, die sie intern für ihre Dienste nutzen, aber nicht frei zur Verfügung stellen. Weitere kommerzielle Anbieter von Linked-Data-Systemen wie zum Beispiel OpenLink Software[4] oder Poolparty[5] (vgl. Schandl & Blumauer, 2010) beliefern unter anderem eine Vielzahl großer Firmen aus den Bereichen Industrie, Telekommunikation, Verlagswesen, E-Commerce, Pharmazie und Finanzen, welche Linked Data ebenfalls intern einsetzen. Die BBC nutzte Linked Data beispielsweise für die Berichterstattung zur Fußball-Weltmeisterschaft 2010 und zu den Olympischen Sommerspielen 2012 in einer eigens entwickelten Publikationssoftware[6]. In anderen Bereichen greift die BBC auf Inhalte der LOD-Cloud zu, um ihre Daten mit weiteren Informationen anzureichern (vgl. Kobilarov et al., 2009).

Eine Nutzung von Linked-Data im privaten und semiprofessionellen Bereich, außerhalb großer Konzerne und Forschungseinrichtungen, findet hingegen kaum statt.

[1] https://lod-cloud.net
[2] https://www.w3.org/TR/rdf11-concepts/
[3] https://www.w3.org/TR/sparql11-overview/
[4] https://www.openlinksw.com
[5] https://www.poolparty.biz
[6] https://www.bbc.co.uk/blogs/bbcinternet/2012/04/sports_dynamic_semantic.html

1.1 Motivation

Während sich Linked Data in einigen Bereichen immer weiter durchsetzen, findet es im privaten und semiprofessionellen Bereich durch Webentwickler und -autoren jedoch kaum Beachtung. Obwohl mit der LOD-Cloud eine enorme Datenmenge allen Nutzern des World Wide Webs frei zur Verfügung steht, werden die sich daraus ergebenden Möglichkeiten zur Wieder- und Weiterverwendung der Daten nicht einmal ansatzweise ausgeschöpft.

Dabei bietet eine Mehrfachverwendung von Inhalten (*Content-Syndication*) auch für private und semiprofessionelle Webseitenbetreiber viele Vorteile. Statische Texte oder auch Informationen aus Datenbanken, welche nicht ständig auf dem neusten Informationsstand gehalten werden, sind schnell veraltet. Doch die Aktualisierung dieser Informationen ist aufwendig und der Arbeitsaufwand wächst mit jedem Artikel, Blog-Post oder sonstiger Veröffentlichung auf der Webseite. Werden jedoch potenziell veränderliche Daten nicht direkt innerhalb der Seite gespeichert, sondern aus externen Quellen wie der LOD-Cloud bezogen, kann dieser Aufwand auf eine größere Community verteilt werden, durch automatisierte Dienste übernommen werden, die Informationen aus anderen Webseiten extrahieren oder, je nach Bereitsteller des Datensatzes, sogar direkt aus erster Hand erfolgen.

Nicht nur der Betreiber einer Webseite sollte ein Interesse daran haben, dass Daten auf seiner Seite stets aktuell sind. Auch andere Akteure, die dort erwähnt werden, können ein solches Interesse aufweisen. Wird innerhalb eines Blogs beispielsweise über eine Band berichtet, könnte sich der Autor dazu entschließen, die ihm bekannten Konzerttermine der Band und bereits veröffentlichte Alben einzutragen. Es ist jedoch nicht davon auszugehen, dass der Autor diese Daten auch nachträglich auf dem neusten Stand hält, womit sie bereits nach kurzer Zeit für viele Leser irrelevant werden. Trägt der Autor jedoch keine statischen Daten auf seiner Webseite ein, sondern bindet sie aus einer externen, dauerhaft aktuell gehaltenen Quelle ein, sind auf seiner Webseite damit auch noch Jahre später relevante Informationen vorzufinden. Gegebenenfalls erfährt ein Leser dadurch, dass die Band in Kürze in seiner Umgebung auftritt oder er erfährt von einem Album, welches er noch nicht kannte. Die Band kann dadurch möglicherweise die Zahl verkaufter Konzerttickets oder Alben erhöhen. Es liegt in diesem Beispiel also auch im Interesse der Band und des dahinterstehenden Musikverlages, dass ständig aktuelle Informationen bereitgestellt werden.

Ein möglicher Grund, warum eine Mehrfachverwendung von Linked Open Data kaum stattfindet, könnte sein, dass Betreiber von Webseiten schlicht keine Inhalte aus fremden Quellen in diese einbinden möchten, um sich beispielsweise nicht von diesen abhängig zu machen. Dies ist jedoch unwahrscheinlich, da ein solches Vorgehen sowohl auf kommerziellen wie auch auf privaten Webseiten häufig stattfindet. Twitter- und Facebook-Kommentare werden in diese eingebunden und ehemals interne Kommentarfunktionen werden vollständig an externe Dienstleister wie Disqus[7] ausgegliedert. Ähnliches gilt für Umfragen (z. B. Civey[8]) und Datenvisualisierungen (z. B. IBM ManyEyes, vgl. Viegas et al., 2007). Insbesondere geographische Kartendaten (z. B. Google Maps[9] oder OpenStreetMap[10]) und Werbeanzeigen werden ganz selbstverständlich aus externen Quellen bezogen. Inhalte aus fremden Quellen können daher in vielen Fällen als akzeptiert betrachtet werden.

[7] https://disqus.com
[8] https://civey.com
[9] https://maps.google.com
[10] https://www.openstreetmap.org

Der Hauptgrund für eine fehlende Content-Syndication im Linked-Open-Data-Bereich, dürfte vielmehr in der Komplexität der damit verbundenen Techniken, Anwendungen und Formate liegen, welche von Webentwicklern und -autoren beherrscht werden müssen, damit diese ihre Ziele zufriedenstellend erreichen können. Mit SPARQL steht eine mächtige, jedoch auch komplexe Abfragesprache für RDF-Daten zur Verfügung, die zwar durch einige übernommene Schlüsselwörter an die Abfragesprache SQL erinnert, welche den meisten Webentwicklern bekannte sein dürfte, sich in ihrer Anwendungsweise jedoch stark von dieser unterscheidet. Da sich das zugrundeliegende Graph-Modell von RDF ebenfalls stark von dem relationaler Datenbanken unterscheidet, können Webentwickler nur bedingt auf bereits bekanntes Wissen zurückgreifen, was zu einem erheblichen Einarbeitungs- und Lernaufwand führt. Vergleichbares gilt für die Software, die für das Einbinden der Daten benötigt wird. Häufig handelt es sich hier um Serveranwendungen, die sich nicht ohne Weiteres in die bereits genutzten Programme zur Publikation von Artikeln, Blog-Posts oder Ähnlichem integrieren lassen, sondern den vollständigen Publikationsprozess übernehmen wollen (vgl. Abschnitt 3.1). Andere Programme oder Software-Bibliotheken erfordern eine manuelle Integration in das bestehende System, was wiederum fortgeschrittene Programmierkenntnisse in der entsprechenden Sprache voraussetzt, wodurch ein Großteil der Webseitenautoren bereits von der Nutzung ausgeschlossen werden. Werden kostenfreie Blogging-Plattformen oder günstiger Webhosting-Angebote genutzt, ist es für deren Nutzer zudem oft nicht möglich, die für das Einbinden von RDF-Daten benötigte Software dort zu installieren.

Der Mehrfachverwendung von Linked Open Data in Webseiten stehen damit zwei große Hindernisse im Weg. Zum einen verfügen die Personen, die diese Daten in Webseiten einbetten würden, nicht über die Kenntnisse hinsichtlich der damit verbundenen Techniken und Formate, um die Daten aus der LOD-Cloud abzurufen, zu verarbeiten und anschließend einzubetten. Zum anderen mangelt es an Werkzeugen, die sich in den bestehenden Publikationsprozess integrieren lassen und den Arbeitsaufwand bei der Einbindung der Daten auf ein angemessenes Maß reduzieren. In dieser Arbeit werden Lösungen für beide Probleme erarbeitet.

1.2 Forschungsfragen und wissenschaftlicher Beitrag

Basierend auf den zuvor vorgestellten Potenzialen der Nutzung von Linked Data und den aufgezeigten Hindernissen, werden nachfolgend die Forschungsfragen benannt, die sich im Vorfeld dieser Arbeit stellten (vgl. Abschnitt 1.2.1) und zu deren Beantwortung ein Konzept zur Präsentation und Visualisierung von Linked Open Data auf Basis von Semantic Data Widgets und die damit verbundene Semwidg-Werkzeug-Suite entwickelt wurden (vgl. Abschnitt 1.2.2). Die Gesamtheit aus Konzept und Werkzeug-Suite wird im Folgenden unter der Bezeichnung »Semwidg-Projekt« zusammengefasst. Nach der Betrachtung der Forschungsfragen werden die potenziellen Nutzer der Semwidg-Werkzeug-Suite mit ihren Fähigkeiten und Bedürfnissen beschrieben und anhand dieser in vier Nutzergruppen eingeordnet (vgl. Abschnitt 1.2.3). Ebenso wird die Hauptzielgruppe des Projekts definiert.

1.2.1 Forschungsfragen

Die beiden zuvor genannten großen Hindernisse bei der Mehrfachverwendung von Linked Open Data in Webseiten – nämlich fehlende Fachkenntnisse der potenziellen Nutzer und ein Mangel an unterstützenden Werkzeugen – führen direkt zu mehreren Forschungsfragen:

1. Wie lassen sich Daten aus der LOD-Cloud auch von Programmierern und Webentwicklern abrufen, die keine Linked-Data-Experten sind?
2. Mit welchen Mitteln und Techniken lassen sich die abgerufenen Daten in reguläre Webseiten einbinden, ohne einen bereits vorhandenen Publikationsprozess oder sogar die Serverinfrastruktur fundamental umzustrukturieren oder zu ersetzen?
3. Wie können Anwender bei der Einbindung der Daten in die Webseite unterstützt werden, um den daraus resultierenden Arbeitsaufwand so weit wie möglich zu reduzieren?

Zur Beantwortung dieser Fragen wurde das Semwidg-Projekt initiiert. Dieses Projekt sowie die Ergebnisse, welche im Rahmen desselben erzielt wurden, werden im weiteren Verlauf dieser Arbeit beschrieben.

1.2.2 Wissenschaftlicher Beitrag

Das Semwidg-Projekt hat zum Ziel, Programmierern und Webentwicklern – also Anwendern, die über ein informatisches Vorwissen verfügen, die jedoch keine Experten auf dem Gebiet von Linked Data und den damit verbundenen Techniken und Anwendungen sind – zu ermöglichen, Daten der LOD-Cloud abzurufen und in Webseiten einzubinden. Um diese Ziele zu erreichen, bedient sich das Semwidg-Projekt des Konzeptes der *Semantic Data Widgets*, welches ihm auch seinen Namen verliehen hat. Dieses Konzept wurde bereits in vorhergehenden Arbeiten entworfen (vgl. Stegemann et al., 2012) und im Rahmen des Semwidg-Projekts weiterentwickelt (vgl. Stegemann & Ziegler, 2014; Ziegler & Stegemann, 2014). Semantic Data Widgets kapseln Datenabfragen und präsentationsrelevante Informationen, die benötigt werden, um Daten aus der LOD-Cloud abzurufen, sie zu verarbeiten und anschließend innerhalb einer Webseite anzuzeigen. Gleichzeitig sind sie wiederverwendbar und lassen sich zum Beispiel durch Parametrisierung der Abfrage auf weitere Daten anwenden.

Um auch Anwendern ohne Fachkenntnisse im Linked-Data-Bereich die Nutzung der Semantic Data Widgets zu ermöglichen, wurde im Rahmen des Semwidg-Projekts angestrebt die Komplexität der damit verbundenen Arbeiten auf drei Ebenen zu reduzieren:

1. Mit SemwidgQL (vgl. Kapitel 2) wurde eine leicht zu erlernende Pfadabfragesprache entwickelt, die sich in ihrer Syntax und Semantik an den technischen Vorkenntnissen von Programmierern und Webentwicklern orientiert. Die Sprache lehnt sich dazu an die Punkt-Notation objektorientierter Programmiersprachen (z. B. Java, C++) an, um den Navigationspfad durch den RDF-Graphen zu beschrieben, ganz so als wären alle RDF-Resources Objektinstanzen mit ausschließlich öffentlich sichtbaren Variablen. Die Variablenbezeichnungen sind mit den Prädikaten des RDF-Graphen gleichzusetzen. Die Werte dieser Variablen sind entweder Literale oder wiederum Instanzen beziehungsweise Resources, die selbst weitere Variablen enthalten können.

   SemwidgQL-Abfragen werden auf Nutzerseite in SPARQL-Abfragen transkompiliert. Somit können sie anschließend an nahezu jeden erreichbaren SPARQL-Endpoint gerichtet werden. Da einige der älteren SPARQL-Endpoints der LOD-Cloud den aktuellen SPARQL-Standard noch nicht vollständig unterstützen, kann es vorkommen, dass manche Abfragen gegebenenfalls nicht verarbeitet werden können.

   Die SemwidgQL-Abfragesprache wurde in Form analytischer Bewertungen und durch eine empirische Nutzerstudie evaluiert. Diese Evaluationsergebnisse wurden zum Teil im Rahmen der »16$^{\text{th}}$ International Semantic Web Conference (ISWC 2017)« vorgestellt.

2. Mit SemwidgJS (vgl. Kapitel 3) wurde eine Widget-Bibliothek und zugleich ein Framework zur Präsentation und Visualisierung von Daten aus öffentlichen SPARQL-Endpoints entwickelt. Diese Bibliothek stellt eine Vielzahl von Widgets für die Ein- und Ausgabe der Daten in Form üblicher Web-UI-Elemente sowie zur Visualisierung von numerischen Daten in Form von Diagrammen bereit. Ebenso lassen sich individuelle, an die anwendungsspezifischen Anforderungen angepasste Widgets programmieren. Widgets können untereinander Daten austauschen oder von außen manipuliert werden, sodass sich mit geringem Aufwand auch interaktive Webseiten erzeugen lassen.

   Widgets werden innerhalb des HTML-Quelltextes der Webseite in Form reguläre HTML-Elemente deklariert. Die SemwidgJS-Bibliothek liest diese automatisch ein, verarbeitet sie und erzeugt an der Stelle, an der das Widget deklariert wurde die entsprechende Datenpräsentation oder Visualisierung. Die gesamte Verarbeitung erfolgt clientseitig durch die JavaScript-Bibliothek und erfordert, abgesehen von den SPARQL-Endpoints, deren Daten abgerufen werden sollen, keine weiteren serverseitigen Dienste.

3. Mit SemwidgED (vgl. Kapitel 4) wurde ein Online-WYSIWYG-Editor entwickelt, der die Nutzer bei der Einbindung von Linked Open Data in ihre Webseiten unterstützt. Ebenso unterstützt der Editor die Nutzer auch bei der Formulierung von SemwidgQL-Abfragen. Für fortgeschrittene Anwender wurde eine textbasierte Eingabeunterstützung implementiert, welche eine möglichst schnelle und effiziente Abfrageerstellung ermöglicht. Eine formularbasierte Eingabe richtet sich an weniger erfahrene Anwender, welche im Rahmen dieser Variante stärker angeleitet werden, wodurch Fehler bei der Formulierung der Abfragen vermieden werden.

   SemwidgED wurde als Plug-In für den freien WYSIWYG-HTML-Editor CKEditor[11] entwickelt, welcher in vielen Content-Management-Systemen und Blogging-Plattformen standardmäßig verwendet wird oder den dort verwendeten Standardeditor ersetzen kann. Sollte die direkte Verwendung nicht möglich sein, kann SemwidgED unabhängig genutzt

[11] `https://ckeditor.com`

werden. Der Quelltext, welcher von SemwidgED erzeugt wird, kann dann exportiert und in die Seite an gewünschter Stelle eingefügt werden.

Die Gebrauchstauglichkeit von SemwidgED wurde im Rahmen einer empirischen Nutzerstudie evaluiert.

Jedes dieser Teilprojekte richten sich direkt an jeweils eine der zuvor gestellten Forschungsfragen und versuchen diese zu beantworten. Zusammen bilden die aus diesen Teilprojekten hervorgegangenen Entwicklungen die Semwidg-Werkzeug-Suite.

1.2.3 Potenzielle Nutzer des Semwidg-Systems

Die potenziellen Nutzer des Semwidg-Werkzeug-Suite lassen sich in verschiedene Gruppen und Untergruppen einordnen. Auf oberster Ebene lassen sie sich in zwei Gruppen aufteilen. Zum einen in einfache Konsumenten von Inhalten, welche Webseiten, die Linked Data mittels SemwidgJS abrufen und anzeigen, betrachten oder auch mit ihnen interagieren. Und zum anderen in Autoren, die diese Webseiten erstellen. Für die Konsumenten ist der Einsatz von SemwidgJS völlig transparent, sodass auf diese nicht weiter eingegangen werden muss. Im weiteren Verlauf wird daher nur noch die Gruppe der Autoren betrachtet und einfachheitshalber der Begriff »Nutzer« synonym für diese genutzt. Diese Gruppe wird im weiteren Verlauf noch stärker differenziert.

Da es die Intention der Nutzer sein wird, Linked Data in ihre Webseiten einzubinden, kann davon ausgegangen werden, dass diese zumindest über ein grundlegendes Verständnis der fundamentalen Web-Publishing-Techniken verfügen und zumindest von SPARQL-Endpoints wie DBpedia[12] gehört haben. Die relevanten hier betrachteten Fähigkeiten für den Einsatz von SemwidgJS und SemwidgQL sind der Grad der Programmierkenntnisse der Autoren sowie der Grad ihre Fachkenntnisse im Linked-Data-Bereich. Daraus lassen sich mindestens vier Untergruppen beziehungsweise archetypische potenzielle Nutzer definieren. Diese stellen die Ausprägungen möglichen Nutzer bezüglich der Programmierfähigkeiten und Kenntnisse der Techniken und Anwendungen des Linked-Data-Bereichs dar (vgl. Abbildung 1.1).

Anfänger: Diese Nutzer sind Laien in den Bereichen Web-Programmierung und Linked Data. Sie können Daten auf Webseiten veröffentlichen, nutzen dazu aber Werkzeuge wie Webseiten-Baukästen und WYSIWYG-Editoren, die sie von den komplexeren Operationen der Web-Programmierung abschirmen. Sie haben bereits von Linked Data gehört und wissen, dass es öffentliche SPARQL-Endpoints wie DBpedia gibt. Sie sind jedoch nicht eigenständig in der Lage SPARQL-Abfragen zu formulieren, um so von den darin enthaltenen Daten zu profitieren.

Programmierer: Diese Nutzer verfügen über fortgeschrittene Programmierkenntnisse. Sie sind jedoch nur wenig mit den Techniken und Anwendungen des Linked-Data-Bereichs vertraut. Da sie über informatisches Vorwissen verfügen, können sie sich aber verhältnismäßig schnell in neue Techniken und Programmiersprachen einarbeiten. Diese Nutzer stehen stellvertretend für die Hauptzielgruppe des Semwidg-Projekts.

Linked-Data-Experten: Diese Nutzer beherrschen die Mehrzahl der Techniken und Anwendungen des Linked-Data-Bereichs und können eigenständig SPARQL-Abfragen formulieren. Sie verfügen jedoch nur über rudimentäre Kenntnisse der Web-Programmierung.

[12] `https://dbpedia.org`

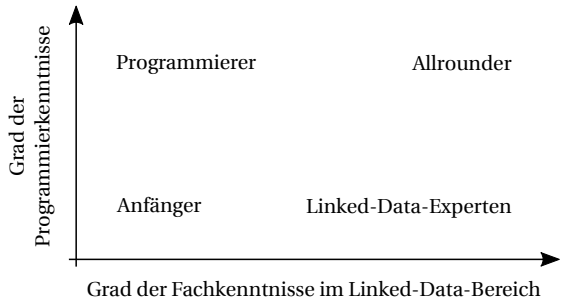


Abbildung 1.1: Die potenziellen Nutzer der Semwidg-Projekts lassen sich anhand des Grads ihrer Programmierkenntnisse und ihrer Fachkenntnisse im Linked-Data-Bereich in vier Gruppen unterscheiden.

Allrounder: Diese Nutzer vereinen jeweils in sich die Kenntnisse der Programmierers und Linked-Data-Experten.

Die realen Anwender der Semwidg-Werkzeug-Suite werden sich zwischen diesen archetypischen Nutzern einordnen lassen. Eine konkrete Einordnung in distinkte Gruppen wird jedoch kaum möglich sein, da die Übergänge zwischen den Anfängern, Programmierern, Linked-Data-Experten und Allroundern fließend sind. Vereinfachend soll hier jedoch davon ausgegangen werden, dass sich jeder potenzielle Nutzer einem archetypischen Nutzer zuordnen lässt, welchem er bezüglich seiner Fähigkeiten am nächsten steht. Im weiteren Verlauf wird daher auch von den vier Gruppen der Anfänger, Programmierer, Linked-Data-Experten und Allroundern gesprochen werden.

Das Semwidg-Projekt wurde in erster Linie auf die Unterstützung der Gruppe der Programmierer ausgerichtet. Diese verfügen bereits über eine Vielzahl der benötigten Fähigkeiten, um Informationen im Internet zu publizieren. Die fehlenden Fähigkeiten dieser Nutzergruppe bezüglich der Techniken und Abfragesprachen des Linked-Data-Bereichs lassen sich leicht mit den entsprechenden Hilfsmitteln ausgleichen, um ihre Mitglieder in die Lage zu versetzen, auch Daten der LOD-Cloud abzurufen, zu verarbeiten und anschließen innerhalb einer Webseite anzuzeigen.

1.3 Grundlagen und allgemeiner Überblick über den Stand der Forschung

Nachfolgend werden mit RDF und SPARQL die grundlegende Beschreibungssprache sowie Abfragesprache für Linked-Data vorgestellt. Es folgt ein allgemeiner Überblick über den aktuellen Forschungsstand in den Bereichen Abfrage, Darstellung und Exploration von Daten aus dem Linked-Data-Bereich. Diese drei Bereiche sind eng miteinander verbunden. Ohne Abfrage ist keine Darstellung und ohne Darstellung ist keine Exploration möglich. Im Gegenzug müssen im Rahmen der Daten-Exploration neue Abfragen erzeugt oder alte Abfragen angepasst werden. Der hier vorgestellte Überblick umfasst Bereiche, die nicht vom Semwidg-Projekt abgedeckt werden und die von diesem auch nicht abgedeckt werden sollen. Allerdings liefern sie wertvolle Informationen und Inspirationen, die in dessen Umsetzung eingeflossen sind. In den jeweiligen Kapitel zu SemwidgQL, SemwidgJS und SemwidgED wird der Forschungsstand des entsprechenden Bereichs detaillierter analysiert. Daher wird hier bei der Betrachtung relevanter Anwendungen und Systeme nur auf einige wenige Stellvertreter eingegangen.

1.3.1 Grundlegende Techniken

Die Daten aus dem Linked-Data-Bereich werden in Graph-Form beschrieben. Dabei sind die Knoten miteinander durch gerichtete und benannte Kanten verbunden. Zwei Knoten und die verbindende Kante bilden ein Triple und beschreiben eine Subjekt-Prädikat-Objekt-Relation. Die beiden Knoten entsprechen dem Subjekt und Objekt, die Kante dem Prädikat. Ein solches Tripel stellt immer eine Aussage dar. Zum Beispiel: »Albert Einstein« (Subjekt) »wurde geboren in« (Prädikat) »Ulm« (Objekt). Subjekte und Prädikate sind immer Ressourcen, welche sich dadurch auszeichnen, dass sie eindeutig durch eine URI definiert sind. Objekte können Ressourcen oder auch nur Literale sein. Objekte, die selbst eine Ressource sind, können wiederum Subjekte eines weiteren Tripels sein. In Kombination können mehrere Triple einen Graphen aufspannen. Die Prädikate werden häufig auch als Eigenschaften beziehungsweise Properties einer Ressource bezeichnet. Das Resource Description Framework (RDF)[13] definiert, wie diese Graphen formal beschrieben werden. Damit bildet RDF die Basis für Linked-Data und das Semantic Web. Die Daten können auf unterschiedliche Weise gespeichert werden. In der Regel werden sie im Textformat in einem RDF-Dokument oder in einer speziellen Datenbank, einem Triple-Store, gespeichert.

Als Standard-Abfragesprache für Daten im RDF-Format hat sich SPARQL[14] (genauer betrachtet in Abschnitt 2.1.1) etabliert. SPARQL ist eine Pattern-basierte Abfragesprache, mit der ebenfalls Subjekt-Prädikat-Objekt-Tripel beschrieben werden. Im Gegensatz zu RDF können jedoch beliebige Teile der Tripel durch Variablen ersetzt werden. Diese Tripel beziehungsweise Graph-Patterns werden mit den RDF-Daten abgeglichen. Passende RDF-Daten bilden die Ergebnismenge der Abfrage. Triple-Stores, die sich mittels SPARQL abfragen lassen, werden auch als (SPARQL-)Endpoints bezeichnet.

1.3.2 Darstellung und Exploration

In den Anfangszeiten von Linked-Data waren nur einfache RDF-Browser wie BrownSauce[15] oder Haystack (Quan et al., 2003) verfügbar, die sich auf die reine Darstellung der RDF-Daten als Tabelle mit anklickbaren Links beschränken. Die Verlinkungen funktionieren nur in eine

[13] https://www.w3.org/TR/rdf11-concepts/
[14] https://www.w3.org/TR/sparql11-overview/
[15] https://www.xml.com/pub/a/2003/02/05/brownsauce.html

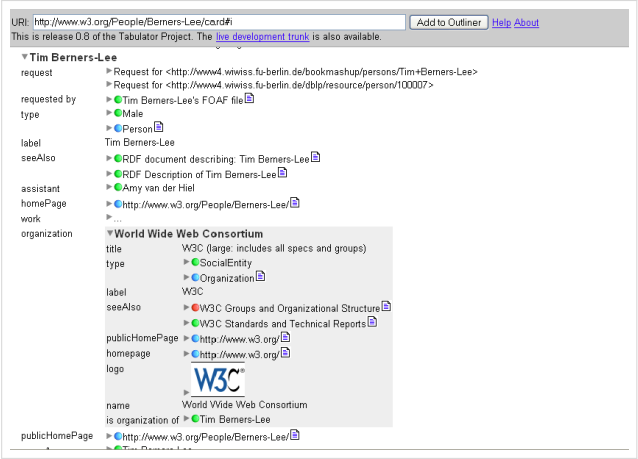


Abbildung 1.2: Ansicht des Tabulator-Browsers (vgl. Berners-Lee et al., 2006). Es werden Daten zu Tim Berners-Lee und verknüpften Ressourcen in einer hierarchisch angeordneten Baumansicht angezeigt. Ressourcen können aufgeklappt werden, um weitere Informationen anzuzeigen.

Richtung, da nur einzelne RDF-Dokumente und keine dokumentenübergreifenden Informationen verarbeitet und angezeigt werden. Entsprechend können nur ausgehende aber keine eingehenden Verlinkungen angezeigt werden. Der Ausgangspunkt zur Exploration der Daten ist hier ein von den Nutzern zu definierendes RDF-Dokument. Von diesem ausgehend können sie über Verlinkungen von einem Dokument zum nächsten navigieren.

Anstatt nur die einzelnen RDF-Dokumente zu laden und anzuzeigen, erzeugen zeitlich später entwickelte generische RDF-Browser wie Tabulator (Berners-Lee et al., 2006) oder Disco[16] intern einen Graphen aus mehreren geladenen Dokumenten. Weiterhin lassen sich die Daten auch mittels SPARQL aus dem so generierten Graphen abrufen. Zudem zeigen diese Browser auch verlinkte Grafiken an und wurden um die Anzeigemöglichkeiten von Datenvisualisierungselementen wie zeitbezogene Ansichten und Karten erweitert. In der Standardansicht transformiert Tabulator den RDF-Graphen in eine hierarchische Baumansicht. Unterelemente lassen sich aufklappen. Farbige Icons zeigen an, ob Unterelemente noch unbekannt sind, bereits geladen wurden oder sich nicht laden ließen (vgl. Abbildung 1.2). Weiterhin verfügt Tabulator über die Möglichkeit, Geokoordinaten innerhalb einer Karte anzuzeigen und zeitbezogene Daten auf einem Zeitstrahl oder innerhalb einer Kalenderansicht darzustellen.

Weitere Datenvisualisierungselementen wurden in nachfolgenden Browsern implementiert. Speziell Fresnel (vgl. Pietriga et al., 2006; genauer betrachtet in Abschnitt 3.1.2) wird von mehreren Browsern als Unterbau genutzt (z. B. Marbles[17]; Arago, vgl. Gassert & Harth, 2005; LENA, vgl. Koch & Franz, 2008; PRISSMA, vgl. Costabello & Gandon, 2014). Mittels Fresnel Stylesheets lassen sich die darzustellenden Elemente bestimmen, zusätzlich um weitere (Meta-)Daten anreichern und gruppieren. Innerhalb des Stylesheets wird noch kein spezifisches Darstellungsparadigma oder Ausgabeformat vorbestimmt. Diese Aufgabe wird erst durch den Browser selbst vorgenommen.

Eine weitere häufig genutzte Darstellungsart für RDF-Daten, die sich bereits dadurch aufdrängt, dass RDF-Daten einen Graphen beschreiben, ist die Graph-Darstellung, wie sie zum

[16] https://web.archive.org/web/20080128155410/http://www4.wiwiss.fu-berlin.de/bizer/ng4j/disco/
[17] https://mes.github.io/marbles/

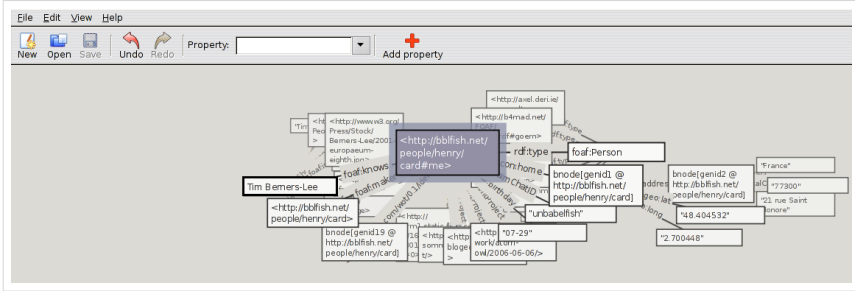


Abbildung 1.3: Ansicht des Fenfire-Browsers (vgl. Hastrup et al., 2008). Wie schon in Abbildung 1.2 werden Daten zu Tim Berners-Lee und verknüpften Ressourcen angezeigt. Ressourcen werden als Knoten, Verknüpfungen als beschriftete Kanten dargestellt. Schon bei der geringen Anzahl von Knoten und Kanten gibt es Überlappungen, sodass nicht alle Informationen direkt sichtbar sind.

Beispiel von IsaViz[18], PGV (vgl. Deligiannidis et al., 2007) und Fenfire (Hastrup et al., 2008) umgesetzt wird. Subjekte und Objekte der RDF-Tripel werden als Knoten, Prädikate als Kanten angezeigt. Die graphbasierte Ansicht ermöglicht zwar einen umfassenden Überblick über die Struktur der RDF-Dokumente und lässt den Betrachter zum Beispiel Cluster schnell erkennen, ist jedoch nicht für die ungefilterte Ansicht aller enthaltenen Informationen geeignet, da sich Elemente schnell überlagern (vgl. Abbildung 1.3).

Andere Anwendungen wie zum Beispiel gFacet (vgl. Heim et al., 2010) aggregieren daher Ressourcen gleicher Art innerhalb eines Knotens. Die enthaltenen Ressourcen können selektiert werden, sodass über eine facettierte Suche in den verknüpften Knoten nur die Ressourcen beziehungsweise Werte angezeigt werden, die zu der ausgewählten Ressource gehören. Ohne Selektion einer bestimmten Ressource ist ansonsten nicht erkennbar, welche Ressourcen direkt mit einander verknüpft sind, da diese Information durch die Aggregation verloren geht.

RelFinder (vgl. Heim et al., 2009) zeigt Suchergebnisse ebenfalls in Graphform an (vgl. Abbildung 1.4). Um Überlagerungen zu vermeiden, stoßen sich Knoten gegenseitig ab. Allerdings ist RelFinder nicht für die generische Anzeige von Daten geeignet, sondern zeigt nur Verbindungen zwischen nutzerdefinierten Knoten an. Knoten des Graphen können selektiert werden. Gleichzeitig wird der Pfad farblich hervorgehoben, auf dem der selektierte Knoten liegt. Dabei werden auch weitere Informationen wie ein Bild und ein Beschreibungstext zu dieser Ressource nachgeladen und angezeigt.

Weitere Browser wurden speziell für die Anzeige von Geokoordinaten und zeitbezogenen Daten entwickelt. Anwendungen wie Sextant (vgl. Nikolaou et al., 2015) oder SPEX (vgl. Scheider et al., 2017) zeigen die RDF-Daten mithilfe von Karten und Zeitstrahl-basierten Slidern an. Sextant kombiniert eine Karte und einen Zeitstrahl, um den örtlichen und zeitlichen Verlauf von Ereignissen darzustellen (vgl. Abbildung 1.5). Auf dem Zeitstrahl lässt sich ein Zeitpunkt oder eine Zeitspanne selektieren. Anhand dieser Selektion werden die anzuzeigenden Geo-Daten auf der Karte ein- oder ausgeblendet. Auf dem Zeitstrahl wird bereits angezeigt, ob zu einem Zeitpunkt Daten in der Karte dargestellt werden würden.

Gemein ist aber all diesen Browsern, dass weder die Endnutzer, noch die Bereitsteller der Daten oder Webseiten, auf denen diese angezeigt werden sollen, einen Einfluss auf die Art der Anzeige der Daten nehmen können. Wie die letztendliche Darstellung der Daten geschieht, wurde durch die Entwickler der anzeigenden Anwendungen vorbestimmt.

[18] https://www.w3.org/2001/11/IsaViz/

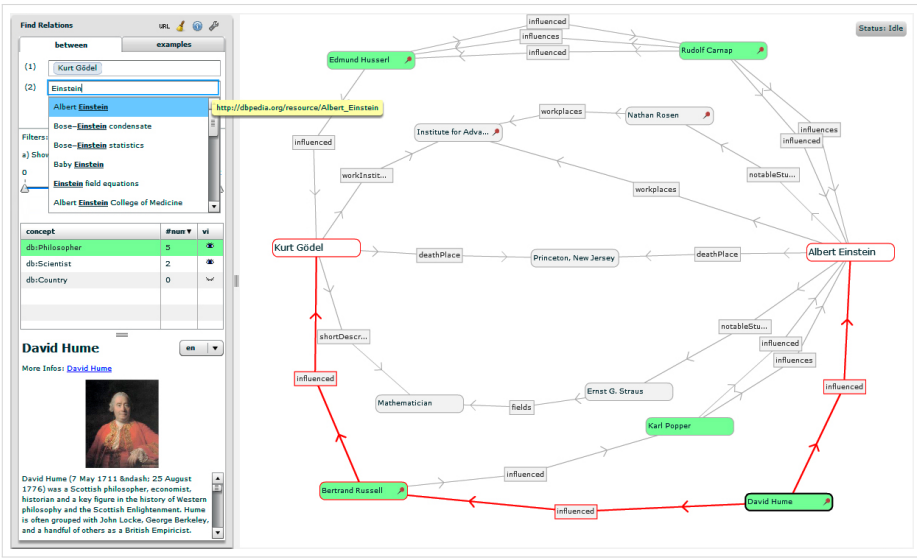


Abbildung 1.4: Ansicht des RelFinder (vgl. Heim et al., 2009). Mittels RelFinder lassen sich Verbindungen innerhalb eines RDF-Graphen über mehrere Knoten hinweg zwischen zwei vorgegebenen Ressourcen suchen und anzeigen. Die Darstellung der Suchergebnisse erfolgt in einer Graph-Ansicht (rechts). Für selektierte Knoten wird der Pfad zwischen den beiden Ausgangsknoten farblich hervorgehoben. Für den selektierten Knoten werden zudem weitere Informationen (unten links) nachgeladen und angezeigt.

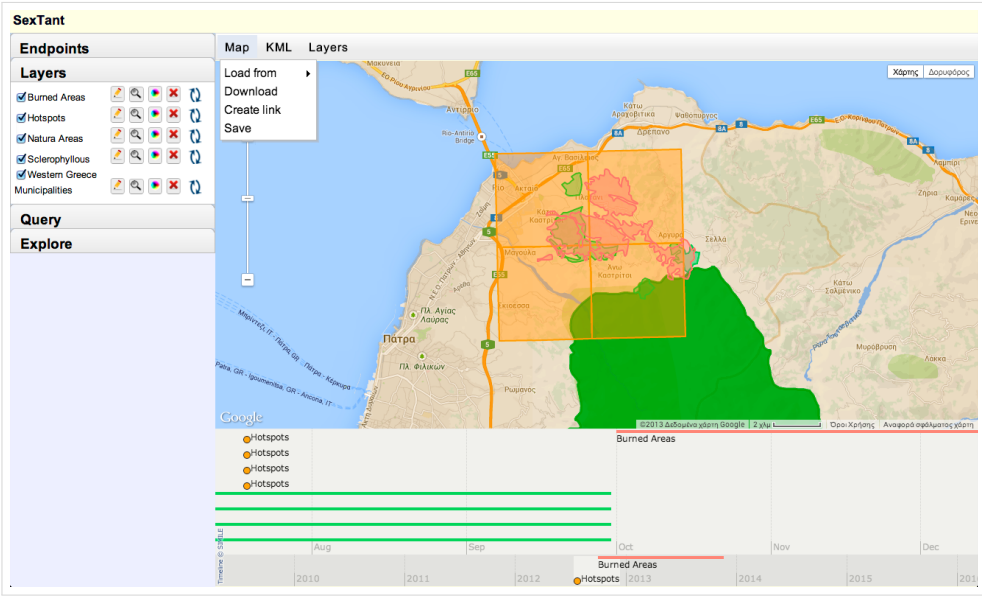


Abbildung 1.5: Ansicht des Sextant-Systems (vgl. Nikolaou et al., 2015). Sextant kombiniert eine Karte (oben) und einen Zeitstrahl (unten), um den örtlichen und zeitlichen Verlauf von Ereignissen darzustellen. Dazu lassen sich mehrere Ebenen auf einer Karte übereinanderlegen. Auf Basis des selektierten Zeitraums auf dem Zeitstrahl werden Punkte und Flächen auf der Karte entsprechend ein- oder ausgeblendet. Im Beispiel wird die Ausbreitung von Waldbränden in Griechenland dargestellt.

Sollen Daten auf eine bestimmte Weise in eine Webseite eingebunden werden, kann auf Systeme wie LESS (vgl. Auer et al., 2010), X3S (vgl. Stegemann et al., 2011), Sgvizler (vgl. Skjæveland, 2012) oder SPARQL Web Pages (SWP)[19] zurückgegriffen werden (die ersten beiden werden in Abschnitt 4.1.1, die letzten beiden in Abschnitt 3.1.3 genauer betrachtet). Hier können Webseitenautoren mittels Widgets oder Templates bestimmen, auf welche Weise, welche Daten angezeigt werden sollen. Die Betrachter der Webseite haben allerdings keinen Einfluss mehr auf die angezeigten Daten. Eine Exploration der Daten ist nur noch möglich, wenn die Webseitenautoren diese auch vorsehen.

1.3.3 Abfrage

Die Weise, wie die anzuzeigenden Daten ausgewählt werden können, unterscheidet sich in den bisher beschriebenen Anwendungen und Systemen teilweise stark voneinander. Während den generischen RDF-Browsern schon die Angabe einer URL zu einem RDF-Dokument reicht, benötigen die anderen Systeme (teilweise mehrere) komplexe SPARQL-Abfragen, um die Daten auf die gewünschte Weise anzeigen zu können. So benötigt Sextant für jede anzuzeigende Ebene eine entsprechende Abfrage. Sollen mittels LESS komplexe Darstellungen realisiert werden, müssen alle Daten über nur eine Abfrage abgerufen werden. Solche Abfragen lassen sich nur von Experten formulieren.

Um auch Anfänger im Linked-Data-Bereich ansprechen zu können, benötigt RelFinder als Eingabe nur einfache Suchbegriffe, anhand derer Ressourcen gesucht und ausgewählt werden können. Für die Nutzung von gFacet wird eine RDF-Klasse benötigt, zu der die anzuzeigenden Ressourcen gehören, welche ebenfalls durch eine einfache Textsuche gesucht und ausgewählt werden kann. Beide Anwendungen erzeugen intern teils sehr komplexe SPARQL-Abfragen, die jedoch vor den Nutzern verborgen bleiben.

Andere Browser und Systeme unterstützen die Nutzer auch bei der individuellen Erzeugung der Abfragen. Zur vereinfachten Erstellung von SPARQL-Abfragen, lassen sich mit Tabulator diese nach dem Query-by-Example-Prinzip (vgl. Zloof, 1975) konstruieren und bei Bedarf anschließend nachbearbeiten. Um eine Abfrage initial zu konstruieren, müssen durch Tabulator bereits Daten angezeigt werden. In diesen Daten können die Nutzer eine oder mehrere gewünschte Eigenschaften (Properties) auswählen, die sie in ihren Abfragen enthalten haben möchten. Aus den ausgewählten Eigenschaften wird eine SPARQL-Abfrage mit dem entsprechenden Triple-Pattern abgeleitet.

SPEX unterstützt die Nutzer bei der Erstellung von Abfragen durch eine graphbasierte Eingabe (vgl. Abbildung 1.6), ähnlich zu QueryVOWL (vgl. Haag et al., 2015). Nutzer starten mit einem einzelnen leeren Knoten, der als Variable innerhalb des Triple-Patterns dient und erweitern ausgehend von diesem Knoten ihre Abfrage. Knoten mit Zeit- oder Geo-Daten werden automatisch mit dem Zeitstrahl oder der Karte verknüpft. Über den Zeitstrahl und die Karte können die Ergebnisse innerhalb der Tabelle gefiltert werden.

Der X3S-Editor (vgl. Stegemann et al., 2011; genauer betrachtet in Abschnitt 4.1.1) arbeitet ebenfalls nach dem Query-by-Example-Prinzip. Nutzer müssen eine RDF-Klasse auswählen und können anschließend Eigenschaften, welche zu dieser gehören in einen hierarchisch aufgebauten Abfragebereich ziehen. Je nachdem, ob die Eigenschaft Literale oder weitere Ressourcen enthält, können im zweiten Fall verschachtelt weitere Eigenschaften in den Arbeitsbereich gezogen werden.

[19] https://uispin.org

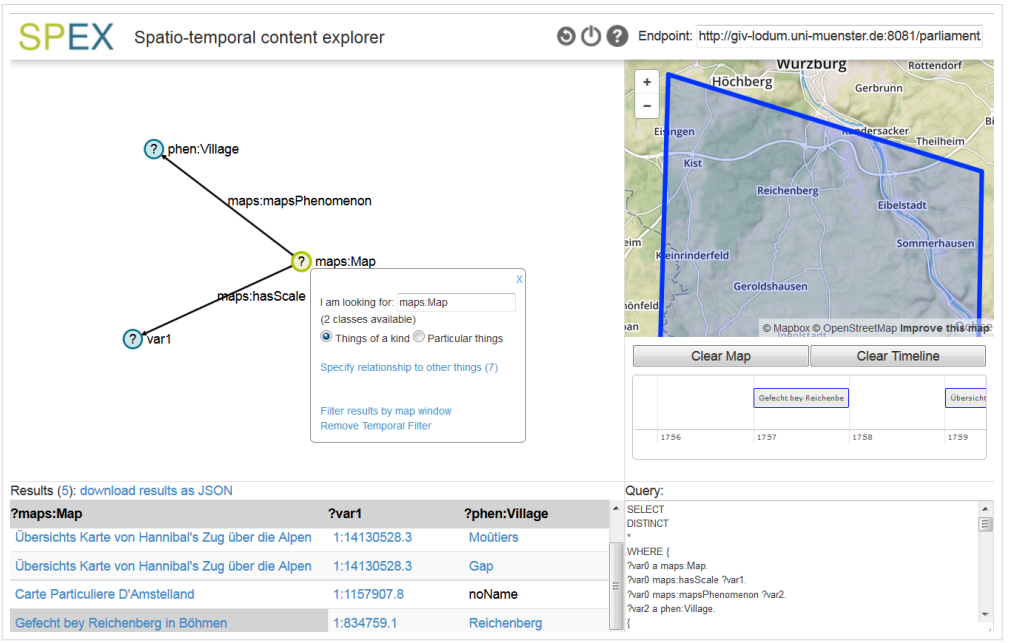


Abbildung 1.6: Ansicht des SPEX-Browsers (vgl. Scheider et al., 2017). In einer graphbasierten Ansicht (oben links) wird die SPARQL-Abfrage konstruiert. Die Ergebnisse der Abfrage werden mit der Karte und dem Zeitstrahl (oben rechts) verknüpft. Innerhalb der Karte und des Zeitstrahls können die Ergebnisse gefiltert werden, was sich direkt auf die Tabellenansicht (unten links) auswirkt.

1.3.4 Zusammenfassung und Fazit

Mit den bereits vorhandenen Anwendungen und Systemen ist es bereits für Nutzer ohne großes Hintergrundwissen bezüglich Linked-Data möglich, diese Daten abzufragen und zu explorieren. Je nach Zielsetzung und Datenart müssen die Nutzer jedoch einen unterschiedlich hohen Aufwand betreiben, um ihr Ziel zu erreichen. Bei stark auf einen Anwendungszweck hin optimierten Anwendungen wie RelFinder oder gFacet müssen die Nutzer nur wenige Ressourcen oder RDF-Klassen mittels einer Textsuche auswählen, um ein Ergebnis zu erhalten. Danach können sie die Ergebnismenge durch Filter einschränken oder im Fall von gFacet auch durch die Auswahl weiterer Eigenschaften erweitern.

Für die Nutzung eines RDF-Browsers müssen die Nutzer zumindest die URL eines RDF-Dokuments kennen und eingeben. Ab diesem Zeitpunkt sind für die reine Daten-Exploration jedoch keine weitgehenden Linked-Data-Kenntnisse mehr nötig. Durch klicken auf weitere Ressourcen werden zusätzliche Daten nachgeladen und angezeigt. Erst wenn komplexere Abfragen erstellt werden sollen, sind SPARQL-Kenntnisse hilfreich.

Der X3S-Editor kann ebenfalls ohne Linked-Data-Kenntnisse genutzt werden. Nutzer müssen jedoch die RDF-Klasse kennen und eingeben, für die sie Daten abrufen möchten. Mit dem X3S-Editor lassen sich nur hierarchisch strukturierte Abfragen erzeugen, wodurch die Konstruktion von komplexen Abfragen ausgeschlossen ist.

Bei der Nutzung der Systeme für Zeit- und Geo-Daten sind tiefergreifende Linked-Data-Kenntnisse jedoch unabdingbar. Zwar lassen sich mittels SPEX auch Abfragen über einen graphbasierten Query-Editor erstellen, doch müssen Nutzer den Aufbau und die Funktionsweise

von RDF-Graphen verinnerlicht haben, um Abfragen zu konstruieren, die auch wirklich die gewünschten Daten abrufen. Für die Nutzung von Sextant sind SPARQL-Kenntnisse unbedingt notwendig. Diese beiden Systeme können somit als Experten-Systeme betrachtet werden, die nur von einer sehr kleinen Nutzergruppe eingesetzt werden können.

Bei all diesen Anwendungen ist die Art der Datendarstellung bereits vorgegeben. Die Nutzer können nicht bestimmen, ob Daten in einer Tabelle, als Graph oder in einem Diagramm dargestellt werden sollen. Der X3S-Editor erlaubt zwar in geringem Maße die Darstellung durch CSS-Regeln anzupassen, einen Einfluss auf die Darstellungsart können die Nutzer jedoch nicht nehmen, sondern nur die vorgegebene Darstellungsart anpassen.

Für die Systeme, die Linked-Data in Webseiten einbinden, sind SPARQL-Kenntnisse ebenfalls verpflichtend. Allerdings wird mit den Webseitenautoren auch eine andere Zielgruppe angesprochen. Bei dieser Gruppe kann von einem deutlich höheren informatischen Vorwissen ausgegangen werden, als bei den Nutzern der RDF-Browser. Ebenso kann hier von einer höheren Bereitschaft ausgegangen werden, sich in die benötigten Techniken einzuarbeiten. Dennoch würde auch diese Nutzergruppe von stärker unterstützenden Anwendungen und Systemen oder allgemein einfacheren Techniken profitieren.

1.4 Aufbau der Arbeit

Diese Arbeit ist in 5 Kapitel unterteilt. In Kapitel 1 wird die Motivation erläutert, die dieser Arbeit zugrunde liegt. Die Forschungsfragen, die sich aus der Motivation ergeben, werden anschließend vorgestellt. Ebenso enthält das Kapitel eine einleitende Beschreibung des Semwidg-Projekts, in dessen Rahmen jene Forschungsfragen beantwortet werden sollen. Es folgt eine Einordnung der potenziellen Nutzer des Semwidg-Projekts in verschieden Nutzergruppen, sowie eine Identifikation der Hauptzielgruppe. Gleichfalls wird ein einleitender allgemeiner Überblick über den aktuellen Forschungsstand bezüglich der Abfrage, Exploration und Darstellung von Daten aus dem Linked-Data-Bereich gegeben.

Den drei Teilprojekten des Semwidg-Projekts – SemwidgQL, SemwidgJS und SemwidgED – ist jeweils ein Kapitel gewidmet. Dementsprechend wird in Kapitel 2 die Abfragesprache SemwidgQL beschrieben, beginnend mit einer Analyse des aktuellen Forschungsstands (vgl. Abschnitt 2.1). Es folgen konzeptionelle Überlegungen hinsichtlich der Analyse des Forschungsstands sowie der Anforderungen potenzieller Nutzer und den Zielen, die sie mit SemwidgQL verfolgen würden (vgl. Abschnitt 2.2). Aus diesen Überlegungen ergibt sich anschließend die konkrete Definition von SemwidgQL (vgl. Abschnitt 2.3), für die zwei Referenzimplementierungen, eine in JavaScript und eine Java, entwickelt wurden. In der nachfolgenden empirischen Nutzerstudie wurde SemwidgQL mit SPARQL, der Standardsprache zur Abfrage von RDF-Daten, verglichen (vgl. Abschnitt 2.4). Untersuchungspunkte waren die objektiven Kriterien Effektivität, Effizienz und Erlernbarkeit der beiden Sprachen, sowie die subjektive Zufriedenheit der Nutzer bei der Verwendung beider Sprachen. In einer weiteren Evaluation wurde die Ausdrucksfähigkeit von SemwidgQL anhand des Linked-SPARQL-Query-Datensatzes (LSQ) untersucht, der mehrere hunderttausend Abfragen enthält, die an einige bedeutende Endpoints der LOD-Cloud gestellt und aus den jeweiligen Logdaten extrahiert wurden (vgl. Abschnitt 2.4). Abschließend werden die gewonnenen Erkenntnisse diskutiert und ein Fazit hinsichtlich der erzielten Ergebnisse zu SemwidgQL gezogen (vgl. Abschnitt 2.6).

In Kapitel 3 wird die JavaScript-Bibliothek SemwidgJS zur Präsentation und Visualisierung von Daten aus öffentlichen SPARQL-Endpoints vorgestellt. Nach einer Analyse des aktuellen Forschungsstands (vgl. Abschnitt 3.1) werden die konzeptionellen Überlegungen zu SemwidgJS beschrieben, die sich aus ebenjener Analyse sowie den Zielen und Fähigkeiten der potenziellen Nutzern ergeben (vgl. Abschnitt 3.2). Darauf aufbauend wird die konkrete Implementierung von SemwidgJS hinsichtlich relevanter Funktionen und Elementen beschrieben, sowie die Möglichkeit zur Erzeugung benutzerdefinierter Widget-Klassen und Erstellung interaktiver Webseiten (vgl. Abschnitt 3.3). Zur Untermauerung der Nützlichkeit von SemwidgJS, wird der Einsatz dieser Bibliothek im Rahmen des Eurostars-geförderten Projekts DESUMA dargestellt (vgl. Abschnitt 3.4). Das Kapitel endet mit einer Diskussion der erzielten Ergebnisse zu SemwidgJS (vgl. Abschnitt 3.5).

Kapitel 4 widmet sich dem Online-WYSIWYG-Editor SemwidgED, welcher seine Anwender bei der Nutzung von SemwidgJS und Formulierung von SemwidgQL-Abfragen unterstützt. Wie die beiden vorhergehenden Kapitel beginnt auch dieses mit einer Analyse des aktuellen Forschungsstands (vgl. Abschnitt 4.1). Dabei werden sowohl Editoren für Linked-Data-Präsentationen wie auch Editoren zur Erstellung von SPARQL-Abfragen betrachtet. Aufbauend auf einer Ermittlung der Anforderungen, welche sich wiederum aus den Zielen und Fähigkeiten der potenziellen Nutzern ergeben (vgl. Abschnitt 4.2), wird die Implementierung von SemwidgED beschrieben (vgl. Abschnitt 4.3). In der nachfolgenden empirischen Nutzerstudie wurde die Gebrauchstauglichkeit von SemwidgED evaluiert (vgl. Abschnitt 4.4). Untersucht wurde Effizienz

und Effektivität des Umgangs der Probanden mit SemwidgED, sowie deren subjektive Zufriedenheit bei diesem. Abschließend werden die gewonnenen Erkenntnisse wiederum diskutiert und ein Fazit hinsichtlich der erzielten Ergebnisse zu SemwidgED gezogen (vgl. Abschnitt 4.5).

Abschließend erfolgt in Kapitel 5 eine Zusammenfassung der gesamten Arbeit (vgl. Abschnitt 5.1), gefolgt von einer Diskussion der Potenziale wie auch Limitationen des Semwidg-Projekts (vgl. Abschnitt 5.2). Ein Ausblick über etwaige weiterführende Forschungsfragen und potenzielle Weiterentwicklungen innerhalb des Semwidg-Projekts schließen diese Arbeit ab (vgl. Abschnitt 5.3).

1.5 Im Zusammenhang mit der Dissertation entstandene Publikationen

Diese Dissertation basiert zum Teil auf folgenden Publikationen, welche im Rahmen verschiedener Konferenzen vorgestellt wurden. Alle Publikationen haben zuvor einen unabhängigen Peer-Review-Prozess durchlaufen.

- Stegemann, T. & Ziegler, J. (2014). *SemwidgJS: A Semantic Widget Library for the Rapid Development of User Interfaces for Linked Open Data.* In: Plödereder, E., Grunske, L., Schneider, E. & Ull, D. (Hrsg.), *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, Bd. 232 d. Reihe *LNI*, S. 479–490. GI.

- Stegemann, T. & Ziegler, J. (2017a). *Investigating Learnability, User Performance, and Preferences of the Path Query Language SemwidgQL Compared to SPARQL.* In: d'Amato, C., Fernandez, M., Tamma, V., Lecue, F., Cudré-Mauroux, P., Sequeda, J., Lange, C. & Heflin, J. (Hrsg.), *The Semantic Web – ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21–25, 2017, Proceedings, Part I*, S. 611–627, Cham. Springer International Publishing.

- Stegemann, T. & Ziegler, J. (2017b). *Pattern-Based Analysis of SPARQL Queries from the LSQ Dataset*. In: Nikitina, N., Song, D., Fokoue, A. & Haase, P. (Hrsg.), *ISWC 2017 Posters & Demonstrations and Industry Tracks (ISWC-PD-Industry)*, Nr. 1963 in *CEUR Workshop Proceedings*, Aachen.

Im weiteren thematischen Kontext dieser Dissertation sind zudem folgende Publikationen entstanden, welche sich ebenfalls mit der Präsentation und Visualisierung von Linked Open Data befassen und auf welche in dieser Dissertation verwiesen wird.

- Heim, P., Hellmann, S., Lehmann, J., Lohmann, S. & Stegemann, T. (2009). *RelFinder: Revealing Relationships in RDF Knowledge Bases*. In: *Proceedings of the 4th International Conference on Semantic and Digital Media Technologies (SAMT 2009)*, S. 182–187, Berlin/Heidelberg. Springer.

- Stegemann, T., Hussein, T., Gaulke, W. & Ziegler, J. (2011). *Interacting with semantic data by using X3S*. In: Diaz, P., Hussein, T., Lohmann, S. & Ziegler, J. (Hrsg.), *Proceedings of the 1st Workshop on Data-Centric Interactions on the Web*.

- Stegemann, T., Ziegler, J., Hussein, T. & Gaulke, W. (2012). *Interactive Construction of Semantic Widgets for Visualizing Semantic Web Data*. In: *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, S. 157–162, New York, NY, USA. ACM.

- Ziegler, J. & Stegemann, T. (2014). *The Role of Semantic Data in Engineering Interactive Systems*. HCI Engineering Charting the Way towards Methods and Tools for Advanced Interactive Systems, S. 43.

KAPITEL 2

SemwidgQL – Eine Abfragesprache für Linked Open Data

Eine Vielzahl von SPARQL-Endpoints werden der Öffentlichkeit von verschiedenen Universitäten, Bibliotheken, Museen, Behörden, Firmen, Gruppen und Einzelpersonen online zur Verfügung gestellt und enthalten umfangreichen Datenmengen im RDF-Format zu verschiedenen Themengebieten. Dennoch ist die Anzahl von Anwendungen und Webseiten, die von diesen Daten Gebrauch machen, im öffentlich zugänglichen Internet sehr gering. Ein Hauptproblem dürfte dabei sein, dass SPARQL, die Standardsprache für RDF-Daten im Linked-Data-Bereich, durch ihre Mächtigkeit und ihren großen Funktionsumfang, einen entsprechend hohen Einarbeitungsaufwand erfordert, der auf viele potenzielle Nutzer abschreckend wirkt.

Zwar wurden in den letzten Jahren einige Alternativen zu SPARQL entwickelt, doch benötigten diese auch immer einen direkten Zugriff auf die Rohdaten, die sie verarbeiten sollen. Auch wenn diese Sprachen vielleicht einfacher zu erlernen waren, entstand so an anderer Stelle für deren Nutzer ein noch viel größeres Problem, nämlich dass sie die Daten und die verarbeitenden Programme jetzt selbst auf einem Server vorhalten mussten, statt einfach auf einen verfügbaren SPARQL-Endpoint zugreifen zu können.

Um diese Probleme zu lösen, wurde die Abfragesprache SemwidgQL entwickelt, welche in diesem Kapitel vorgestellt wird. SemwidgQL ist eine leicht zu erlernende Pfadabfragesprache, die sich auf Nutzerseite zu SPARQL transkompilieren lässt und sich in ihrer Syntax und Semantik an den Vorkenntnissen von Programmierern und Webentwicklern hinsichtlich ihnen bekannter Techniken orientiert. Dadurch dass SemwidgQL in SPARQL übersetzt wird, kann die Abfragesprache mit nahezu allen SPARQL-Endpoints genutzt werden. Die Sprache wurde mit dem Ziel entwickelt, dass neue Nutzer, wie zum Beispiel Webentwickler, die die zuvor bereits erwähnten umfangreichen Datenmengen der öffentlichen SPARQL-Endpoints erschließen und für sich und weitere Personen nutzbar machen können. Letztendlich soll mit SemwidgQL eine Antwort auf die zuvor aufgestellte erste Forschungsfrage (»Wie lassen sich Daten aus der LOD-Cloud auch von Programmierern und Webentwicklern abrufen, die keine Linked-Data-Experten sind?«) gefunden werden.

Der Quelltext von SemwidgQL wurde in JavaScript und in Java, zusammen mit einer ausführlichen Beschreibung der Sprache und dessen Syntax, auf der Semwidg-Projektseite[1] veröf-

[1] https://semwidg.org/page/semwidgql

fentlicht. Zusätzlich wird dort beschrieben, wie sich eine SemwidgQL-Abfrage in eine SPARQL-Abfrage übersetzen lässt, so dass Entwickler bei Bedarf SemwidgQL auch in anderen Programmiersprachen verfügbar machen können.

Zu Beginn dieses Kapitels wird der aktuelle Stand der Forschung im Bezug auf Abfragesprachen für RDF-Daten beschrieben. Dabei werden in erster Linie Pattern-basierte Abfragesprachen und Pfadabfragesprachen betrachtet (vgl. Abschnitt 2.1). Anschließend wird das Konzept für die eigene Abfragesprache SemwidgQL vorgestellt, die es auch Nutzern, die nur bedingt mit den Konzepten und Techniken des Linked-Data-Bereichs vertraut sind, ermöglichen soll, Daten von öffentlichen SPARQL-Endpoints abzurufen (vgl. Abschnitt 2.2). Dazu wurden anhand der potenziellen Nutzer und dem bisherigen Stand der Forschung verschiedene Anforderungen ermittelt. Aufbauend auf diesem Konzept wurde die Abfragesprache entwickelt, deren Beschreibung nachfolgend vorgestellt wird (vgl. Abschnitt 2.3). Anschließend werden zwei Studien zu SemwidgQL präsentiert. Die erste ist eine empirische Nutzerstudie, in der verglichen wurde, wie effizient und effektiv Nutzer SemwidgQL und SPARQL nutzen konnten. Ebenfalls wurde untersucht, welche Sprache leichter zu erlernen war (vgl. Abschnitt 2.4). In der zweiten Studie wurde die Ausdrucksfähigkeit von SemwidgQL untersucht. Auf Basis von Log-Daten großer öffentlicher SPARQL-Endpoints wurde überprüft, wie groß der Anteil der Abfragen ist, der sich auch mit SemwidgQL statt mit SPARQL hätte realisieren lassen (vgl. Abschnitt 2.5). Abschließend werden die Ergebnisse dieses Kapitels diskutiert und ein Fazit gezogen (vgl. Abschnitt 2.6).

2.1 Stand der Forschung zu RDF-Abfragesprachen

Auch wenn sich das Semwidg-Projekt auf die Nutzung von SPARQL-Endpoints zur Datenabfrage beschränkt, soll im nachfolgenden Abschnitt zusätzlich ein Überblick über weitere gängige Abfragesprachen für RDF-Daten gegeben werden, die sich jedoch nicht in Kombination mit solchen Endpoints nutzen lassen. Diese Abfragesprachen unterscheiden sich nicht nur in ihrem Funktionsumfang und ihrer Mächtigkeit, sondern auch hinsichtlich ihre genutzten Abfrageparadigmen.

Die aus der Klassifikation von Programmiersprachen stammende Unterscheidung in deklarative sowie imperative Sprachen wird häufig herangezogen, um auch Abfragesprachen einzuordnen (vgl. Olston et al., 2008). Für Daten im RDF-Format werden in der Regel jedoch andere Unterscheidungskriterien bemüht. Ghanem (2017) unterscheidet zwischen prozeduralen, deklarativen und Scripting-Sprachen. Furche et al. (2006) unterscheiden wiederum zwischen relationalen beziehungsweise Pattern-basierten Abfragesprachen (z. B. SPARQL), »reactive rules«-Abfragesprachen (z. B. Algae[2]) sowie »navigational access«-Abfragesprachen (z. B. Versa[3]), lassen jedoch auch Raum für Abfragesprachen, die sich nicht auf diese Weise klassifizieren lassen. Weitere Klassen von Abfragesprachen, die in der Forschung eine hohe Bedeutung haben, sind natürlichsprachliche (z. B. AquaLog, vgl. Lopez & Motta, 2004), grafische (z. B. QueryVOWL, vgl. Haag et al., 2015) und funktionale Abfragesprachen (z. B. Gremlin, vgl. Rodriguez, 2015) sowie Abfragesprachen nach dem Query-by-Example-Prinzip (vgl. Zloof, 1975) wie zum Beispiel SPARQLBYE (vgl. Diaz et al., 2016) und RDF-QBE (vgl. Reynolds, 2002).

Eine eindeutige Einordnung der Abfragesprachen aus dem Linked-Data-Bereich ist allerdings nur selten möglich, da eine Sprache teilweise mehreren dieser Klassen zugeordnet werden kann. So ist SPARQL beispielsweise eine deklarative, Pattern-basierte Abfragesprache, die, durch die mit Version 1.1 eingeführten *Property Paths*[4], auch als Pfadabfragesprache klassifiziert werden kann. Bailey et al. (2009) nahmen daher auch keine explizite Klassifikation für Abfragesprachen aus dem Linked-Data-Bereich mehr vor, sondern definieren eine Reihe von Eigenschaften, in denen diese Sprachen sich voneinander unterscheiden können. Eine der wichtigsten Eigenschaften ist das genutzte Abfrageparadigma, welches hier entweder pfadbasiert (bzw. navigational) oder logikbasiert (bzw. positional oder Pattern-basiert) ist. Eine Einordnung von Abfragesprachen auf Grundlage dieser Eigenschaften wird auch von Angles et al. (2017) vorgenommen.

In Anlehnung an diese Einordnung, wird nachfolgend ein Überblick über relevante Pattern-basierte Abfragesprachen und Pfadabfragesprachen für Daten aus dem Linked-Data-Bereich gegeben. Zuvor wird jeweils ein Einführung in das entsprechende Abfrageparadigma geboten. Abschließend werden die Erkenntnisse, die aus diesem Überblick gewonnen wurden, zusammengefasst.

2.1.1 Pattern-basierte Abfragesprachen

Pattern-basierte Abfragesprachen fragen Daten in RDF-Datenbanken mittels Graph-Pattern-Matching ab. Nach Angles et al. (2017) lassen sich die dazu benötigten Graph-Patterns in einfache und komplexe Graph-Patterns unterscheiden. Einfache Graph-Patterns (*Basic Graph Patterns*, BGPs) entsprechen in ihrer Struktur dem Graph, auf den die Abfrage angewandt werden

[2] https://www.w3.org/2004/05/06-Algae/
[3] https://github.com/uogbuji/uche.ogbuji.net/tree/master/files/etc/tech/rdf/versa
[4] https://www.w3.org/TR/sparql11-property-paths/

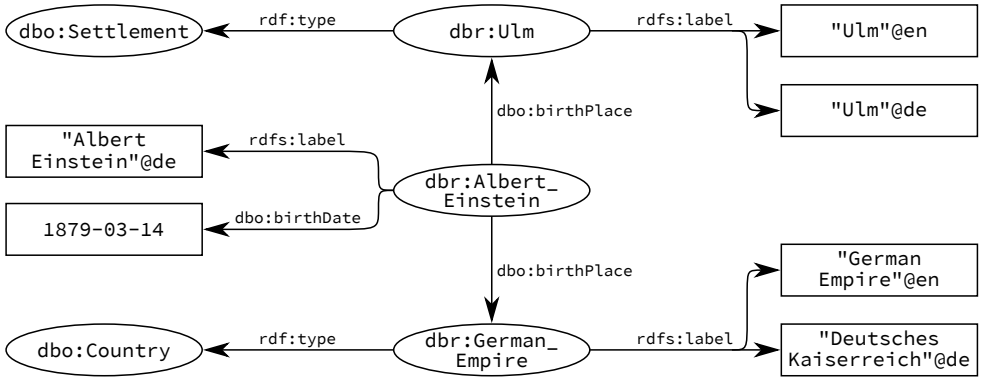
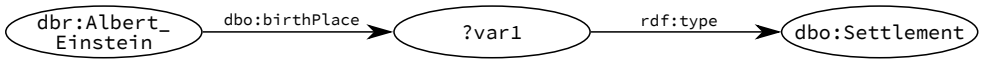
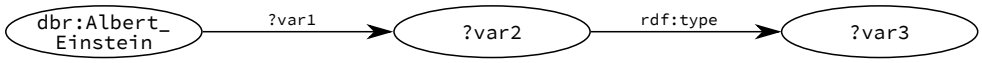


Abbildung 2.1: Ein einfacher Beispielgraph, basierend auf Daten des DBpedia-Datensatzes, der Informationen zu Albert Einstein und seinem Geburtsdatum und -ort enthält. Innerhalb des Graphen sind zwei Geburtsorte für Albert Einstein mit unterschiedlicher geographischer Schärfe gespeichert. Zum einen die Geburtsstadt und zum anderen das Geburtsland. In beiden Fällen wird jedoch dieselbe Bezeichnung für diese Eigenschaft verwendet.



(a) BGP mit einer Variablen.



(b) BGP mit drei Variablen.

Abbildung 2.2: Zwei Basic Graph Patterns (BGPs), die auf den Beispielgraph aus Abbildung 2.1 angewandt werden können. Beide BGPs umfassen jeweils zwei Subjekt-Prädikat-Objekt-Relationen, weisen jedoch eine unterschiedliche Anzahl an Variablen auf.

soll, jedoch mit dem Unterschied, dass mindestens ein Teil des BGPs durch eine Variable repräsentiert wird. Die Datenbasis wird mit diesem BGP abgeglichen. Entspricht ein Teil der Datenbasis der Struktur des BGPs, werden die entsprechenden konstanten Werte der Datenbasis auf die Variablen des BGPs abgebildet und als Ergebnis zurückgeliefert.

Abbildung 2.2 enthält zwei BGPs mittels derer ein Graph-Pattern-Matching mit dem Beispielgraph aus Abbildung 2.1 durchgeführt werden kann, welcher auf Daten des DBpedia-Datensatzes basiert. Wird der Graph mit dem ersten BGP (vgl. Abbildung 2.2a) abgeglichen, wird die Resource `dbr:Ulm` auf die Variable `?var1` abgebildet. Weitere Möglichkeiten für eine erfolgreiche Abbildung sind in diesem Fall nicht möglich. Im zweiten Beispiel (vgl. Abbildung 2.2b) werden jedoch zwei verschiedene Abbildungskombinationen zurückgeliefert. In beiden Fällen wird die Variable `?var1` mit `dbo:birthPlace` belegt. Die Variablen `?var2` und `?var3` werden im ersten Fall mit den Werten `dbr:Ulm` beziehungsweise `dbo:Settlement` belegt. Im zweiten Fall werden sie hingegen mit den Werten `dbr:German_Empire` beziehungsweise `dbo:Country` belegt.

Komplexe Graph-Patterns (*Complex Graph Patterns*, CGPs) fügen den BGPs weitere Operationen hinzu. Unter anderem lassen sich mittels *Projektion* die zurückgelieferten Variablen der Ergebnismengen festlegen. *Joins* und *Unions* ermöglichen die Kombination der Daten und *Filter* ermöglichen den Ausschluss bestimmter Daten anhand definierbarer Konditionen. Nachfolgend wird eine Auswahl relevanter Pattern-basierter Abfragesprachen vorgestellt. Sie alle nutzen CGPs zur Beschreibung der abzufragenden Daten, wobei sich die genutzten Syntaxen jedoch teilweise stark von einander unterscheiden.

Ein früher Vertreter der Pattern-basierten Abfragesprachen für Daten im RDF-Format ist RQL (vgl. Karvounarakis et al., 2002). RQL erlaubt sowohl die Abfrage auf Daten- wie auch auf Schema-Ebene. So ermöglichen unter anderem die Funktionen `sub-` und `superClassOf` sowie `sub-` und `superPropertyOf`, welche ein fester Bestandteil der Sprache sind, die Inferenz auf RDF-Schema-Basis (RDFS). Allerdings erwartet die Abfragesprache auch, dass die abzufragenden Daten anhand von RDFS annotiert wurden, was einen höheren Aufwand bei der Erstellung und Pflege der Daten erfordert. Weiterhin enthält die Sprache die Möglichkeit zur Formulierung von generalisierten Pfadausdrücke zur Navigation innerhalb des Graphen beziehungsweise zur Filterung der Daten. Dazu wird die Punkt-Notation genutzt, wie sie in vielen objektorientierten Programmiersprachen und auch in SemwidgQL (vgl. Abschnitt 2.3.2) zum Einsatz kommt. RQL unterstützt eine Vielzahl weiterer Funktionen, um unter anderem Klassen und Datentypen abzufragen. Abfragen auf Schema-Ebene erfordern die Präfigierung der darin genutzten Variablen. So müssen den Variablen für RDFS-Klassen das Präfix »$« vorangestellt werden, Variablen für Prädikate hingegen das Präfix »@«. URIs müssen grundsätzlich mit dem Präfix »&« versehen werden. RQL wurde aufgrund dieser syntaktischen Konstrukte und der hohen Anzahl von Funktionen häufig kritisiert (z. B. Furche et al., 2006). Infolgedessen wurden verschiedene Nachfolgersprachen entwickelt, die zwar nicht die gleiche Mächtigkeit wie RQL aufweisen, jedoch deutlich einfacher zu nutzen sind. Mit eRQL (vgl. Wleklinski, 2003) wurde eine Abfragesprache entwickelt, die in RQL übersetzt wird und damit die bereits bestehende Software-Infrastruktur weiternutzen kann. Eine weitere Nachfolgersprache ist SeRQL (vgl. Broekstra & Kampman, 2003), welche neben SPARQL eine der beiden Abfragesprachen ist, die im Java-Framework RDF4J[5] (vormals Sesame, vgl. Broekstra et al., 2002) unterstützt wird.

Ein weiterer Schwäche von RQL ist die starre Fixierung auf RDFS als einziges Datenschema. Daten, die andere Schemata oder Erweiterungen von RDFS, wie zum Beispiel DAML+OIL (vgl. Horrocks, 2002) und OWL[6], nutzen, lassen sich mittels RQL nur unzureichend abfragen, da die Inferenzregeln, die von RQL unterstützt werden, nicht an diese Schemata angepasst werden können. Die Abfragesprache TRIPLE (vgl. Sintek & Decker, 2002) unterstützt hingegen ein generisches Inferenzsystem, auf Basis von Horn-Klauseln (vgl. Horn, 1951), die in der Syntax von F-Logic (vgl. Kifer et al., 1995) formuliert werden. Damit lassen sich auch andere Datenschemata flexibel verarbeiten.

Weitere Pattern-basierte Abfragesprachen für Daten im RDF-Format, welche eine gewisse Relevanz im Linked-Data-Bereich vorweisen können, sind Xcerpt (vgl. Bry & Schaffert, 2002), LDQL (vgl. Hartig & Pérez, 2015) und Cypher (vgl. Francis et al., 2018). Letztere Abfragesprache kann auch eine breite Verwendung im kommerziellen Bereich aufweisen. Unter anderem ist Cypher in Neo4j und SAP HANA Graph implementiert.[7]

Die mit Abstand wichtigste Abfragesprache im Linked-Data-Bereich ist jedoch SPARQL, welche vom World Wide Web Consortium (W3C) als Abfragesprache für Daten im RDF-Format

[5] `https://rdf4j.org`
[6] `https://www.w3.org/TR/owl-overview/`
[7] `https://www.opencypher.org/projects`

Listing 2.1: Eine SPARQL-SELECT-Abfrage. Im WHERE-Teil der Abfrage werden die durch die Relation birthPlace mit Albert Einsteins verknüpften Orte und deren Labels mit dem Graphen aus Abbildung 2.1 abgeglichen. Die Labels werden anschließend durch einen Filterausdruck auf ausschließlich deutschsprachige Ergebnisse reduziert. Im SELECT-Teil wird aus der Ergebnismenge des WHERE-Teils die Variable, welche die Labels enthält, für die Rückgabe ausgewählt. Die Ergebnisse der Variable, welche die Resource-URIs der Geburtsorte enthält, werden nicht zurückgegeben.

```
1 SELECT ?label
2 WHERE {
3   dbr:Albert_Einstein dbo:birthPlace ?birthPlace .
4   ?birthPlace rdfs:label ?label .
5   FILTER (langMatches(lang(?label), "de"))
6 }
```

empfohlen wird[8]. Hervorgegangen ist SPARQL aus den Abfragesprachen SquishQL (vgl. Miller et al., 2002) und RDQL[9]. Letztere war für lange Zeit ein fester Bestandteil des Open-Source-Java-Frameworks (Apache[10]) Jena (vgl. McBride, 2002), wurde jedoch in den letzten Jahren vollständig durch SPARQL ersetzt. Die Beschreibung der Triple-Pattern in SPARQL erfolgt nach der Turtle-Notation[11] für RDF-Daten. SPARQL unterstützt vier Formen der Datenabfrage: SELECT, CONSTRUCT, ASK und DESCRIBE. Die meisten SPARQL-Abfragen werden jedoch als SELECT-Abfragen formuliert (vgl. Abschnitt 2.5.1).

SELECT-Abfragen bestehen mindestens aus einem SELECT- und einem WHERE-Teil. Der WHERE-Teil enthält eine beliebige Anzahl Graph-Patterns, welche mit dem Graphen abgeglichen werden. Die Ergebnisse dieses Abgleichs können über Filterausdrücke (FILTER) eingeschränkt werden. Der SELECT-Teil ermöglicht die Auswahl der zurückgelieferten Variablen durch Projektion. Listing 2.1 zeigt ein Beispiel für eine solche SELECT-Abfrage.

Weiterhin bietet SPARQL unter anderem die Möglich zur Kombination von Daten (UNION), Beschreibung von optional abzugleichenden BGPs (OPTIONAL, entspricht einem LEFT-OUTER-JOIN (vgl. Codd, 1979) in relationalen Datenbanken), Gruppierung (GROUP BY) und Aggregation von Daten sowie die Limitierung der Anzahl der zurückgegebenen Daten (LIMIT).

Seit der Version 1.1[12] von SPARQL, besteht zudem die Möglichkeit Pfadabfragen, sogenannte Property-Path-Ausdrücke[13], innerhalb des WHERE-Teils einer Abfrage zu definieren. Dies war mit SPARQL in der Version 1.0[14] noch nicht möglich. Die Syntax dieser Pfadabfragen orientiert sich an der von regulären Ausdrücken (vgl. Kleene, 1956), nur dass diese auf Prädikate beziehungsweise Eigenschaften (Properties) angewandt werden, und nicht auf Zeichenketten. Während Pfadausdrücke mit fester Länge auch immer durch ein Menge semantisch äquivalenter Graph-Patterns ohne Pfadausdrücke ersetzt werden können, ist dies bei Pfadausdrücke mit beliebiger Länge nicht mehr möglich. Dadurch vereinfachen Pfadabfragen die Notation von SPARQL-Abfragen nicht nur, sondern erweitern die Mächtigkeit der Abfragesprache deutlich. Allerdings ist es innerhalb von Property-Path-Ausdrücken nicht mehr möglich, Variablen zu nutzen. Die zu traversierten Eigenschaften müssen also innerhalb des Ausdrucks definiert

[8] https://www.w3.org/standards/techs/sparql#w3c_all
[9] https://www.w3.org/Submission/RDQL/
[10] https://jena.apache.org
[11] https://www.w3.org/TeamSubmission/turtle/
[12] https://www.w3.org/TR/sparql11-overview/
[13] https://www.w3.org/TR/sparql11-query/#propertypaths
[14] https://www.w3.org/TR/rdf-sparql-query/

Listing 2.2: Eine SPARQL-CONSTRUCT-Abfrage. Im WHERE-Teil der Abfrage werden die zu Albert Einstein gespeicherten Geburtsorte abgerufen und entsprechend ihres Typs auf das Template innerhalb des CONSTRUCT-Teils abgebildet. Dadurch werden zwei neue RDF-Tripel erzeugt. Das eine beschreibt die Relation zwischen Albert Einstein und seiner Geburtsstadt, das andere die Relation zwischen ihm und seinem Geburtsland. Während diese Informationen zuvor nur implizit vorlagen und nur durch entsprechendes Zusatzwissen abgeleitet werden konnten, sind sie nun explizit in den neu generierten Tripeln enthalten.

```
1  PREFIX ex: <http://www.example.com/>
2  CONSTRUCT {
3    dbr:Albert_Einstein ex:cityOfBirth ?city ;
4                        ex:countryOfBirth ?country .
5  }
6  WHERE {
7    dbr:Albert_Einstein dbo:birthPlace ?city .
8    ?city rdf:type dbo:Settlement .
9
10   dbr:Albert_Einstein dbo:birthPlace ?country .
11   ?country rdf:type dbo:Country .
12 }
```

werden und müssen demnach auch bekannt sein, was die Exploration unbekannter Datenbestände mit Hilfe diese Ausdrücke erschwert.

Durch CONSTRUCT-Abfragen lassen sich Subgraphen aus dem Gesamtgraph extrahieren und auch eigene Graphen konstruieren. Während der WHERE-Teil der Abfrage in seinem Aufbau dem einer SELECT-Abfrage entspricht, wird im CONSTRUCT-Teil ein Template beschrieben, auf welches die Variablen des WHERE-Teils abgebildet werden. Listing 2.2 enthält ein Beispiel für eine solche CONSTRUCT-Abfrage. Darin werden im WHERE-Teil der Abfrage, die zu Albert Einstein gespeicherten Geburtsorte abgerufen und entsprechend ihres Typs auf das Template innerhalb des CONTRUCT-Teils abgebildet. Dadurch werden zwei neue RDF-Tripel erzeugt, wobei das erste die Relation zwischen Albert Einstein und seiner Geburtsstadt, das zweite die Relation zwischen ihm und seinem Geburtsland beschreibt. Auf diese Weise lassen sich implizite Informationen, die sonst nur durch entsprechendes Zusatzwissen ableiten ließen, in expliziter Form ausgeben.

Weiterhin lassen sich in SPARQL mittels ASK aussagenlogische Antworten auf (Ab-)Fragen finden. Dabei wird eine Frage mit `true` beantwortet, wenn ein Ergebnis für die gestellte Abfrage existiert, und als FALSE, wenn kein Ergebnis existiert. Listing 2.3 zeigt zwei Beispiele für solche ASK-Abfragen. Die erste Abfrage überprüft, ob Albert Einstein in Ulm geboren wurde. Die Abfrage enthält keine Variablen, sondern nur feste URIs. Da das zu überprüfende BGP im RDF-Graph enthalten ist, wird die Frage mit `true` beantwortet. Die zweite Abfrage überprüft, ob Albert Einstein an einem Ort geboren wurde, der das deutsche Label »Duisburg« besitzt. Da kein Wert des Graphen auf die Variable der Abfrage abgebildet werden kann, wird die Frage dementsprechend mit `false` beantwortet.

Als letzte Form der Datenabfrage existiert in SPARQL die DESCRIBE-Abfrage. Diese Abfrageform liefert weitere Informationen zu explizit definierten Resources aber auch Variablen. Die offizielle SPARQL-Definition lässt offen, welche Daten genau zurückgeliefert werden sollen, doch die meisten SPARQL-Implementationen liefern das jeweilige Prädikat und Objekt aller ein- und ausgehenden Subjekt-Prädikat-Objekt-Relationen für die entsprechende Resource zurück.

Im Gegensatz zu anderen Sprachen existiert in SPARQL keine direkte Möglichkeit zur Nutzung von Inferenzregeln, seien sie nun frei definierbar wie in TRIPLE oder fest vorgegeben wie

Listing 2.3: Zwei SPARQL-ASK-Abfragen. In beiden Abfragen wird überprüft, ob Albert Einstein in einer bestimmten Stadt geboren wurde. Die erste Abfrage enthält keine Variablen, sondern nur feste URIs. Da das zu überprüfende BGP im RDF-Graph enthalten ist, wird die Frage mit `true` beantwortet. Die zweite Abfrage nutzt ein Variable, auf die jedoch kein Wert des Graphen abgebildet werden kann. Die Frage wird dementsprechend mit `false` beantwortet.

```
1 ASK {
2   dbr:Albert_Einstein dbo:birthPlace dbr:Ulm .
3 }
```

```
4 ASK {
5   dbr:Albert_Einstein dbo:birthPlace ?birthPlace .
6   ?birthPlace rdfs:label "Duisburg"@de .
7 }
```

in RQL, welches explizite Funktionen auf RDFS-Basis zur Inferenzbildung enthält. SPARQL überlässt dies dem SPARQL-Endpoint, dem, abhängig von der jeweiligen Implementierung, Inferenzregeln zugeführt werden können. Die abgeleiteten Relationen werden dann automatisch bei der Bearbeitung der Abfrage berücksichtigt. Die Abfragesprache SPARQL selbst ist Schema-frei. SPARQL-Endpoints, welche SPARQL nach der Version 1.1 unterstützen, hingegen sind Schema-agnostisch. Je nach Bedarf kann der SPARQL-Endpoint verlangen, dass die gespeicherten Daten einem Schema oder einer Ontologie gehorchen müssen, oder beliebig gespeichert werden dürfen. Der SPARQL-Standard definiert eine Reihe solcher Vorgaben, sogenannte Entailment Regimes[15], welche gleichzeitig entsprechende Inferenzregeln vorgeben (z. B. auf Basis von RDFS[16]).

Mit Version 1.1 ist auch die Möglichkeit zur Manipulation bestehender und Speicherung neuer Daten hinzugefügt worden.[17] Damit kann SPARQL nicht nur als Abfragesprache, sondern als vollwertige Datenbanksprache betrachtet werden.

Eine Vielzahl von Abfragesprachen wurde veröffentlicht, die SPARQL um verschiedene Eigenschaften erweitern. So ist es mit SPARQL-LD (vgl. Fafalios et al., 2016) möglich, Daten nicht nur aus SPARQL-Endpoints abzufragen, die ihre Daten bereits persistiert vorliegen haben müssen, sondern auch Daten aus anderen Quellen, wie statischen RDF-Dokumente aber auch dynamisch generierte Daten verschiedener Webservices.

Weiterhin wurden SPARQL-Erweiterung zur Verarbeitung von Linked Stream Data (vgl. Sequeda & Corcho, 2009) geschaffen. Hier werden die Daten nicht nur dynamisch zum Zeitpunkt der Abfrage generiert, sondern werden in sich dynamisch zeitlich ändernden Datenströmen übermittelt, wie sie beispielsweise von semantischen Sensornetzen (SSN, vgl. Calbimonte et al., 2012) oder Sozialen Netzwerken erzeugt werden. In EP-SPARQL (vgl. Anicic et al., 2011) lassen sich zeitliche Sequenzen definieren, mittels derer überprüft werden kann, ob ein Ereignis zeitlich nach einem anderen Ereignis eingetreten ist. In Streaming-SPARQL (vgl. Bolles et al., 2008), C-SPARQL (vgl. Barbieri et al., 2009) und CQELS (vgl. Le-Phuoc et al., 2011) lassen sich reguläre SPARQL-Abfrage um Zeitfenster erweitern, welche die Menge der für die Verarbeitung der Abfrage einzubeziehenden Zeitpunkte festlegen. SPARQL-ST (vgl. Perry et al., 2011) erweitert dieses Konzept um eine zusätzliche räumliche Komponente.

[15] https://www.w3.org/TR/sparql11-entailment/
[16] https://www.w3.org/TR/rdf-mt/#RDFSRules
[17] https://www.w3.org/TR/sparql11-update/

Listing 2.4: Ein einfacher Pfadausdruck. Dieser ist, bis auf den Filterausdruck, äquivalent zu der SPARQL-Abfrage aus Listing 2.1. Es werden alle Labels abgefragt, die zu den gespeicherten Geburtsorten Albert Einsteins gehören.

```
1  dbr:Albert_Einstein ⇒ dbo:birthPlace ⇒ rdfs:label
```

Listing 2.5: Ein regulärer Pfadausdruck. Der Punkt (».«) steht für den Wildcard-Operator, das Sternchen (»*«) steht für den Kleene-Stern, welche der Operator für die Berechnung der kleenesche Hülle (vgl. Kleene, 1956) ist. Die Abfrage überprüft also, von Albert Einstein ausgehend, jeden Pfad der Länge gleich Null oder mehr mit jeder beliebigen Eigenschaft. Am Ende des Pfades müssen ein Label oder ein Geburtsdatum stehen.

```
1  dbr:Albert_Einstein ⇒ (.)* ⇒ (rdfs:label | dbo:birthDate)
```

2.1.2 Pfadabfragesprachen

Pfadabfragesprachen fragen Daten in RDF-Datenbanken mittels Pfadausdrücken ab. Diese lassen sich anhand ihrer Ausdrucksstärke in einfache Pfadausdrücke und reguläre Pfadausdrücke unterteilen. Ein einfacher Pfadausdruck beginnt mit einer bestimmten oder unbestimmten Start-Resource, gefolgt von einer beliebig, jedoch fest definiert langen Reihe von Prädikaten, welche einen Pfad innerhalb des RDF-Graphen beschreiben. Diese Ausdrücke können als Abkürzungen für aufeinanderfolgende Subjekt-Prädikat-Objekt-Relationen der Pattern-basierten Abfragesprachen gesehen werden. Bis auf die verkürzte Schreibweise liefern sie keinen Mehrwert gegenüber diesen. Da Pfadabfragesprachen in der Regel eine Ergebnisliste erzeugen, deren Einträge sich ausschließlich auf das letzte Element des Pfades beziehen, entsteht sogar der Nachteil, dass auf die zwischenzeitlich traversierten Elemente des Graphen nicht zugegriffen werden kann, ohne eine zusätzliche Pfadabfrage zu stellen. Listing 2.4 zeigt ein Beispiel eines einfachen Pfadausdrucks. Dieser ist, bis auf den Filterausdruck, äquivalent zu der SPARQL-Abfrage aus Listing 2.1 und fragt alle Labels ab, die zu Albert Einsteins Geburtsorten gespeichert sind. In diesem Beispiel wurde ein Pfeil als generischer Pfadoperator gewählt. In den jeweiligen konkreten Implementationen der Abfragesprachen werden jedoch unterschiedliche Zeichen als Pfadoperatoren genutzt. Gängige Zeichen sind der Schrägstrich (»/«), der umgekehrte Schrägstrich (»\«), der Doppelpunkt (»:«), der Punkt (».«) oder das Größer-als-Zeichen als stilisierter Pfeil (»>«), manchmal auch in Kombination mit dem Bindestrich (»->«).

Viele Pfadabfragesprachen für RDF-Daten ermöglichen die Definition von Abfragepfaden mittels regulärer Pfadausdrücke (vgl. Mendelzon & Wood, 1995), welche sich an der Notation von regulären Ausdrücken orientieren. Wie die bereits in Abschnitt 2.1.1 beschriebenen Property-Path-Ausdrücke in SPARQL, erlauben sie die Beschreibung beliebig langer Pfade. Allerdings erlauben sie im Gegensatz zu diesen auch den Wildcard-Operator, sodass Pfade mit beliebigen Eigenschaften traversiert werden können. Abbildung 2.5 zeigt einen solchen regulärer Pfadausdruck. Dieser fragt alle Labels und Geburtsdaten ab, die über einen beliebig langen Pfad mit Albert Einstein verbunden sind.

Da RDF ursprünglich ausschließlich in XML-Form serialisiert wurde[18], wurden auch immer wieder XML-Pfadabfragesprachen genutzt, um Informationen aus RDF-Dokumenten abzufra-

[18] https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/

gen. XML-Pfadabfragesprachen wie XPath[19], welches auch von XSLT[20] und XQuery[21] zur Datenabfrage genutzt wird, XQL[22] oder Quilt (vgl. Chamberlin et al., 2001) betrachten die Daten eines XML-Dokuments als geordneten Baum. Ein RDF-Dokument beschreibt hingegen einen Graphen, in welchem weder eine Ordnung noch eine Hierarchie existiert, wodurch viele der Funktionen dieser Abfragesprachen nicht wie angedacht genutzt werden können. Hinzu kommt, dass die Serialisierung von RDF-Daten in XML nicht eindeutig ist. Informationen können beispielsweise als Attribute oder Elemente gespeichert werden, was dazu führt, dass semantisch äquivalente Dokumente eine unterschiedliche Syntax aufweisen können. Das Ergebnis der Verarbeitung eines RDF-Dokuments durch eine dieser XML-Abfragesprachen ist daher in erster Linie von der Syntax statt der Semantik des Dokuments abhängig. Auch sind diese Abfragesprachen nicht dazu in der Lage, implizites Wissen aus den RDF-Daten mittels Inferenz abzuleiten.

Die von Pietriga et al. (2006) im Rahmen des Fresnel-Projekts[23] (vgl. Abschnitt 3.1.2) entwickelte Fresnel Selector Language (FSL)[24], orientiert sich stark an XPath und wurde als einfach zu erlernende und zu implementierende Pfadabfragesprache entworfen. Aus diesen Gründen wurden einige Einschränkungen bezüglich der Ausdrucksfähigkeit gemacht, sodass FSL nicht als vollwertige reguläre Pfadabfragesprache bezeichnet werden kann. XPR (vgl. Cohen-Boulakia et al., 2006) erweitert FSL daher um fehlende Funktionen, wie einen Operator für die kleensche Hülle und um Funktionen, um auch auf Eigenschaften und nicht nur Resources zugreifen zu können. Trotz der Einschränkungen denen FSL unterliegt, wurde die Abfragesprache in einigen Projekten implementiert (z. B. IsaViz[25]; Piggy Bank, vgl. Huynh et al., 2005; LENA, vgl. Koch & Franz, 2008).

nSPARQL (vgl. Pérez et al., 2008) ermöglicht nicht nur den Einsatz von einfachen regulären Pfadausdrücken, sondern auch den von verschachtelte reguläre Pfadausdrücken. LDPath (vgl. Schaffert et al., 2012) wurde als Abfragesprache für das Linked Media Framework entwickelt, mit dem Hauptschwerpunkt der Suche und Indizierung von Linked Data. Dementsprechend verfügt die Sprache über eine Reihe von Funktionen, die für die gewöhnliche Nutzung von Abfragesprachen überflüssig sind, wie zum Beispiel die Abfrage ob ein Element des Graphen bereits in den Suchindex aufgenommen wurde. Das Linked Media Framework und LDPath wurden mittlerweile in das Apache Marmotta-Projekt[26] überführt und werden dort weiterentwickelt. RDFPath (vgl. Przyjaciel-Zablocki et al., 2012) wurde als Pfadabfragesprache zur Analyse von großen RDF-Graphen entwickelt und enthält daher spezielle Funktionen, welche diese Analyse ermöglichen, wie zum Beispiel die Berechnung des kürzesten Pfades zwischen zwei Knoten.

Während einige der Pattern-basierte Abfragesprachen auch die Möglichkeit zur pfadbasierte Datenabfragen bieten, nutzt die Pfadabfragesprache NautiLOD (vgl. Fionda et al., 2012) Pattern-basierte Abfragen auf Basis von SPARQL, um Daten zu Filtern oder Aktionen auszuführen. NautiLOD wurde entwickelt, um Abfragen über mehrere SPARQL-Endpoints hinweg zu stellen. Dabei werden gleiche Resources anhand der `owl:sameAs`-Relation zusammengefügt. Die Syntax des pfadbasierten Teils der Abfragesprache orientiert sich an XPath und dessen regulären Pfadausdrücken. Filter (*tests*) bestehen aus ASK-Abfragen, die bestimmen, ob eine Relation der

[19] `https://www.w3.org/TR/xpath/`
[20] `https://www.w3.org/TR/xslt/`
[21] `https://www.w3.org/XML/Query/`
[22] `https://www.ibiblio.org/xql/xql-proposal.html`
[23] `https://www.w3.org/2005/04/fresnel-info/`
[24] `https://www.w3.org/2005/04/fresnel-info/fsl/`
[25] `https://www.w3.org/2001/11/IsaViz/`
[26] `https://marmotta.apache.org`

Ergebnismenge hinzugefügt werden soll. Weiterhin können Aktionen (*actions*) ausgeführt werden, die während der Navigation durch den Linked-Data-Graph ausgelöst werden. Beispielsweise kann eine Aktion so definiert sein, dass eine E-Mail an bestimmte Personen geschickt werden soll. Welche Personen dafür ausgewählt werden sollen, kann durch eine SPARQL-SELECT-Abfrage bestimmt werden, wobei diese SELECT-Abfrage auf die Variablen einer möglicherweise zuvor gestellten ASK-Abfrage eines Filters zurückgreifen kann.

Eine große Schwäche all dieser zuvor vorgestellten Pfadabfragesprachen ist es, dass sie lediglich dazu in der Lage sind, eindimensionale Ergebnislisten zurückzugeben, welche sich in der Regel auf das letzte Element des Pfadausdrucks bezieht. Ihnen fehlt die Möglichkeit zur Projektion der während der Traversierung des Pfades besuchten Elemente, auf vorgegebene Variablen. Da die Pfade der regulären Pfadausdrücke unterschiedlich und beliebige lang, in zyklischen Graphen sogar unendlich lang sein können, lässt sich dies auch nicht trivial ermöglichen. Um die einzelnen Elemente des Pfades zu erhalten, müssten daher jeweils einzelne Abfragen fester Länge erstellt werden, deren Ergebnisse nachträglich wieder verknüpft werden. Dabei ist nicht sichergestellt, dass sich zwischen zwei Abfragen die Werte des Pfades nicht geändert haben, insbesondere bei sich stetig ändernden Graphen (z. B. bei SSNs oder sozialen Netzwerken), und sich die Daten somit nicht mehr korrekt zusammenfügen lassen.

Eine weitere Schwäche einiger dieser Pfadabfragesprachen ist es, dass sie ausschließlich auf Schema-Ebene und nicht auf Instanz-Ebene arbeiten können. Dies bedeutet, dass sie alle Daten einer vorgegebenen Klasse oder eines Typs abfragen. Um mit diesen Sprachen Daten zu einer bestimmten Instanz beziehungsweise einer bestimmten Resource abzufragen, sind zusätzliche Filterabfragen erforderlich. Listing 2.6 zeigt diese Auswahl in FSL. Hier werden zunächst alle Elemente des Graphen gewählt und anhand ihrer URI auf die Resource zu Albert Einstein eingegrenzt.

2.1.3 Zusammenfassung und Fazit

Die hier vorgestellten Abfragesprachen weisen eine Vielzahl individueller Stärken und Schwächen auf, von denen sich einige auf das jeweils genutzte Abfrageparadigma zurückführen lassen. So weisen die Pattern-basierten Abfragesprachen einen größeren Funktionsumfang auf, als die pfadbasierten Abfragesprachen. Keine der Pfadabfragesprachen besitzt beispielsweise eine Funktion, mit der sich die selben Ergebnisse, wie mit einer SPARQL-CONSTRUCT-Abfrage erzielen lassen. Demgegenüber steht die Schwäche der Pattern-basierten Abfragesprache hinsichtlich der Abfrage beliebig langer Pfade. Für diese Zwecke binden diese wiederum Pfadabfragen ein, die teilweise jedoch nicht die selbe Mächtigkeit wie die der reinen Pfadabfragesprachen aufweisen und aufgrund ihrer Syntax als Fremdkörper innerhalb der Pattern-Abfrage wirken. Im Gegensatz zu den Abfragen der reinen Pfadabfragesprachen, können Property-Path-Abfragen in SPARQL beispielsweise keine Variablen oder Wildcard-Operatoren enthalten, wie es beispielsweise in Listing 2.5 dargestellt wird.

Pfadabfragesprachen arbeiten in der Regel auf Schema-Ebene und nicht auf Instanz-Ebene. Das Startelement des Pfades entspricht also keiner Resource, wie in Listing 2.4 oder 2.5, bei denen die Abfrage mit der Resource zu Albert Einstein startet, sondern beginnt direkt mit einer Eigenschaft, die auf sämtliche Elemente des Graphen abgeglichen wird. Soll der Pfad mit einer Resource beginnen muss diese daher meist mittels eines komplexen Filterausdrucks ausgewählt werden.

Listing 2.6: Auswahl einer Resource in FSL. Zunächst werden alle Elemente des Graphen ausgewählt. Diese Elementenmenge wird anschließend gefiltert. Dabei wird mittels einer Funktion die URI des jeweiligen Elements ermittelt und mit der präfigierten URI der Resource zu Albert Einstein abgeglichen.

```
1  *[uri(.) = exp('dbr:Albert_Einstein')]
```

Eine weitere Schwäche der Pfadabfragesprachen ist das Fehlen einer Möglichkeit zur Projektion der während der Traversierung des Pfades besuchten Elemente auf vorgegebene Variablen. Damit lassen sich ausschließlich eindimensionale Ergebnislisten erzeugen, die in der Regel das jeweils letzte Element der möglichen Pfade innerhalb des Graphen enthält. Aus selben Grund fehlt diesen Abfragesprachen die Möglichkeit zur Verzweigung von Pfaden. Zwar können reguläre Pfadausdrücke alternative Eigenschaften definieren, doch werden die Ergebnisse immer wieder auf eine eindimensionale Ergebnisliste reduziert, der nicht zu entnehmen ist, welcher der alternativen Eigenschaften der Pfad gefolgt ist.

Leider sind zu den hier vorgestellten Abfragesprachen keine Studien bekannt, in denen untersucht wurde, wie gut oder auch schlecht deren Anwender diese nutzen oder erlernen konnten. Da der Funktionsumfang der Pattern-basierten Abfragesprachen im Vergleich zu den Pfadabfragesprachen jedoch höher ist, einhergehend mit einer höheren Zahl von Schlüsselwörtern, welche von den Anwendern gelernt werden müssen, wird hier die Hypothese aufgestellt, dass Pfadabfragesprachen von Nutzern leichter zu erlernen und zu nutzen sind als Pattern-basierte Abfragesprachen. Die Ergebnisse der in Abschnitt 2.4 vorgestellten Studie sprechen – zumindest hinsichtlich der beiden Abfragesprachen SPARQL und SemwidgQL als Vertreter der beiden Abfrageparadigmen – für diese Hypothese.

2.2 Konzeptionelle Überlegungen

Nachfolgend werden die konzeptionellen Überlegungen zur Entwicklung der Abfragesprache SemwidgQL beschrieben. Dazu wird zuerst ein Überblick über die Ziele und Fähigkeiten der potenziellen Nutzer des Semwidg-Projekts (vgl. Abschnitt 2.2.1) gegeben. Anschließend werden die Anforderungen, welche sich aus diesen Zielen und Fähigkeiten sowie der zuvor erfolgten Analyse der Abfragesprachen für Daten im Linked-Data-Bereich (vgl. Abschnitt 2.1) ergeben, abgeleitet.

2.2.1 Potenzielle Nutzer von SemwidgQL

Die potenziellen Anwender von SemwidgQL sind in erster Linie Programmierer und Webentwickler mit Programmierkenntnissen (vgl. Abschnitt 1.2.3), die Linked Data in Webseiten einbinden und diese Daten gegebenenfalls visualisieren möchten. Die Anwender sind mit objektorientierten Programmiersprachen vertraut und haben die dahinterstehenden Konzepte verinnerlicht. Sie sind jedoch nur bedingt mit den Abfragesprachen des Linked-Data-Bereichs vertraut. Eine SPARQL-Abfrage können sie daher nicht ohne weitere Hilfe formulieren.

2.2.2 Anforderungen an SemwidgQL

Aufbauend auf den zuvor beschriebenen Zielen und Fähigkeiten der potenziellen Nutzer des Semwidg-Projekts (vgl. Abschnitt 2.2.1) und den Erkenntnissen, die sich aus der Analyse der Abfragesprachen für Daten im Linked-Data-Bereich (vgl. Abschnitt 2.1), im Speziellen deren herausgearbeiteten Stärken und Schwächen (vgl. Abschnitt 2.1.3), werden im nachfolgenden Abschnitt die Anforderungen an die Abfragesprache SemwidgQL beschrieben.

Effektivität, Effizienz und leichte Erlernbarkeit: Im Vordergrund der konzeptionellen Überlegungen zu SemwidgQL stehen die klassischen Usability-Kriterien Effektivität und Effizienz sowie eine leichte Erlernbarkeit der Sprache. Insbesondere die Pattern-basierten Abfragesprachen weisen einen großen Funktionsumfang auf, welcher gleichsam zu einem hohen Lernaufwand für neue Nutzer führt. Auch einfachste Abfragen erfordern erheblichen Schreibaufwand, welcher die Effizienz bei der Formulierung von Abfragen mindert. Aus dem hohen Lern- und Schreibaufwand resultiert eine erhöhte Gefahr für Fehler, die dem Nutzer bei der Erstellung der Abfragen unterlaufen können. SemwidgQL sollte Abfragen daher auf die nötigsten Informationen beschränken und nur relevante Funktionen bieten.

Übersetzung in SPARQL: In der Praxis hat sich SPARQL als Abfragesprache für Daten im RDF-Format durchgesetzt. Die meisten Datenbanken, Frameworks und Programme aus dem Linked-Data-Bereich bieten mittlerweile eine Schnittstelle, um die gespeicherten Daten mittels dieses Standards abzufragen. Von den 1350 in der LOD-Cloud[27] verzeichneten Datenbanken verfügen 572 über einen SPARQL-Endpoint[28]. Diese wiederum enthalten über 80 % der mehr als 200 Milliarden RDF-Tripel, aus welchen die LOD-Cloud insgesamt besteht. Um auf diesen enormen Datenbestand zugreifen zu können, muss es mit SemwidgQL möglich sein, SPARQL-Endpoints anzusprechen.

[27] `https://lod-cloud.net`
[28] Stand: Juni 2019

Da es illusorisch ist anzunehmen, dass all diese SPARQL-Endpoints einmal SemwidgQL nativ verarbeiten werden können, muss es stattdessen möglich sein, eine SemwidgQL-Abfrage in eine SPARQL-Abfrage zu übersetzen beziehungsweise zu transkompilieren. Im Vergleich zu den zuvor vorgestellten Abfragesprachen wäre dies ein großer Vorteil, denn für sie existiert, bedingt durch den jeweils unterschiedlichen Funktionsumfang und daraus resultierender Inkompatibilität, eine solche Möglichkeit nicht. Auch wenn die SPARQL-Spezifikation in der Version 1.1 nun schon einige Jahre alt ist, unterstützen immer noch nicht alle SPARQL-Endpoints diesen Standard. SemwidgQL sollte sich daher, soweit es möglich ist, auf Funktionen von SPARQL 1.0 beschränken.

Reduktion auf relevante Funktionen: Die Analyse des LSQ-Datensatzes (vgl. Abschnitt 2.5.1) hat ergeben, dass der Großteil der genutzten SPARQL-Abfragen SELECT-Abfragen sind. Um SemwidgQL möglichst einfach zu halten, kann die Sprache daher auf SELECT-Abfragen begrenzt werden. Ebenso hat sich gezeigt, dass viele der SPARQL-Ausdrücke wie UNION, GROUP BY und HAVING nur sehr selten genutzt werden. Es kann also auch darauf verzichtet werden, diese Ausdrücke in SemwidgQL abzubilden.

Stattdessen sollten Funktionen eingeführt werden, welche in SPARQL sonst nur aufwendig zu formulierende Teile von Abfragen erheblich vereinfachen können. Beispielsweise erfordert die Filterung von Zeichenketten anhand ihrer Sprache zwei verschachtelte Funktionsaufrufe innerhalb eines Filterausdrucks. Noch komplexer gestaltet sich der Vergleich von zwei Zeichenketten. Dieser erfolgt ebenfalls wieder durch eine Vielzahl ineinander verschachtelter Funktionsaufrufe innerhalb eines Filterausdrucks oder mittels regulärer Ausdrücke. Solche komplexen Ausdrücke können deutlich vereinfacht werden. Die Abfragesprache XIRQL (vgl. Fuhr & Großjohann, 2001) verfügt beispielsweise neben den üblichen Vergleichsoperatoren über einen *contains*-Operator, der überprüft, ob eine Zeichenkette in einer anderen Zeichenkette enthalten ist.

Objekt-orientierte Betrachtung: Die Daten eines RDF-Graphen lassen sich verhältnismäßig einfach in eine untereinander verknüpfte Menge von Klasseninstanzen einer objektorientierten Programmiersprache umwandeln. Abbildung 2.3 zeigt eine Möglichkeit, wie sich der RDF-Graph aus Abbildung 2.1 als UML-Objektdiagramm darstellen lässt. Ein großer Unterschied von RDF zu den Klassen objektorientierter Programmiersprachen ist es, dass Eigenschaften nicht nur einzelne Werte, sondern ihnen stets eine beliebig große Anzahl von Werten zugeordnet sein kann. Dies lässt sich zwar mittels Ontologien einschränken, sodass eine Eigenschaft immer nur einmal pro Resource existieren darf, doch muss ein generischer Ansatz davon ausgehen, dass zu einer Eigenschaft stets eine Liste von Werten existiert.

Die Datentypen dieser Werte sind auch nicht zwingend homogen. So existiert im hier genutzten Beispiel nur ein Wert zu Albert Einsteins Geburtstag, in DBpedia sind jedoch zwei Werte gespeichert. Einmal als einfaches String-Literal und einmal als Datumswert (`xsd:date`). Aus Sicht eines Programmierers können die Eigenschaften einer Resource also vereinfacht als untypisiertes Array betrachtet werden und der gesamte RDF-Graph als eine untereinander verknüpfte Menge von Objekten.

Zwar ist diese Betrachtungsweise nicht absolut korrekt, doch kann das komplexe Gebilde eines RDF-Graphen so auch Anwendern aus der Gruppe der Programmierer – weiterhin

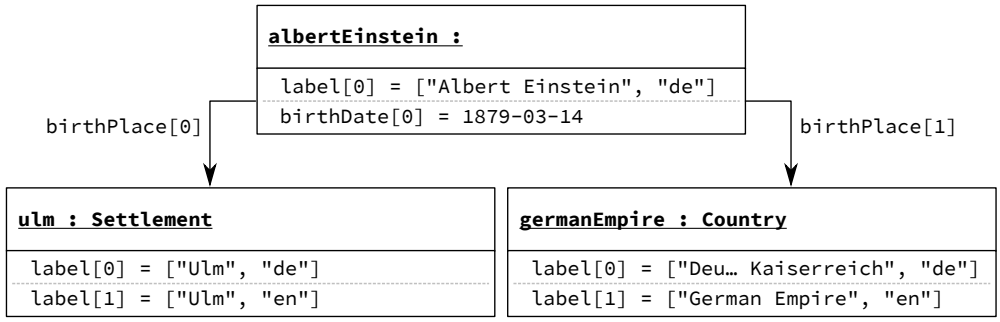


Abbildung 2.3: Ein UML-Objektdiagramm. Die Daten entsprechen denen des RDF-Graphen aus Abbildung 2.1. Eigenschaften enthalten keine einzelnen Werte, sondern können stets eine unbestimmte Menge von Werten enthalten, und werden daher als Array aufgefasst.

die Hauptzielgruppe des Semwidg-Projekts (vgl. Abschnitt 1.2.3) – recht leicht verständlich gemacht werden. So ist in dem hier gezeigten UML-Objektdiagramm nicht abgebildet, dass auch Eigenschaften selbst wieder als Resource fungieren können, die wiederum selbst beliebige Eigenschaften besitzen können. Daher existieren Eigenschaften im RDF-Graph nicht nur im lokalen Kontext einer Klasseninstanz, welcher sie zugeordnet sind, sondern auch global. Ebenfalls ist nicht abgebildet, dass beispielsweise die Eigenschaft `label` nicht nur zufälligerweise die gleich Bezeichnung in allen drei Klasseninstanzen aufweist, sondern dass es sich tatsächlich in allen Fällen um die gleiche Eigenschaft handelt, der nur unterschiedliche Werte zugeordnet wurden.

Letztendlich ist es jedoch naheliegend, für die Konzeption von SemwidgQL den kompletten RDF-Graphen als Menge von verknüpften Objekten im Kontext der objektorientierten Programmierung zu betrachten, welche in einem global verfügbaren Model persistiert wurde. Auf die einzelnen Objekte des Models kann anhand der entsprechenden URIs zugegriffen werden, so als entsprächen sie statischen Variablen dieses Models. Nachdem ein bestimmtes Objekt aus dem Model gewählt wurde, kann auf dessen Eigenschaften zugegriffen werden.

Um auf die Eigenschaften eines Objekts zuzugreifen, nutzen viele Programmiersprachen die Punktnotation. Unter andrem auch JavaScript, welches im Rahmen von SemwidgJS (vgl. Kapitel 3) genutzt wird. Es bietet sich also an, diese Punktnotation auch für SemwidgQL zu wählen.

Pfadabfragesprache: Wie zuvor beschrieben, sollen die Resources des RDF-Graphen direkt als Objekte ausgewählt werden können und sich dessen Eigenschaften mittels der Punktnotation auswählen lassen. Da die Ergebnisse dieser Auswahl selbst wieder Objekte sein können, können deren Eigenschaften ebenfalls mittels Punktnotation ausgewählt werden. Die Konzeption von SemwidgQL als Pfadabfragesprache ist daher ebenfalls naheliegend. Es ist jedoch zu beachten, dass die Eigenschaften im RDF-Kontext immer mehrerer Werte enthalten können.

Der nächste Schritt einer Pfadabfrage geht also nicht von einem einzelnen Objekt sondern von allen Objekten der vorherigen Ergebnismenge aus. Diese Funktion ist für objektorientierte Programmiersprachen eher ungewöhnlich, wird in der Praxis dennoch häufig

Listing 2.7: Traversierung einer Webseite mit jQuery. Zuerst werden alle geordneten Listen (»ol«) innerhalb der Webseite gesucht. Davon ausgehend werden alle Kindelemente dieser Listen traversiert und alle Elemente an gerader Stelle herausgefiltert. Der Hintergrund der Elemente an ungerader Stelle wird anschließend rot gefärbt.

```
1  $( "ol" ).children().filter( ":even" ).css( "background-color", "red" );
```

eingesetzt. So können beispielsweise in der JavaScript-Bibliothek jQuery[29] bei der Traversierung[30] des Inhalts einer Webseite Funktionen auf einer Menge von Ergebnissen ausgeführt werden, ohne jedes Element einzeln auszuwählen. Listing 2.7 zeigt eine solche Traversierung zuzüglich eines Funktionsaufrufs.

Da SemwidgQL-Abfragen in erster Linie innerhalb von SemwidgJS-Widgets genutzt werden sollen und dort als Werte von HTML-Attributen abgelegt werden, erhöht die Formulierung der Abfragen in Pfadform zusätzlich die Lesbarkeit im Vergleich zu einer SPARQL-Abfrage, welche über mehrere Zeilen übergreifend formuliert wird.

Möglichkeit der Verzweigung von Pfaden: Eine große Schwäche der regulären Pfadabfragen ist das Fehlen von verzweigten Pfaden. Um mehrere Eigenschaften einer Resource abzufragen sind daher immer mehrere separate Abfragen nötig. Dies erhöht nicht nur den Aufwand für den Nutzer bei der Datenabfrage, sondern kann auch zu inkonsistenten Ergebnissen führen.

Werden beispielsweise die Koordinaten der Geburtsorte Albert Einsteins oder gegebenenfalls auch aller Träger des Nobelpreises für Physik abgefragt, erhält der Nutzer einer Pfadabfragesprache zwei getrennte Listen. Eine für die Längen- und eine für die Breitengrade. Es ist jedoch nicht sichergestellt, dass die Elemente an gleicher Stelle in beiden Listen auch zusammengehören. Der intern genutzte Algorithmus, der für die Ermittlung der Ergebnisse verantwortlich ist, kann beide Liste unterschiedlich sortiert haben. Noch problematischer wird es bei sich häufig verändernden Datenbanken, wie zum Beispiel semantischen Sensornetzen. Zwischen den beiden Abfragen können bereits weitere Elemente hinzugefügt worden sein, sodass getrennt gestellte Abfragen auf unterschiedlichen Graphen ausgeführt werden.

SemwidgQL sollte es also ermöglichen, mehrere Eigenschaften einer Resource in einer einzelnen Abfrage abzufragen. Dies führt zu einer Verzweigung des Pfades. Die abzweigenden Pfade sollten beliebig lang formuliert und selbst erneut verzweigt werden können, um auch Daten über mehrere verlinkte Resources hinweg abfragen zu können.

Rückgabe des gesamten Pfades: Für komplexe Visualisierungen der abgerufenen Daten können nicht nur die Daten selbst, sondern auch die Daten des traversierten Pfades von Interesse sein. Zum Beispiel bei der Darstellung von Zusammenhängen zwischen zwei Resources in Graphform, ähnlich der Darstellungsweise des RelFinders (vgl. Heim et al., 2009). Es sollte daher mit SemwidgQL möglich sein, nicht nur die letzte Eigenschaft eines Pfades abzurufen, sondern auch alle Elemente, aus denen sich der Pfad zusammensetzt.

Die hier beschriebenen Anforderungen stellen die Grundanforderungen dar, welche an SemwidgQL gestellt werden. Zusätzlich wurden weitere Funktionen implementiert, welche in

[29] https://jquery.com
[30] https://api.jquery.com/category/traversing/

Abschnitt 2.3 beschrieben werden. So wurden beispielsweise Funktionen hinzugefügt, die die Abfrage von mit Zeitstempeln annotierter Daten vereinfacht, welche unter anderem bei der Speicherung von Sensordaten anfallen können.

2.2.3 Fazit

Aus den hier vorgestellten Anforderungen an SemwidgQL ergeben sich einige Vorteile gegenüber SPARQL aber auch einige Einschränkungen. Ein großer Vorteil ist beispielsweise, dass aufgrund der einfachen Syntax und geradlinigen Beschreibung der Pfade, entsprechende Anwendungen Nutzer bei der Formulierung von SemwidgQL-Abfragen leichter unterstützen können. Da die potenziellen Nutzer von SemwidgQL nur bedingt mit den Konzepten und Techniken des Linked-Data-Bereichs vertraut sind, werden sie eine solche Hilfe benötigen, die sich für SPARQL-Abfragen nur schwer umsetzen lässt. Innerhalb einer SPARQL-Abfrage kann jedes Triple-Pattern unabhängig von den anderen Elementen der Abfrage existieren oder auch mit diesen verknüpft sein. Ebenso kann an beliebiger Stelle des WHERE-Teils der Abfrage ein neuer Filterausdruck eingeführt werden, sodass eine unterstützende Anwendung nur schwer eine konkrete Aussage über die möglichen nachfolgenden Optionen des Nutzers machen kann. Die Zahl der Möglichkeiten kann unendlich groß sein. Die Beschreibung von Pfaden schränkt diese Möglichkeiten hingegen deutlich ein. Beginnt ein Pfad mit einer konkreten Resource, kann leicht bestimmt werden, welche Eigenschaften für den nächsten Schritt in Frage kommen. Diese können dem Nutzer bei der Pfadformulierung vorgeschlagen werden.

Die gestellten Anforderungen an SemwidgQL sollen zwar eine einfache Nutzung ermöglichen, führen jedoch zu einigen Einschränkungen. Da sich SemwidgQL wie eine Pfadabfragesprache verhalten, jedoch zu SPARQL transkompiliert werden soll, kann SemwidgQL natürlich höchstens die Funktionen bieten, die auch SPARQL bietet. Im Gegensatz zu regulären Pfadabfragesprachen können mit SemwidgQL keine beliebig langen Pfade gebildet werden. Denn die Anforderung, den gesamten Pfad zurückgeben zu können, verhindert, dass SPARQL Property Paths genutzt werden können. Ebenso ist dies eine der Funktionen der SPARQL-Version 1.1, welche, soweit möglich, gemieden werden sollten. Die Möglichkeit zur Verzweigung von Pfaden und zur Rückgabe des gesamten Pfaden, erweitern jedoch das Konzept der Pfadabfragesprachen deutlich.

2.3 Beschreibung der Abfragesprache SemwidgQL

SemwidgQL wurde als Pfadabfragesprache konzipiert, die sich zu SPARQL transkompilieren lässt. Im Gegensatz zu den anderen Pfadabfragesprachen des Linked-Data-Bereichs (vgl. Abschnitt 2.1.2) ist es daher mit SemwidgQL möglich, ohne Weiteres auf eine Vielzahl öffentlicher SPARQL-Endpoints und deren umfangreiche Datenmengen zuzugreifen. Bei der Definition der Sprache wurde darauf geachtet, soweit es möglich war, kompatibel zum SPARQL-Standard 1.0 zu bleiben. Einige der im später folgenden Abschnitt 2.3.3 beschriebenen Filter-Schlüsselwörter und Pseudo-Filter-Schlüsselwörter sorgen jedoch dafür, dass beim Einsatz dieser nur noch zu SPARQL 1.1 kompatible Abfragen erzeugt werden. Ansonsten konnten die in Abschnitt 2.2.2 aufgestellten Anforderungen an SemwidgQL allesamt umgesetzt werden.

2.3.1 Sprachdefinition

Eine vollständige SemwidgQL-Abfrage wird definiert durch ein Tripel (P, R, Q) bestehend aus einer Zuordnungsliste mit Präfixdefinitionen P, einer Zuordnungsliste mit benannten Resources R (vgl. Abschnitt 2.3.2: *Resource-Auswahl*) und dem eigentlichen Abfragetext Q. Die beiden Listen P und R können leer sein. Der SemwidgQL-zu-SPARQL-Transcompiler erzeugt aus den drei Elementen des Tripels (P, R, Q) eine SPARQL-Abfrage. Damit unterscheidet sich SemwidgQL von den meisten anderen Abfragesprachen, bei denen sämtliche relevanten Informationen im Abfragetext enthalten sein müssen. Sprachlich vereinfachend wird im weiteren Verlauf dieser Arbeit, wenn nicht anderes beschrieben, eine SemwidgQL-Abfrage mit dem eigentlichen Abfragetext Q gleichgesetzt.

Präfixdefinitionen

Die Zuordnungsliste P enthält beliebig viele Paare (p, u) von Präfixen p und URIs u in Form von Zeichenketten. Werden gleiche Präfixe mehrfach genutzt, überschreibt das nachfolgende (p, u)-Paar immer das Vorgängerpaar, welches das selbe Präfix nutzt. Der Transcompiler überprüft, welche Präfixe in der SemwidgQL-Abfrage vorkommen und schreibt die entsprechenden Präfixdeklarationen, falls das Präfix in P vorhanden ist, in den Prolog der SPARQL-Abfrage. Ein in einer SemwidgQL-Abfrage genutztes Präfix muss nicht zwingend in P enthalten sein. Manche SPARQL-Endpoints können diese selbst auflösen.

Während das Abtrennen der Präfixdefinitionen vom eigentlichen Abfragetext bei einer einzelnen Abfrage zu einem geringen Mehraufwand bei der Erstellung führen kann, kann es jedoch bei mehreren Abfragen den Schreibaufwand deutlich reduzieren, da so häufig redundant zu definierende Präfixe in verschiedenen Abfragen wiederverwendet werden können.[31]

Benannte Resources

Anstatt eine fixe und häufig lange vollständige oder präfigierte URI in die Abfrage einzufügen, kann eine selbstdefinierte benannte Resource genutzt werden, die sich wie eine Variable verhält. Diese wird anhand der Zuordnungsliste R definiert, welche beliebig viele Paare (r, u) von Resource-Namen r und vollständigen oder präfigierte URIs u in Form von Zeichenketten enthält. Werden gleiche Resource-Namen mehrfach genutzt, überschreibt das nachfolgende (r, u)-

[31] Der SemwidgJS-Bibliothek wurde eine Liste mit mehreren hundert häufig genutzten Präfixen beigefügt, die automatisch an den Transcompiler übergeben wird. Dies führt dazu, dass in vielen Fällen gar keine Präfixdefinition durch den Anwender von SemwidgJS mehr erfolgen muss.

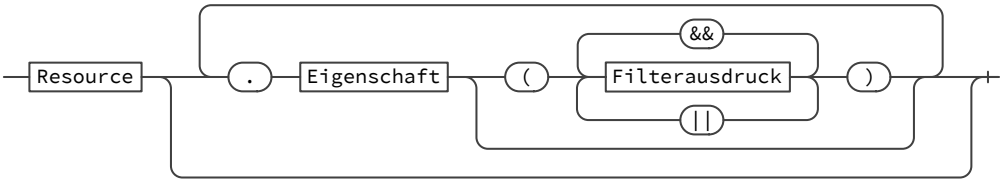


Abbildung 2.4: Vereinfachte Struktur einer SemwidgQL-Abfrage. Üblicherweise beginnt eine Abfrage mit einer Resource, gefolgt von einer oder mehrerer Eigenschaften, die durch Punkte von einander getrennt werden. Eine Eigenschaft kann durch beliebig viele Filterausdrücke eingeschränkt werden, die mittels Konjunktion oder nicht-ausschließender Disjunktion miteinander verknüpft werden.

Paar immer das Vorgängerpaar, welches den selben Resource-Namen nutzt. Der SemwidgQL-zu-SPARQL-Transcompiler ersetzt während der Übersetzung jeden Resource-Namen durch die entsprechende URI. Dies ermöglicht die einfache Wiederverwendung von Abfragen mit unterschiedlichen Resources sowie die einfache Manipulation dieser Abfragen. Jeder in der SemwidgQL-Abfrage genutzte Resource-Name muss in R definiert sein.

Abfragetext

Der Abfragetext Q einer SemwidgQL-Abfrage ist eine Zeichenkette, deren Aufbau einer Reihe von Regeln in der EBNF-Syntax (vgl. Wirth, 1996) folgen muss, welche in Anhang A, zusammen mit den dazugehörigen Syntaxdiagrammen, beschrieben werden.

Abbildung 2.4 zeigt den vereinfachten Aufbau einer SemwidgQL-Abfrage anhand eines Syntaxdiagramms. Üblicherweise beginnt der Abfragepfad mit einer Resource, optional gefolgt von einer oder mehrerer Eigenschaften, die durch Punkte von einander getrennt werden. Eine Eigenschaft kann durch beliebig viele Filterausdrücke eingeschränkt werden, die mittels logischer Konjunktion oder Disjunktion miteinander verknüpft werden.

Die Grundfunktionalitäten von SemwidgQL, welche in dem vereinfachten Syntaxdiagramm (vgl. Abbildung 2.4) zum Einsatz kommen, werden im nachfolgenden Abschnitt 2.3.2 beschrieben. Erweiterte Funktionen, die die Mächtigkeit von SemwidgQL erhöhen und deren Nutzung vereinfachen, werden anschließend in Abschnitt 2.3.3 beschrieben.

Das Ergebnis einer SemwidgQL-Abfrage ist das Ergebnis, das aus der daraus erzeugten SPARQL-Abfrage resultiert, und besteht aus einer Ergebnisliste die beliebig viele gleich strukturierte assoziative Datenstrukturen enthält. Diese Datenstrukturen enthalten jeweils mindestens die Werte einer möglichen Lösung für die Abfrage sowie die dazugehörigen Bezeichnungen der genutzten Variablen. SemwidgQL-Abfragen erzeugen grundsätzlich »distinkte« SPARQL-Abfragen, womit die Elemente der Ergebnisliste innerhalb dieser Liste einmalig sind. Der SemwidgQL-zu-SPARQL-Transcompiler fügt dem SELECT-Ausdruck der erzeugten SPARQL-Abfrage nur die Start-Resource der SemwidgQL-Abfrage sowie alle Variablen hinzu, die sich aus den Eigenschaften ergeben, die auf dem direkten Pfad der SemwidgQL-Abfrage liegen. Werte aus Filterausdrücken werden also nicht zurückgeliefert.

Nach Angles et al. (2017) ist die Komplexität des Auswertungsproblems einer SPARQL-Abfrage, die auf diese Weise aus einer SemwidgQL-Abfragen erzeugt wird, NP-vollständig. Weitere Funktionen von SemwidgQL, welche in Abschnitt 2.3.3 beschrieben werden, können der erzeugen SPARQL-Abfrage zusätzliche Elemente hinzufügen, womit sich die Komplexität des Auswertungsproblems zu PSPACE-vollständig (vgl. Vardi, 1982) reduziert.

Listing 2.8: Eine SemwidgQL-Pfadabfrage. Ausgehend von Albert Einstein, werden dessen Geburtsorte und anschließend deren Labels traversiert. Unterhalb der SemwidgQL-Abfrage befindet sich deren Übersetzung in SPARQL.

```
1  dbr:Albert_Einstein.dbo:birthPlace.rdfs:label
```

⇓

```
2  SELECT DISTINCT
3     (dbr:Albert_Einstein AS ?Albert_Einstein) ?birthPlace ?label
4  WHERE {
5    dbr:Albert_Einstein dbo:birthPlace ?birthPlace .
6    ?birthPlace rdfs:label ?label .
7  }
```

2.3.2 Grundfunktionalität

Nachfolgend werden die grundlegenden Funktionen von SemwidgQL erläutert, die für die erfolgreiche Formulierung von Abfragen essenziell sind. Dies umfasst die einzelnen Bestandteile der Abfrage, die in Abbildung 2.4 enthalten sind.

Resource-Auswahl: In SemwidgQL lassen sich Resources auf drei verschiedene Arten auswählen. Entweder anhand ihrer vollständigen URI, welche von spitzen Klammern umschlossen werden muss, als präfigierte URI oder als benannte Resource. Während die ersten beiden Varianten auch so in SPARQL angewandt werden, sind die benannten Resources ein Alleinstellungsmerkmal von SemwidgQL.

Pfad-Traversierung: SemwidgQL ist eine Pfadabfragesprache mit der es möglich ist, Linked-Data-Graphen zu traversieren. Dabei nutzt SemwidgQL die Punkt-Notation, um die einzelnen Schritte des Pfades von einander abzugrenzen. Damit ähnelt die Syntax eine SemwidgQL-Abfrage der von objektorientierten Programmiersprachen und gibt Anwendern mit Programmiererfahrung einen Hinweis auf die Nutzung der Abfragesprache. Listing 2.8 zeigt ein einfaches Beispiel einer SemwidgQL-Abfrage, in der ausgehend von Albert Einstein, zuerst dessen Geburtsorte und anschließend deren Labels traversiert werden. Darunter befindet sich die Übersetzung dieser SemwidgQL-Abfrage in SPARQL.

Filterung von Eigenschaften: Die Ergebnismenge einer Abfrage kann anhand von Filterausdrücken eingeschränkt werden. Ein Filterausdruck wird dabei einem Pfadelement zugewiesen und auf Eigenschaften angewandt, die mit diesem Pfadelement verknüpft sind. Der Filterausdruck besteht aus drei Teilen. Der verknüpften Eigenschaft, einem Vergleichsoperator und einem Vergleichswert. Mehrere Filterausdrücke können mittels Konjunktion und nicht-ausschließender Disjunktion miteinander kombiniert werden.

Listing 2.9 demonstriert die Nutzung eines Filterausdrucks. Von Albert Einstein ausgehend, werden dessen Geburtsorte abgefragt. Der Filterausdruck schränkt diese auf diejenigen ein, die vom Typ `dbo:Settlement` sind. Bezogen auf die Daten aus dem Beispiel aus Abbildung 2.1, wird also nur die Resource zu Ulm in die Ergebnismenge aufgenommen. Anschließend werden die Labels von Ulm abgefragt.

Listing 2.9: Eine SemwidgQL-Abfrage mit einem Filterausdruck. Die Abfrage fragt alle Geburtsorte von Albert Einstein und deren Labels ab. Die Geburtsorte müssen dem Kriterium des Filterausdrucks genügen. In diesem Beispiel muss der Ort also über die Relation `rdf:type` mit der Resource `dbo:Settlement` verknüpft sein. Unterhalb der SemwidgQL-Abfrage befindet sich deren Übersetzung in SPARQL.

```
1  dbr:Albert_Einstein.dbo:birthPlace(rdf:type = dbo:Settlement).rdfs:label
```

```
2  SELECT DISTINCT
3     (dbr:Albert_Einstein AS ?Albert_Einstein) ?birthPlace ?label
4  WHERE {
5    dbr:Albert_Einstein dbo:birthPlace ?birthPlace .
6    ?birthPlace rdf:type ?type .
7    FILTER (
8      ?type = dbo:Settlement
9    )
10   ?birthPlace rdfs:label ?label .
11 }
```

Listing 2.10: Eine SemwidgQL-Abfrage mit verzweigten Pfaden. Von Ulm werden innerhalb einer Abfrage die Koordinaten, bestehend aus Längen- und Breitengrad, abgerufen. Die hier genutzten Eigenschaften sind kein Teil des Beispiels aus Abbildung 2.1, sollten jedoch selbsterklärend sein. Unterhalb der SemwidgQL-Abfrage befindet sich deren Übersetzung in SPARQL.

```
1  dbr:Ulm.[geo:lat, geo:long]
```

```
2  SELECT DISTINCT
3     (dbr:Ulm AS ?Ulm) ?lat ?long
4  WHERE {
5     dbr:Ulm geo:lat ?lat ;
6              geo:long ?long .
7  }
```

2.3.3 Erweiterte Funktionalität

Neben den grundlegenden Funktionen, enthält SemwidgQL ein Reihe weiterer Funktionen, die die Mächtigkeit der Abfragesprache erhöhen und deren Nutzung erleichtern. Diese werden im Folgenden beschrieben.

Verzweigung von Pfaden: In einigen Fällen kann es nötig sein, mehrere Werte einer Resource gleichzeitig abzufragen. Beispielsweise die Koordinaten eines Ortes, bestehend aus Längen- und Breitengrad, oder bei Messergebnissen der Messwertwert und der Zeitstempel der Messung. Anstelle einer einzelnen Eigenschaft kann eine Liste von Eigenschaften oder auch ganzen Pfaden als letztes Element eines Pfades angegeben werden. Diese Liste wird in eckigen Klammern zusammengefasst und die einzelnen Elemente der Liste durch Kommata getrennt. Listing 2.10 enthält eine SemwidgQL-Abfrage die Längen- und Breitengrad von Ulm abfragt sowie die Übersetzung dieser Abfrage in SPARQL.

Listing 2.11: Eine SemwidgQL-Abfrage mit Umkehrung einer Subjekt-Prädikat-Objekt-Relation. Die Abfrage fragt alle Personen und deren Label ab, die in Ulm geboren wurden. Unterhalb der SemwidgQL-Abfrage befindet sich deren Übersetzung in SPARQL.

```
1  dbr:Ulm.^dbo:birthPlace.rdfs:label
```

⇓

```
2  SELECT DISTINCT
3    (dbr:Ulm AS ?Ulm) ?birthPlaceOf ?label
4  WHERE {
5    ?birthPlaceOf dbo:birthPlace dbr:Ulm .
6    ?birthPlaceOf rdfs:label ?label .
7  }
```

Umkehrung von Subjekt-Prädikat-Objekt-Relationen: Daten im RDF-Format beschreiben einen gerichteten Graphen, in welchem in der Regel nur unidirektionale Verbindungen zwischen zwei Knoten über eine Kante in der Form Subjekt-Prädikat-Objekt existieren. So existiert im hier genutzten Beispiel nur eine gerichtete Verbindung von Albert Einstein zu seiner Geburtsstadt Ulm, jedoch keine Verbindung von Ulm zu Albert Einstein. Damit also nicht nur zu einem gegebenen Subjekt das gewünschte Objekt, sondern auch das gewünschte Subjekt zu einem gegebenen Objekt abgefragt werden kann, ermöglicht SemwidgQL die Umkehrung der abgefragten Relation. Um in einer Abfrage die Umkehrung der Relation anzuweisen, wird der abzufragenden Eigenschaft ein Zirkumflex (»^«) vorangestellt.

In dem Beispiel aus Abbildung 2.1 ist die Resource zu Albert Einstein mit den entsprechenden Resources zu dessen Geburtsorten über eine gerichtete Relation verknüpft. Eine gerichtete Relation in Gegenrichtung ist jedoch nicht vorhanden. Um in einer Pfadabfrage von Ulm zu Albert Einstein navigieren zu können, muss daher die Relation über die Eigenschaft `dbo:birthPlace` invertiert werden. Listing 2.11 enthält eine entsprechende SemwidgQL-Abfrage und deren Übersetzung in SPARQL.

Wildcard-Elemente: Anstelle einer definierten Eigenschaft oder auch Resource kann ein Wildcard-Element genutzt werden. Dieses steht stellvertretend für sämtliche mögliche Eigenschaften oder Resources und kann mittels Filterausdrücken eingeschränkt werden. Wildcard-Elemente sind beispielsweise hilfreich bei der Exploration der Datenbasis, wenn nicht bekannt ist, welche Eigenschaften zu einem Objekt im RDF-Graphen enthalten sind. Ebenso werden sie benötigt, wenn nicht nur Daten für eine bestimmte Resource, sondern alle Resources eines vorgegebenen Typs abgefragt werden sollen.

Listing 2.12 enthält eine SemwidgQL-Abfrage mit zwei Wildcard-Elementen, einmal anstelle der Start-Resource und einmal anstelle einer Eigenschaft. Die Start-Resource wird auf die Objekte eingeschränkt, welche vom Typ `dbo:Settlement` sind. Im hier genutzten Beispiel aus Abbildung 2.1 trifft dies nur auf die Ulm-Resource zu. Anschließend werden alle Werte der Eigenschaften abgefragt, die mit Ulm verknüpft sind. Dies umfasst die beiden Labels (`rdfs:label`) von Ulm in deutscher und englischer Sprache sowie den Typ (`rdf:type`).

Listing 2.12: Eine SemwidgQL-Abfrage mit Wildcard-Elementen. Die Abfrage fragt alle Resources ab, die vom Typ `dbo:Settlement` sind. In dem genutzten Beispiel aus Abbildung 2.1 trifft dies nur auf Ulm zu. Anschließend werden alle Werte der Eigenschaften abgefragt, die mit Ulm verknüpft sind. Hier also die beiden Labels und der Typ. Unterhalb der SemwidgQL-Abfrage befindet sich deren Übersetzung in SPARQL.

```
1  *(rdf:type = dbo:Settlement).*
```

⇓

```
2  SELECT DISTINCT ?wildcardA ?wildcardB
3  WHERE {
4    ?wildcardA ?predicateForWildcardB ?wildcardB .
5    ?wildcardA rdf:type ?type .
6    FILTER (
7      ?type = dbo:Settlement
8    )
9  }
```

Verschachtelte Abfragen: Die Vergleichswerte von Filteroperationen können anhand von verschachtelten Abfragen definiert werden. Der Vergleichswert beziehungsweise die möglicherweise entstehende Liste der Vergleichswerte wird dabei immer durch das letzte Element der verschachtelten Abfrage gebildet. Enthält die verschachtelte Abfrage eine Pfadverzweigung, werden die jeweils letzten Elemente der einzelnen Zweige innerhalb des Filterausdrucks mittels nicht-ausschließender Disjunktion miteinander kombiniert.

Listing 2.13 zeigt eine SemwidgQL-Abfrage, die als Vergleichswert eine verschachtelte Anfrage enthält, welche eine Pfadverzweigung mit zwei Eigenschaften nutzt. In diesem Beispiel werden alle Personen abfragen, die das selbe Geburtsdatum besitzen, wie Albert Einstein. Das Geburtsdatum wird dabei aus der im Beispiel aus Abbildung 2.1 tatsächlich existierenden Eigenschaft `dbo:birthDate` und der hinzugedachten Eigenschaft `ex:dateOfBirth` abgefragt.

Filter-Schlüsselwörter: Die nachfolgenden Filter-Schlüsselwörter erweitern oder vereinfachen die Filterung von Eigenschaften in Filterausdrücken. Filterausdrücke, welche Filter-Schlüsselwörter enthalten können wie ganz normale Filterausdrücke genutzt werden und mittels logischer Konjunktion oder Disjunktion mit anderen Filterausdrücken verknüpft werden. Auf der linken Seite eines Filterausdrucks mit Filter-Schlüsselwort steht immer das Schlüsselwort gefolgt von einem logischen Vergleichsoperator. Je nach Filter-Schlüsselwort ist die Menge der nutzbaren logischen Vergleichsoperatoren eingeschränkt. Auf der rechten Seite steht der Vergleichswert, auf dessen Basis die Filterung erfolgen soll.

`@lang`: Mit diesem Schlüsselwort kann eine Eigenschaft anhand des angegebenen Sprachcodes (z. B. `de`, `de-DE`, `en-US`), welcher als rechter Operand als Zeichenkette formatiert sein muss, gefiltert werden. Erfahrungsgemäß werden Texte in SPARQL-Endpoints, wie zum Beispiel im Fall von DBpedia, nur mit Sprachcodes annotiert, wenn diese auch in verschiedenen Sprachen vorhanden sind. Texte, die nur in einer Sprache vorhanden sind, werden hingegen meist nicht annotiert. Bei Nutzung dieses Schlüsselworts werden daher nicht nur die Texte zurückgeliefert, welche mit dem angegebenen Sprachcode annotiert wurden, sondern auch immer die Texte,

Listing 2.13: Eine SemwidgQL-Abfrage mit verschachtelter Abfrage innerhalb des Filterausdrucks. Die Abfrage fragt alle Personen ab, die ein bestimmtes Geburtsdatum haben. Der Vergleichswert für das Geburtsdatum wird durch einer verschachtelten Abfrage bestimmt, in welcher das Geburtsdatum von Albert Einstein aus der im Beispiel aus Abbildung 2.1 tatsächlich existierenden Eigenschaft `dbo:birthDate` und der hinzugedachten Eigenschaft `ex:dateOfBirth` abgefragt wird. Unterhalb der SemwidgQL-Abfrage befindet sich deren Übersetzung in SPARQL.

```
*(dbo:birthDate = {albert_einstein.[dbo:birthDate, ex:dateOfBirth]})
```

⇓

```
SELECT DISTINCT ?wildcard (dbr:Albert_Einstein AS ?Albert_Einstein)
    ↪ ?birthDate ?dateOfBirth
WHERE {
  ?wildcard dbo:birthDate ?birthDate_0 .
  dbr:Albert_Einstein dbo:birthDate ?birthDate .
  dbr:Albert_Einstein dbo:dateOfBirth ?dateOfBirth .
  FILTER (
    ?birthDate_0 = ?birthDate  ||
    ?birthDate_0 = ?dateOfBirth
  )
}
```

die nicht annotiert wurden. Zwar verringert dieses Verhalten die Präzision, mit der die Filterfunktion arbeitet, vereinfacht jedoch die Anwendbarkeit für Nutzer von SemwidgQL erheblich. Als Vergleichsoperator ist nur der Gleich-Operator (»=«) zugelassen.

`@self`: Anstatt eine Eigenschaft anhand des Wertes einer weiteren Eigenschaft zu filtern, lässt sich mittels dieses Schlüsselworts eine Eigenschaft anhand ihres eigenen Wertes filtern. Es sind alle Vergleichsoperatoren zugelassen, die auch für normale Filterausdrücke genutzt werden können.

`@timestart` / `@timeend`: Diese beiden Schlüsselwörter erlauben es, Eigenschaften zu filtern, die einen zeit- oder datumsbezogenen Wert besitzen. Das Schlüsselwort `@timestart` ermöglicht es, alle Werte herauszufiltern, die zeitlich nach dem angegebenen Zeitpunkt liegen, `@timeend` entsprechend alle Werte, die zeitlich davor liegen. Werden beide Schlüsselwörter kombiniert, lassen sich auch Werte herausfiltern, die außerhalb oder innerhalb eines Zeitintervalls liegen. Der rechte Operand dieses Filterausdrucks ist eine Zeichenkette, die ein absolutes Datum oder einen relativen Zeitpunkt beschreibt. Das absolute Datum ist ein Zeitpunkt, formatiert als `xsd:dateTime`[32], der optional um eine angegebene Zeitspanne verschoben werden kann. Der relative Zeitpunkt wird durch die Zeichenkette `now` bestimmt, die ebenfalls optional um eine angegebene Zeitspanne verschoben werden kann. Diese Zeitspanne kann in Sekunden (`s`), Minuten (`m`), Stunden (`h`), Tagen (`d`), Wochen (`w`) oder einer Kombination aus diesen definiert werden. Als Vergleichsoperator ist nur der Gleich-Operator (»=«) zugelassen.

Listing 2.14 zeigt eine SemwidgQL-Abfrage, in der das Schlüsselwort `@timestart` genutzt wird. Dadurch werden nur noch Werte zurückgeliefert, die innerhalb der

[32] `https://www.w3.org/TR/xmlschema-2/#dateTime`

Listing 2.14: Eine komplexe SemwidgQL-Abfrage, die verschiedene vereinfachende Filterausdrücke und Pseudo-Filterausdrücke nutzt. Die Abfrage fragt alle Temperatur-Messwerte ab, die von einem bestimmten Sensor innerhalb der letzten sieben Tage aufgezeichnet wurden. Aus den Messwerten (diese werden minütlich gespeichert) wird für jede Stunde ein Durchschnittswert berechnet und zurückgeliefert. Unterhalb der SemwidgQL-Abfrage befindet sich deren Übersetzung in SPARQL.

```
*(@type = ic:Measurement && ip:sensor = ir:TH_LF285_01
  && ip:type = 'Temperature').[ip:value(@aggregate = 'AVG'),
  ip:measuredAt(@timeinterval = '60 min' && @timestart = 'now - 7 days'
  && @aggregate = 'MIN')]
```

⇓

```
PREFIX ic: <http://linkeddata.interactivesystems.info/sensor/class#>
PREFIX ip: <http://linkeddata.interactivesystems.info/sensor/property#>
PREFIX ir: <http://linkeddata.interactivesystems.info/sensor/resource#>
SELECT DISTINCT SAMPLE(?wildcard) AS ?wildcard
    AVG(?value) AS ?value MIN(?measuredAt) AS ?measuredAt
WHERE {
  ?wildcard ip:value ?value .
  ?wildcard ip:measuredAt ?measuredAt .
  ?wildcard rdf:type ?typeOf_wildcard .
  ?wildcard ip:sensor ?sensor .
  ?wildcard ip:type ?type .
  FILTER (
    ?typeOf_wildcard = ic:Measurement && ?sensor = ir:TH_LF285_01  &&
    STR(?type) = "Temperature"
  )
  FILTER (
    xsd:dateTime(?measuredAt) >= now() - 604800
  )
  BIND(FLOOR((xsd:dateTime(?measuredAt) -
      xsd:dateTime("1970-01-01T00:00:00")) / (3600)
      ) AS ?measuredAt_timeinterval)
}
GROUP BY ?measuredAt_timeinterval
ORDER BY DESC(?measuredAt_timeinterval)
```

letzten sieben Tage gemessen wurden, basierend auf dem aktuellen Ausführungszeitpunkt der Abfrage.

`@type`: Dieses Schlüsselwort ist äquivalent zu der Eigenschaft `rdf:type`. Als Vergleichsoperatoren sind der Gleich-Operator (»=«) und der Ungleich-Operator (»!=«) zugelassen.

Listing 2.14 zeigt eine SemwidgQL-Abfrage, in der das Schlüsselwort `@type` genutzt wird. Dadurch werden nur noch Werte betrachtet, die vom Typ `ic:Measurement` sind.

Pseudo-Filter-Schlüsselwörter: Die nachfolgenden Pseudo-Filter-Schlüsselwörter lassen sich wie die zuvor beschrieben Filter-Schlüsselwörter verwenden. Während normale Filter-Schlüsselausdrücke tatsächlich dafür sorgen, dass in einer SPARQL-Abfrage ein FILTER-Ausdruck erzeugt wird, wirken sich Ausdrücke, die mittels Pseudo-Filter-Schlüsselwörter

gebildet wurden auf andere Teile der SPARQL-Abfrage aus. Pseudo-Filterausdrücke können sowohl untereinander wie auch mit vereinfachenden Filterausdrücken mittels logischer Konjunktion oder Disjunktion verknüpft werden. Da mittels ihnen kein logischer Vergleich durchgeführt wird, ist definiert, dass sie sich immer neutral verhalten. Bei einer Konjunktion liefert der Ausdruck, der mit einem Pseudo-Filter-Schlüsselwort gebildet also immer »wahr« und bei einer Disjunktion immer »falsch« zurück. Der in einem Pseudo-Filterausdruck genutzte Operator muss immer der Gleich-Operator (»=«) sein. Dieser ist jedoch nicht als Vergleichs- sondern als Zuweisungsoperator zu verstehen, welcher dem Schlüsselwort den Wert des rechten Operanden zuweist.

`@aggregate`: Dieses Schlüsselwort ermöglicht es, eine der SPARQL-Aggregat-Funktion auf den Wert einer Eigenschaft anzuwenden. Diese Funktion wird dem SELECT-Ausdruck der SPARQL-Abfrage hinzugefügt. Erlaubte Werte für den rechten Operanden, als Zeichenkette formatiert, sind `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`, `GROUP_CONCAT` und `SAMPLE`.[33] Sobald eine Aggregat-Funktion in SPARQL auf einen beliebigen Eigenschaftenwert angewandt werden soll, müssen auch alle anderen Eigenschaftenwerte aggregiert werden. Wird kein entsprechender Aggregat-Wert für die anderen Eigenschaftenwerte innerhalb der Abfrage definiert, wird automatisch der Wert SAMPLE genutzt, wodurch ein zufälligen Wert ausgewählt wird. Der angefragte SPARQL-Endpoint muss mindestens SPARQL in der Version 1.1 unterstützen, damit dieses Schlüsselwort genutzt werden kann.

Listing 2.14 zeigt eine SemwidgQL-Abfrage, in der das Schlüsselwort `@aggregate` genutzt wird. Einmal mit dem Wert `AVG` und einmal mit dem Wert `MIN`.

`@hide`: Dieses Schlüsselwort ermöglicht es, zu steuern, ob ein Eigenschaftenwert aus dem SELECT-Ausdruck auszublenden beziehungsweise er nicht in den SELECT-Ausdruck aufzunehmen ist. Um eine Variable auszublenden muss der boolesche Wert `true` als rechter Operand genutzt werden.

`@optional`: Wird in Verbindung mit diesem Schlüsselwort der boolesche Wert `true` als rechter Operand genutzt, wird das Triple-Pattern, zu welchem die Eigenschaft dieses Funktionsausdrucks gehört, in einen OPTIONAL-Ausdruck eingebettet.

`@predicate`: Das Prädikat eines Triple-Patterns ist Teil der SemwidgQL-Abfrage und in der Regel bekannt. Es ist daher standardmäßig kein Teil des zurückgelieferten Ergebnisses. Wird jedoch keine fest definiertes Prädikat, sondern ein Wildcard-Prädikat genutzt, kann mittels dieses Schlüsselworts das Prädikat dem SELECT-Ausdruck der generierten SPARQL-Abfrage hinzugefügt werden. Dazu muss der boolesche Wert `true` als rechter Operand genutzt werden. Dieses Schlüsselwort lässt sich auch mit fest definiertes Prädikaten nutzen, falls dieses beispielsweise für eine Visualisierung der Ergebnisdaten benötigt wird.

`@timeinterval`: Dieses Schlüsselwort kann genutzt werden, um zeitsequentielle Werte zu gruppieren und zu aggregieren. Mittels des rechten Operanden des Pseudo-Filterausdrucks kann ein Sampling-Intervall definiert werden. Alle zurückgelieferten Werte, welche sich im selben Intervall befinden, werden aggregiert. Standardmäßig wird die Aggregat-Funktion `SAMPLE` genutzt. Diese kann jedoch in Kombination mit `@aggregate` beliebig geändert werden. Vergleichbar mit `@timestart` und `@timeend` kann die Länge des Intervalls, in einer Zeichenkette, in Sekunden (`s`), Minuten (`m`),

[33] https://www.w3.org/TR/sparql11-query/#aggregates

Stunden (`h`), Tagen (`d`), Wochen (`w`) oder einer Kombination aus diesen definiert werden.

Listing 2.14 zeigt eine SemwidgQL-Abfrage, in der `@timeinterval` genutzt wird. In dieser Abfrage werden mittels `@timestart` nur die Werte betrachtet, welche innerhalb der letzten sieben Tage, bezogen auf den Ausführungszeitpunkt der Abfrage, aufgezeichnet wurden. Diese Werte werden dann mittels `@timeinterval` in Abschnitte von jeweils 60 Minuten gruppiert und anschließend aggregiert.

2.3.4 Implementierung

Für SemwidgQL wurden zwei Referenzimplementierungen entwickelt. Eine Implementierung in JavaScript und eine in Java.[34] Die JavaScript-Version ist integraler Bestandteil von SemwidgJS, kann aber auch unabhängig davon genutzt werden.

Die Syntax von SemwidgQL wird durch einer Reihe von EBNF-Regeln (vgl. Anhang A) beschrieben. Anhand dieser Regeln wurde mittels des Parser-Generators ANTLR (vgl. Parr & Fisher, 2011) jeweils ein Parser für die beiden oben genannten Programmiersprachen generiert. Die Parser erzeugen aus einer ihnen übermittelten SemwidgQL-Abfrage einen abstrakten Syntaxbaum, aus dem der SemwidgQL-zu-SPARQL-Transcompiler eine SPARQL-Abfrage ableitet. Anhang B beschreibt mittels Pseudocode, wie sich aus dem Syntaxbaum eine SPARQL-Abfrage konstruieren lässt.

Aus der Definition der Anforderungen an SemwidgQL (vgl. Abschnitt 2.2.2) ergab sich die Notwendigkeit zur Rückgabe aller Elemente die sich auf dem traversierten Abfragepfad befinden. Um den Pfad, auch ohne den ursprünglichen Abfragetext zu kennen, rekonstruieren zu können, selbst wenn der Pfad nicht rein linear ist, sondern Verzweigungen enthält, nutzen die Referenzimplementierungen von SemwidgQL ein Benennungssystem für die erzeugten SPARQL-Variablen, welches eine solche Pfadrekonstruktion ermöglicht (vgl. Listing B.25). Dieses Benennungssystem ist kein Bestandteil der Sprachdefinition sondern lediglich ein Teil der jeweiligen Implementierung.

In diesem Benennungssystem erhält jede Variable ein Suffix, welches den Typ des Graphelements enthält (Resource oder Eigenschaft) sowie seine Position im Syntaxbaum, der sich beim Parsen des Pfads auf Basis der EBNF-Regeln ergibt. Der Syntaxbaum ist ein gewurzelter Baum, dessen Kanten von der Wurzel ausgehen (*Out-Tree*). Die genaue Position eines Knotens in diesem Baum lässt sich durch ein einfaches Verfahren beschreiben. Dabei werden alle Kindknoten eines jeden Knotens des Baum beginnend mit 0 (Null) von links nach rechts durchnummeriert. Der Wurzelknoten erhält ebenfalls die Nummer 0. Die Position eines Knotens ist die Sequenz der Zahlen, die sich aus dem direkten Pfad ergibt, der vom Wurzelknoten zum gewünschten Knoten führt.

In beiden Implementierungen existiert eine Funktion (`toSparql`), welche als Parameter die SemwidgQL-Abfrage als Zeichenkette sowie die Listen für benannte Resources und Präfixe entgegen nimmt, und die daraus die entsprechende SPARQL-Abfrage generiert (vgl. Listing B.1).

[34] `https://semwidg.org/page/download-semwidgql`

2.4 Empirische Nutzerstudie zum Vergleich von SemwidgQL und SPARQL

In der nachfolgenden empirischen Nutzerstudie wurden SemwidgQL und SPARQL miteinander verglichen. Zweck der Studie war es, zu untersuchen, wie gut die Ziele, die bei der Entwicklung von SemwidgQL im Vordergrund standen – nämlich Effektivität, Effizienz und leichte Erlernbarkeit (vgl. Abschnitt 2.2.2) – umgesetzt werden konnten. Zusätzlich sollte die Nutzerzufriedenheit untersucht werden. Effektivität, Effizienz und Erlernbarkeit wurden anhand mehrerer Query-Interpretations- und Query-Formulierungsaufgaben gemessen. Die Nutzerzufriedenheit wurde anhand der Ergebnisse eines Fragebogens beurteilt. Zu Beginn dieses Abschnitts werden verwandte Arbeiten vorgestellt. Anschließend werden das Design und der Ablauf der Studie beschrieben und die Ergebnisse der Studie vorgestellt. Abschließend werden die Ergebnisse interpretiert und diskutiert.

> Die nachfolgenden Studienergebnisse und die daraus gezogenen Schlussfolgerungen wurden zum Großteil im Rahmen der »16th International Semantic Web Conference (ISWC 2017)« vorgestellt und in Stegemann & Ziegler (2017a) veröffentlicht.

2.4.1 Verwandte Arbeiten

Im nachfolgenden Abschnitt wird ein Überblick über relevante Arbeiten gegeben, in welchen vergleichende Nutzerstudien im Bereich der Datenbankabfragesprachen beschrieben wurden. Das Hauptaugenmerk richtet sich bei der Untersuchung dieser Arbeiten auf das Design und den Ablauf der jeweiligen Studien. Dieser Studienbeschreibung nachfolgend wurden Arbeiten betrachtet, in denen sich die Autoren mit der Komplexität von Aufgaben und Datenbankabfragen befasst haben. Die darin vorgestellten Modelle und Metriken wurden auf ihre Eignung zur Bestimmung der Komplexität von SPARQL-Abfragen hin analysiert und bewertet.

Nutzerstudien

Erste Nutzerstudien, die verschiedene Datenbankabfragesprachen miteinander bezüglich der Leistung ihrer Anwender verglichen haben, wurden bereits in den 1970er Jahren durchgeführt. Reisner et al. (1975) und Reisner (1977) untersuchten die Gebrauchstauglichkeit und Erlernbarkeit von SQUARE und des SQL-Vorgängers SEQUEL in einer vergleichenden Nutzerstudie. Diese Studie fand im Rahmen einer vorlesungsähnlichen Veranstaltungsreihe statt, während derer die Probanden wöchentliche Tests und eine abschließende Klausur schrieben. Zusätzlich absolvierten sie einen weiteren Test in der darauffolgenden Woche, in dem überprüft werden sollte, welche Inhalte auch nach der bereits durchgeführten notenrelevanten Klausur noch verinnerlicht waren. Die Probanden wurden bezüglich ihrer Programmierfähigkeiten (Programmierer bzw. Nicht-Programmierer) und einer der beiden Abfragesprachen in vier Gruppen unterteilt und getrennt unterrichtet. Der zeitliche Umfang der Vorlesung wurde an die Gruppen angepasst, sodass die Gruppen der Programmierer insgesamt 12 Stunden und die Gruppen der Nicht-Programmierer 14 Stunden lang unterrichtet wurden. Da sich die Veranstaltungsreihe auf einen mehrwöchigen Zeitraum erstreckte, konnte allerdings nicht sichergestellt werden, dass auch alle Probanden im gleichen Umfang daran teilgenommen haben. Durch Verspätungen oder krankheitsbedingtes Fehlern haben einige Probanden nicht die gesamte Zeit an der Veranstaltung teilgenommen. Für die Tests und die Klausur bekamen die Probanden jeweils eine Liste von Formulierungsaufgaben (z. B. »Find the names of all employees who make more

than their managers«) zu denen sie eine Abfrage in der entsprechenden Abfragesprache formulieren sollten. Für die Tests, aber nicht die Klausur, gab es zusätzliche Interpretationsaufgaben, in denen sie zu einer gegebenen Abfrage in einer der Datenbankabfragesprachen die natürlichsprachliche Version ableiten sollten. Die Leistung der Probanden wurde anhand der Korrektheit der bearbeiteten Aufgaben gemessen, wobei zwischen vollständig korrekten Antworten, Antworten mit geringfügigen Fehlen und falschen Antworten unterschieden wurde. Da die Tests und die Klausur ausschließlich handschriftlich erfolgten, konnten neben der Korrektheit der Antworten und den jeweiligen Fehlertypen keine weiteren abhängigen Variablen erhoben werden.

Welty & Stemple (1981) führten zwei vergleichbare Studien für die Abfragesprachen SQL und TABLET in zwei aufeinanderfolgenden Jahren mit unterschiedlichen Probanden durch. Die Studie wurde ebenfalls im Rahmen einer Universitätsveranstaltung durchgeführt. Die Probanden wurden jeweils auf zwei Gruppen aufgeteilt und mussten eine der beiden Sprachen selbstständig mit Hilfe eines Handbuchs erlernen. Es wurden jeweils 14 Übungsstunden abgehalten, in denen die Probanden Fragen stellen konnten und Übungsaufgaben bearbeitet werden sollten. Am Ende des Semesters wurde eine Klausur gestellt, die ausschließlich über die Note der Probanden entschied. Jeweils drei Wochen nach der Klausur wurde eine Wiederholung der Klausur geschrieben. Im ersten Jahr war diese verpflichtend, im zweiten Jahr wurde den Probanden für die Teilnahme eine monetäre Vergütung ausgezahlt. Die Probanden wurden angehalten, für die Wiederholungsklausur nicht zu lernen. Die Leistung der Probanden wurde wieder anhand der Korrektheit der bearbeiteten Aufgaben gemessen. Bezüglich Korrektheit der Aufgaben wurde zwischen korrekten Antworten, Antworten mit geringfügigen Fehlern, Antworten die zwar falsch sind aber von einem guten Compiler korrigiert werden können, falschen Antworten, unvollständigen Antworten und nicht bearbeiteten Aufgaben unterschieden. Zusätzlich wurden die Aufgaben in leichte und schwierige Aufgaben unterteilt. Auch hier konnten neben der Korrektheit der Antworten und den jeweiligen Fehlertypen keine weiteren abhängigen Variablen erhoben werden.

Für die XML-Abfragesprachen XQuery und XIL wurde von Lassila et al. (2015) eine vergleichende Nutzerstudie durchgeführt. XIL wurde von den Autoren der Studie selbst entwickelt. Die Studie fand ebenfalls wieder im Rahen einer Universitätsveranstaltung statt, in welcher den Probanden diesmal jedoch beide Abfragesprachen beigebracht wurden. Der Umfang, in dem die beiden Sprachen gelehrt wurde unterschied sich jedoch deutlich. Während XQuery drei Vorlesungseinheiten gewidmet wurde, wurde XIL nur eine Vorlesungseinheit gewidmet. Zusätzlich wurden die Kenntnisse über beide Sprachen in den regulären Übungseinheiten vertieft. Auch hier fand eine schriftliche Klausur statt. In dieser sollten die Probanden Anfragen in beiden Sprachen formulieren. Es wurde zwar für jeden Probanden die gesamte Bearbeitungszeit festgehalten, doch konnte nachträglich nicht bestimmt werden, wie lange jeder Proband für die Bearbeitung der einzelnen Aufgaben und Sprachen aufgewandt hat. Auswertbar war also nur die Korrektheit der Antworten und der entsprechende Fehlertyp.

Lochovsky & Tsichritzis (1977) haben verschiedene Abfragesprachen verglichen denen jeweils unterschiedliche Datenmodelle zugrunde lagen. Das von ihnen entwickelte und untersuchte Educational Data Base System (EDBS) unterstützte hierarchische, netzwerkbasierte und relationale Datenmodelle. Den Probanden wurde jeweils ein Modell und die dazugehörige Abfragesprache beigebracht. Anschließend mussten sie eine Reihe von Coding-Aufgaben bearbeiten, bei denen sie ein lauffähiges Programm entwickeln sollten, welches das Datenbanksystem nutzt. Ebenso mussten sei eine Reihe von Debugging-Aufgaben bearbeiten, bei denen sie Fehler in einem vordefinierten Programm finden und beseitigen sollten. Die Bearbeitung fand

am Computer statt, sodass die Bearbeitungszeiten mitgeloggt werden konnten. Entsprechend konnte nicht nur die Korrektheit der Antworten, sondern auch die Bearbeitungszeiten ausgewertet werden.

Eine umfassendere Nutzerstudie wurde von Chan et al. (1997) durchgeführt. Insgesamt wurden vier Abfragesprachen untersucht, die sich jeweils in ihrem zugrundeliegenden Datenmodell (relationales Modell oder Entity-Relationship-Modell) oder ihrem Eingabeart (textuelle Eingabe oder grafische Eingabe) von einander unterschieden. Untersucht wurden die Sprachen SQL (relationales Modell, textuelle Eingabe), QBE (relationales Modell, grafische Eingabe), KQL (Entity-Relationship-Modell, grafische Eingabe) und VKQL (Entity-Relationship-Modell, grafische Eingabe). Die Probanden, welche wieder aus dem studentischen Umfeld stammen, wurden randomisiert auf vier Gruppen aufgeteilt. Jeder Gruppe wurde zu Beginn eine der vier Abfragesprachen beigebracht. Die Zeit dieser initialen Trainingseinheit variierte zwischen etwa einer Stunde (QBE, KQL, VKQL) und eineinhalb Stunden (SQL). Anschließend fand eine praktische Übungseinheit statt, in der sich die Probanden mit dem Testsystem vertraut machen sollten. In einem finalen Test mussten die Probanden eine Reihe von Abfragen formulieren. Diese Abfragen wurden direkt in das Testsystem eingegeben, welches allerdings mit keiner Datenbank verbunden war. Die Probanden bekamen also keine Antwort zu ihrer Abfrage. Nach jeder Aufgabe bewerteten die Nutzer anhand einer sechsstufigen Skala, wie sicher sie sich waren, dass ihre Aufgabe korrekt ist. Die Bearbeitungszeit wurde automatisch vom Testsystem aufgezeichnet. Die Korrektheit der Aufgaben wurde von den Studienverantwortlichen anhand sechsstufigen Skala bewertet. Insgesamt wurden in dieser Studie also die drei abhängigen Variablen Korrektheit der Antworten, Bearbeitungszeit und das Vertrauen in die eigene Antwort erfasst und ausgewertet.

Topi et al. (2005) haben untersucht, welchen Einfluss die Komplexität der Aufgaben und Zeitdruck auf die Leistung der Probanden bei der Erstellung von Datenbankabfragen haben. Hierbei wurde jedoch kein Vergleich zwischen zwei Abfragesprachen angestellt, sondern nur eine einzelne Abfragesprache untersucht. Die Studie nutzte ein 2×3-faktorielles Design mit zwei Between-Subject-Faktoren. Die Komplexität wurde anhand von zwei Faktoren (einfach und komplex) und die verfügbare Zeit anhand von drei Faktoren (wenig, mittel und viel) untersucht. Die Komplexität der Aufgaben beziehungsweise der Abfragen wurde subjektiv von vier Experten bestimmt. Diese bewerteten eine Menge von Abfragen anhand einer siebenstufigen Skala. Das Drittel der Abfragen mit den niedrigsten Werten wurde als einfach eingestuft und das Drittel mit den höchsten Werten wurde als komplex eingestuft. Die restlichen Abfragen wurden verworfen. An der Studie nahmen 120 Studenten als Probanden teil, welche randomisiert auf sechs Gruppen aufgeteilt wurden. Die Probanden hatten keine Vorerfahrungen in den Bereichen der Programmierung und Datenbanken. Zu Beginn der Studie bekamen die Probanden ein Übungshandbuch, welches sie eigenständig in 30 bis 75 Minuten verinnerlichen sollten. Ein bis zwei Wochen später mussten sie sechs Aufgaben bestimmter Komplexität in einer vorgegebenen Zeit erstellen. Dabei durften sie das Übungshandbuch nutzen und konnten die Abfragen an die Datenbank stellen. Das Feedback konnten sie nutzen um ihre Anfragen gegebenenfalls anzupassen. Während der Studie hat eine Uhr auf dem Computermonitor die noch vorhandene Zeit angezeigt. Ausgewertet wurden die gesamte Anzahl von Abfragen, welche pro Minute an die Datenbank gestellt wurde, die Korrektheit der Aufgaben und die Anzahl der korrekt bearbeiteten Aufgaben pro Minute. Ebenso wurde das von den Probanden angegebene Vertrauen in die eigene Antwort ausgewertet.

Da der Aufwand für Nutzerstudien, die verschiedene Abfragesprachen miteinander vergleichen, sehr groß ist, werden sie nur selten durchgeführt. Die Probanden benötigen eine ausführ-

Tabelle 2.1: Ausgewertete Variablen innerhalb der untersuchten Nutzerstudien.

	Korrektheit	Zeit	Vertrauen
Reisner et al. (1975)	5 Kategorien	–	–
Welty & Stemple (1981)	9 Kategorien	–	–
Lassila et al. (2015)	9 Kategorien	–	–
Lochovsky & Tsichritzis (1977)	richtig / falsch	Zeit pro Aufgabe	–
Chan et al. (1997)	6-stufig	Zeit pro Aufgabe	6-stufig
Topi et al. (2005)	richtig / falsch	korrekte Antworten pro Minute & Abfragen pro Minute	7-stufig

liche Einführung in die Abfragesprachen, um die gestellten Aufgaben in einem zufriedenstellenden Ausmaß ausführen zu können. Häufig finden diese Nutzerstudien daher im Rahmen einer Vorlesung oder vergleichbaren Veranstaltung statt. Von den hier vorgestellten Studien wurde nur die Studie von Topi et al. (2005) außerhalb eines solchen Rahmens durchgeführt. Da einige der Studien auf den Ergebnissen von Klausuren bestehen, konnten dort nur die Korrektheit der gegebenen Antworten bewertet werden. Reisner et al. (1975), Welty & Stemple (1981) und Lassila et al. (2015) unterteilen die Korrektheit der Antworten grob in die Kategorien »korrekt«, »geringfügigen Fehler« und »schwere Fehler« auf. Die beiden letzten Kategorien werden dann noch weiter um den genauen Fehlertyp erweitert. Lochovsky & Tsichritzis (1977) und Topi et al. (2005) unterscheiden lediglich zwischen richtigen und falschen Antworten, während Chan et al. (1997) die Korrektheit subjektiv anhand einer sechsstufigen Skala bewerten. Die genaue Zeit für die Bearbeitung der einzelnen Teilaufgaben konnte nur in den Studien erhoben werden, welche direkt am Computer durchgeführt wurden. Hier wurde entweder die Zeit für die Bearbeitung pro Aufgabe oder, im Fall der begrenzten Bearbeitungszeit, die Anzahl der korrekten Antworten beziehungsweise der Abfragen pro Minute gemessen. In wenigen Studien wurden die Probanden zusätzlich danach befragt, wie sicher sie sich waren, dass ihr gegebene Antwort korrekt ist. Ein Überblick über die untersuchten Variablen wird in Tabelle 2.1 gegeben.

In den meisten Fällen mussten die Probanden die Abfragen ohne ein Feedback der Datenbank erstellen. Entweder, weil die Abfragen analog erstellt wurden oder weil das Programm, mit dem die Abfrage erstellt wurde, mit keiner Datenbank verbunden war. Die Studienergebnisse lassen sich daher nur bedingt auf die reale Nutzung von Datenbankabfragesprachen abbilden.

Auch nach intensiver Recherche konnte nicht eine Publikationen gefunden werden, die eine Nutzerstudie beschreibt, in der SPARQL und eine weitere Abfragesprache aus dem Linked-Data-Bereich miteinander verglichen wurden.

Komplexität von Aufgaben und Abfragen

Sowohl Reisner (1981) wie auch Welty & Stemple (1981) haben bereits Vorüberlegungen gemacht, inwiefern sich die Komplexität der Aufgaben auf die Leistung der Probanden bei der Nutzung unterschiedlicher Abfragesprachen auswirkt. In den zuvor vorgestellten Nutzerstudien wurde in wenigen Fällen zwischen einfachen und komplexen Aufgaben unterschieden. Wie groß der Unterschied bezüglich der Komplexität jedoch war, wurde meist aber nicht bestimmt. Lediglich Lassila et al. (2015) untersuchten ihre Studienergebnisse in Relation zum Komplexitätsmaß nach Halstead (1977). Nachfolgend werden einige Komplexitätsmetriken aus den Bereichen der Linguistik und Softwareentwicklung vorgestellt und hinsichtlich ihrer Eignung analysiert.

Ein allgemeines theoretisches Modell für die Bestimmung der Komplexität von Aufgaben wurde von Wood (1986) aufgestellt. Nach diesem Modell bestehen Aufgaben aus drei wesentlichen Komponenten: *Produkten*, *Handlungen* und *Informationssignalen* (»*information cues*«). Produkte sind die Ergebnisse, die sich aus den Handlungen der Person ergeben, die eine Aufgabe bearbeitet. Informationssignale sind Fakten, die von dieser Person benötigt werden, um Entscheidungen bei ihren Handlungen zu treffen. Die Komplexität einer Aufgabe ergibt sich aus der gewichteten Summe der Teilkomplexitäten für die *Komplexität der Komponenten*, die *koordinative Komplexität* und die *dynamische Komplexität*. Die Komplexität der Komponenten einer Aufgabe ergibt sich direkt aus der Anzahl der unterschiedlichen Handlungen und der Informationssignale, die für die Bearbeitung dieser Handlungen benötigt werden. Die koordinative Komplexität hängt von der Form und Stärke der Beziehungen zwischen Produkten, Handlungen und Informationssignalen ab. Unter anderem fließen aber auch die Abfolge, das Timing und die Position von Handlungen oder Informationssignalen mit ein. Die dynamische Komplexität erfasst Änderungen im Zustand des Gesamtsystems, die sich im Laufe der Aufgabenbearbeitung ergeben, und Auswirkungen auf die Beziehung zwischen Produkten, Handlungen und Informationssignalen hat. Auch wenn der Autor des Modells Formeln für die Bestimmung der Aufgabenkomplexität vorstellt, lassen sich diese nur unzureichend auf den Bereich der Erstellung von Datenbankabfragen abbilden. Zudem fehlen konkrete Erfahrungswerte, die für die Bestimmung verschiedener Variablen in diesen Formeln benötigt würden.

In der Linguistik wurden bereits zahlreiche aufgabenbasierte Nutzerstudien durchgeführt, welche die Lern- und Lehrleistung bei der Erlernung einer Zweitsprache überprüfen sollen. Hierbei wurden den Probanden Aufgaben unterschiedlicher Komplexität gestellt. Brünken et al. (2003) unterteilen die Techniken zur Messung der kognitiven Belastung und damit indirekt der Komplexität der Aufgaben in vier Kategorien: *indirekt-subjektiv* (z. B. Selbsteinschätzung des investierten mentalen Aufwand der Probanden), *direkt-subjektiv* (z. B. Selbsteinschätzung des Stressniveaus), *indirekt-objektiv* (z. B. Messung der Bearbeitungszeit, Messung der Herzrate) und *direkt-objektiv* (z. B. Messung der Gehirnaktivität). Sasayama (2016) hat jedoch gezeigt, dass Probanden die Komplexität von Aufgaben nicht zwingend so einschätzen, wie es die Autoren einer Studie tun. Ist der durch die Autoren der Studie erwartete Unterschied hinsichtlich der kognitiven Komplexität zur Lösung zweier Aufgaben verhältnismäßig hoch, so werden diese auch von den Probanden als stark unterschiedlich eingeschätzt. Ist der erwartete Unterschied der Komplexität jedoch eher gering, fanden sich in den Bewertungen durch die Probanden keine statistisch signifikanten Unterschiede. Indirekt-objektive Verhaltensmessungen wie die Bearbeitungszeit einer Aufgabe haben den Nachteil, dass sie keinen direkten Aufschluss über die kognitive Arbeit des Probanden liefern und von weiteren Moderatorvariablen beeinflusst sein könnten. Da die Einschätzungen der Probanden nicht verlässlich sind, und der Aufwand für direkt- oder indirekt-objektive physikalische Messungen, wie Herzrate oder Gehirnaktivität, extrem hoch sind, sind andere Metriken zur objektiven Bestimmung der Komplexität von Aufgaben erstrebenswert. Zudem hängen die Messwerte stark von den jeweiligen Probanden ab. Ein objektiver Komplexitätswert, mittels dessen sich unter Umständen sogar eine Prognose für die Performanz der Probanden vor der Studie aufstellen lässt, kann dadurch nicht ermittelt werden.

Wang (1970) untersuchte in einer Studie, wie sich die syntaktische Komplexität verschiedener Sätze auf ihre Verständlichkeit auswirkte. Dabei untersuchte er acht Komplexitätsmetriken. Fünf dieser Komplexitätsmetriken basieren auf der Anzahl der Knoten der Syntaxbäume, welche sich aus den Sätzen ableiten lassen. Hierzu gehören unter anderem die von Miller & Chomsky (1963) vorgestellten Metriken *Node/Terminal-Node Ratio* (NTNR) und *Total Number of Nodes*

(TNN). Yngve (1960) stellte die These auf, dass sich die Komplexität eines Satzes anhand der maximalen Tiefe (»*Max. Depth*«) des Syntaxbaums bestimmen ließe, wobei der linke Teil einer Verzweigung als tiefer zu betrachten ist, als der rechte. Die von Martin & Roberts (1966) vorgestellte Metrik verwendet die durchschnittliche Tiefe (»*Mean Depth*«) nach Yngves Berechnung. Eine Metrik, welche nicht auf der Anzahl der Knoten basiert, sondern auf der syntaktisch rekursiven Struktur eines Satzes (self-embedded), ist die *S-Emb*-Metrik. Die Studie zeigte, dass die Maße von Mean Depth, S-Emb und TNN stark mit der Verständlichkeit korrelieren. Zwischen dem Komplexitätsmaß NTNR und der Verständlichkeit eines Satzes konnte hingegen keine statistisch signifikante Korrelation nachgewiesen werden.

In der Softwareentwicklung wurde ein Reihe verschiedener Metriken zur Bestimmung der Komplexität von Algorithmen und Programmen entwickelt. Die zyklomatische Komplexität nach McCabe (1976) definiert die Komplexität eines Computerprogramms anhand der linear unabhängigen Abläufe, die sich bei der Verwendung bedingter Anweisungen ergeben. Da im Bereich der Datenbankabfragen solche Verzweigungen nur in Sonderfällen auftreten, ist diese Metrik für den hier erforderlichen Einsatzzweck ungeeignet. Halstead (1977) definiert die Komplexität eine Computerprogramms beziehungsweise die Schwierigkeit es zu schreiben anhand der Anzahl der genutzten distinkten Operatoren (η_1) und Operanden (η_2) und der Anzahl der insgesamt genutzten Operanden (N_2). Der Halstead-Schwierigkeitsgrad D wird berechnet durch:

$$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$$

Diese Metrik wurde bereits genutzt, um SPARQL-Abfragen (z. B. Leinberger et al., 2014) oder Abfragen in anderen Sprachen (z. B. Casterella & Vijayasarathy, 2013; Lassila et al., 2015) hinsichtlich ihrer Komplexität zu bewerten. Im Fall der SPARQL-Abfragen zählen zu den Operatoren alle Schlüsselwörter, Gruppierungszeichen, mathematische Operatoren und andere Terminalsymbole der Abfragesprache. Die Operanden sind alle Variablen, Resources und Literale. Da einzelne FILTER-Funktionen die Anzahl der distinkten Operatoren stark, die Anzahl der Operanden jedoch nur leicht erhöhen, wird aus der Formel zur Berechnung von Halsteads D schnell ersichtlich, dass diese FILTER-Funktionen einen überproportional starken Einfluss auf den errechneten Wert nehmen, was intuitiv der Korrektheitsannahme dieses Ansatzes widerspricht. Darüber hinaus wird Halsteads gesamter Ansatz auf verschiedenen Ebenen kritisiert (vgl. Shepperd & Ince, 1994).

2.4.2 Komplexitätsbestimmung von Datenbankabfragen im Linked-Data-Bereich

Da die im vorherigen Abschnitt 2.4.1 vorgestellten Komplexitätsmetriken nur unzureichende Ergebnisse für Abfragen im Linked-Data-Bereich liefern oder nicht direkt auf den hier untersuchten Anwendungsfall angewandt werden können, soll hier eine weitere Metrik abgeleitet werden, welche den Anforderungen besser entspricht. Diese soll sich an den mentalen Prozesse orientieren, die eine Person bei der Erstellung einer Abfrage von Daten im RDF-Format verarbeiten muss. Eine fein gegliederte Aufschlüsselung aller Teilschritte die bei der Formulierung einer SPARQL-Abfrage durchlaufen werden müssen, kann gemäß verschiedener Notation erzeugt werden. Die hier vorgestellte Metrik basiert auf der Notation in »SPARQL Syntax Expressions« (SSE)[35].

[35] https://jena.apache.org/documentation/notes/sse.html

Der Syntaxbaum S einer SPARQL-Abfrage wird gemäß der SSE-Notation erstellt. Die Funktion $c(S)$ berechnet dessen Komplexität mittels der Summe der Gewichtungen w eines jeden Knotens n.

$$c(S) = \sum_{n \in S} w(n)$$

Die Gewichtung eines Knotens n wird anhand des Knotentyps $t(n)$ und, gegebenenfalls rekursiv, des Typen des Elternknotens $p(t(n))$ bestimmt, um kontextabhängige Gegebenheiten einfließen lassen zu können. Soll beispielsweise eine Zeichenkette bezüglich ihrer annotierten Sprache gefiltert werden, ist in SPARQL dafür die Funktion `langMatches` vorgesehen, welcher in der Regel als erstes Argument das Ergebnis der Funktion `lang` übergeben wird. Wird die erste Funktion eingesetzt wird also auch in den meisten Fällen die zweite genutzt, sodass diese Kombination als zusammenhängender Ausdruck betrachtet werden kann[36]. Es scheint daher nicht gerechtfertigt zu sein, hierfür den doppelten mentalen Aufwand zu berechnen. Ebenso existieren Knotentypen, wie der Basic-Graph-Pattern-Typ, welcher lediglich dazu dient eine Menge von Triple-Pattern zu gruppieren und voraussichtlich keinen mentalen Aufwand bei der Formulierung einer Abfragen erzeugt. Für alle anderen Knotentypen wird eine Gewichtung von 1 angenommen.

$$w(n) = \begin{cases} 0, & t(n) \mapsto \text{LANG} \wedge t(p(n)) \mapsto \text{LANGMATCHES} \\ 0, & t(n) \mapsto \text{BGP} \\ 1, & \text{sonst} \end{cases}$$

Im Rahmen der Auswertung der Evaluationsergebnisse, die im späteren Abschnitt 2.4.4 dargelegt werden, zeigte sich, dass die hier vorgestellte Komplexitätsmetrik sehr zufriedenstellende Ergebnisse liefert und diese erheblich besser zu den erzielten Messergebnissen passen, als die der Halstead-Metrik. Zudem passen diese nicht nur zu den Ergebnissen der erstellten SPARQL-Abfragen, sondern auch zu denen der SemwidgQL-Abfragen, obwohl diese einen deutlich verschiedenen Aufbau besitzen. Die Metrik scheint daher sprachunabhängig zu sein und die mentalen Prozesse bei der Arbeit mit graphbasierten Daten widerzuspiegeln.

Für die Zukunft lässt diese Metrik auch genauere Gewichtungen als »1« für die verschiedenen Knotentypen zu, wodurch sich noch präzisere Ergebnisse erzielen lassen könnten. Zur genaueren Bestimmung bedarf es jedoch einer umfangreicheren Datenbasis über eine umfassende Menge von verschiedenen Abfragen.

Betrachtet man die in Abschnitt 2.4.1 von Wang (1970) untersuchten Komplexitätsmetriken, ist erkennbar, dass die Halstead-Metrik stark der NTNR-Metrik ähnelt, welche in der Studie sehr schlechte Ergebnisse erzeugt hat. Die SSE-basierte Metrik ähnelt eher der TNN-Metrik, welche vergleichsweise gute Ergebnisse erzeugt hat.

2.4.3 Methodik

Im folgenden Unterabschnitt werden das Studiendesign, die Stichprobe, der Ablauf der Studie und die Erfassung der Versuchsdaten beschrieben.

[36] In 83 % der Fälle, in denen in den Abfragen des LSQ-Datensatzes (vgl. Saleem et al., 2015) die Funktion `lang` genutzt wurde, wurde auch die Funktion `langMatches` genutzt. Umgekehrt wurde in 100 % der Fälle, in denen die Funktion `langMatches` genutzt wurde, auch die Funktion `lang` genutzt.

Studiendesign

Die hier beschriebene Nutzerstudie folgt dem Mixed-Methods-Ansatz mit Messwiederholung. Es wurden objektive Messungen der Performanz der Probanden und subjektive Antworten der Probanden aus einem Fragebogen betrachtet. Für die Messung der Performanz mussten die Probanden mehrere Query-Interpretations- und Query-Formulierungsaufgaben bearbeiten.

Die Effektivität wurde anhand der Korrektheit der Antworten bei den Query-Interpretations- und Query-Formulierungsaufgaben gemessen. Die Effizienz wurde anhand der Performanz der Probanden bei den Query-Formulierungsaufgaben gemessen. Hierbei wurden neun abhängige Variablen untersucht:

a) Bearbeitungszeit: die Zeit, die ein Proband für die Bearbeitung einer Aufgabe benötigt hat.

b) Anzahl der Tastenanschläge: die Anzahl der Tastenanschläge, die ein Proband ausgeführt hat, inklusive aller Löschungen und Ersetzungen von Zeichen.

c) Anzahl der Korrekturen: die Anzahl der Aktionen die eine Korrektur des bisher geschriebenen Antworttextes ausgelöst haben. Darunter fallen unter anderem die Nutzung der Rücklöschtaste, der Entfernen-Taste oder das Markieren mehrerer Zeichen und anschließende Einfügen von anderen Zeichen.

d) Anzahl der zusammenhängenden Korrekturen: werden mehrere Korrekturaktionen aufeinanderfolgend ausgeführt, werden sie als zusammenhängende Korrektur betrachtet (beispielsweise das mehrmalige drücken der Entfernen-Taste). Das Einfügen eines reguläres Zeichens beendet eine zusammenhänge Korrektur.

e) Anzahl der Pausen: die Anzahl der Pausen, die ein Proband während der Bearbeitung einer Aufgabe genommen hat. Eine Pause beginnt, wenn ein Proband zwei Sekunden lang keine Aktion durchgeführt hat. Eine Pause könnte ein Indikator dafür sein, dass ein Proband etwas Nachdenkzeit benötigt, um weitere Aktionen zu planen, die für die Lösung der Aufgabe nötig sind.

f) Dauer der Pausen: die kumulierte Zeit der Pausen in Sekunden die ein Proband während der Bearbeitung einer Aufgabe genommen hat. Der Wert operationalisiert die Nachdenkzeit.

g) Anzahl der Abfragen: die Anzahl der Abfragen, die ein Proband an den SPARQL-Endpoint gestellt hat. Die Probanden hatten während der Studie die Möglichkeit, ihre Abfragen an einem SPARQL-Endpoint zu testen.

h) Anteil der fehlerhaften Abfragen: der Anteil der Abfragen, die der SPARQL-Endpoint nicht bearbeiten konnte, da es sich nicht um syntaktisch korrekte Abfragen handelte.

i) Anzeigedauer der Musterlösung: die Zeit in Sekunden, die sich ein Proband nach dem Bearbeiten einer Aufgabe die dazugehörige Musterlösung angeschaut hat. Eine hohe Anzeigedauer könnte ein Indikator dafür sein, dass sich ein Proband unsicher bezüglich seiner eigenen Antwort ist, und diese daher besonders gründlich mit der Musterlösung abgleicht.

Um die Erlernbarkeit zu evaluieren, wurden die Ergebnisse der Abfrage-Formulierungsaufgaben der initialen Messung und der Messwiederholung bezüglich der oben genannten neun abhängigen Variablen miteinander verglichen.

Die Präferenzen der Probanden bezüglich der Sprachen wurde durch die Antworten eines Fragebogens ermittelt. Für den Fragebogen sollten die Probanden sechs Charakteristiken von

SPARQL und SemwidgQL anhand einer äquidistanten fünfstufigen numerischen Rating-Skala bewerten. Dabei wurden sich jeweils die maximal negativen und maximal positiven Ausprägungen einer Charakteristik gegenübergestellt. Der Minimalwert der Rating-Skala wurde immer mit der negativen, der Maximalwert immer mit der positiven Ausprägung belegt. Die Bewertungen bezogen sich auf die subjektiven Einschätzungen der Erlernbarkeit, der Intuitivität, dem logischen Aufbau, der Verständlichkeit, dem Schreibaufwand und der Eleganz der beiden Sprachen. Zusätzlich wurden die Probanden nach der von ihnen persönlich bevorzugten Sprache befragt.

Beschreibung der Stichprobe

An der Studie, welche im Rahmen der Einführungsveranstaltung stattfand, nahmen sieben Studenten (6 männlich, 1 weiblich) aus den Master-Studiengängen Angewandte Kognitions- und Medienwissenschaft (6) und Angewandte Informatik (1) an der Universität Duisburg-Essen teil. Das Alter der Probanden lag zwischen 23 und 28 Jahren ($M = 25{,}57$; $SD = 2{,}07$). Alle Probanden haben die Vorlesung Grundlegende Programmiertechniken (oder eine vergleichbare Vorlesung) gehört. Die Vorlesung Fortgeschrittene Programmiertechniken wurde von 4 Probanden und die Vorlesung Datenbanken von 3 Probanden gehört. Drei Probanden hatte keine Vorerfahrung im Bereich Linked Data und Semantic Web. Aus unterschiedlichen Vorlesungen hatten 3 Probanden bereits Vorerfahrungen zu diesen Themen gesammelt und ein Proband hat bereits im Rahmen seiner Bachelorarbeit mit RDF und SPARQL gearbeitet.

Ablauf

Die Studie fand im Rahmen der Einführungsveranstaltung eines Seminars zum Thema »Techniken und Anwendungen des Semantic Web« statt. Den Probanden wurden zu Beginn des Seminars ein dreiseitiges Handout ausgeteilt, das einen RDF-Graphen enthält, der für sämtliche Beispiele und Aufgaben des Vortrags und der Evaluation genutzt wurde (Abbildung C.1), sowie eine Übersicht über alle relevanten SPARQL- und SemwidgQL-Befehle (Abbildung C.2 und C.3). Der erste Teil der Einführungsveranstaltung bestand aus einem etwa dreistündigen Vortrag, der in drei Abschnitte aufgeteilt war. Zwischen jedem der Abschnitte fand eine kurze Pause statt. Im ersten Abschnitt wurden den Probanden die grundlegenden Überlegungen, Techniken und Formate vermittelt, auf denen Linked Data und das Semantic Web aufbauen. Dazu gehörten unter anderem die Unterscheidung von Syntax und Semantik, der Unterschied zwischen dem klassischen dokumentenorientierten Web und dem Semantic Web, RDF, alternative Formate, RDF Schema und Ontologien.

Im zweiten Abschnitt wurde den Probanden eine Einführung in SPARQL gegeben. Nachdem den Probanden der Aufbau von Abfragen, die Funktionsweise von Triple-Pattern-Matching, die Verknüpfung mehrerer Triple-Patterns und die Nutzung von Filterfunktionen erklärt wurde, sollten sie vier einfache Aufgaben selbstständig lösen. Dazu sollten sie auf eine vorgefertigte Website zurückgreifen, um dort ihre SPARQL-Abfragen zu formulieren (vgl. C.4). Die Webseite enthielt zwei gegenübergestellte Bereiche für die Eingabe der SPARQL-Abfragen und die Ausgabe der Ergebnisse. Der zu nutzende SPARQL-Endpoint sowie die benötigten Präfixe waren vordefiniert. Die Aufgaben waren:

- »Frage das Label von Essen ab«
- »Frage die Einwohner aller Stadtbezirke von Essen ab«

- »Frage alle Essener Stadtbezirke mit 50 000 bis 60 000 Einwohnern ab«
- »Frage alle Großstädte des Ruhrgebiets ab«

Nach der Absolvierung der Aufgaben wurde den Probanden unter anderem die Verwendung von Literalen in SPARQL, die Eliminierung von Duplikaten mittels des DISTINCT-Schlüsselworts sowie die Sortierung, Gruppierung, Aggregation und Vereinigung von Ergebnismengen erläutert. Ebenso wurde auf die Anwendbarkeit von zeitbezogenen Funktionen eingegangen.

Im dritten Abschnitt wurde den Probanden eine Einführung in SemwidgQL gegeben. Um eine unerwünschte Beeinflussung der Probanden zu verhindern, wurde ihnen hierbei verschwiegen, dass es sich bei SemwidgQL um eine Eigenentwicklung handelt. Der Ablauf entsprach, soweit es möglich war, dem Ablauf des vorherigen Abschnitts. Es wurde höchste Sorgfalt darauf gelegt, beide Sprachen in einem vergleichbaren Umfang zu erläutern. Zu Beginn wurde der Aufbau von Pfadabfragen mittels SemwidgQL beschrieben sowie die Verzweigung von Pfaden. Anschließend wurde auf die Nutzung von *Wildcards*, die Invertierung von Pfadabschnitten und die Verwendung von einfachen Filterausdrücken eingegangen. Daraufhin sollten die Probanden, wie schon beim Vortrag bezüglich SPARQL, selbstständig vier Aufgaben lösen. Die Aufgaben hatten den selben Wortlaut und eine Webseite mit vordefinierten Einstellungen für die Formulierung der SemwidQL-Abfragen wurde ebenfalls zur Verfügung gestellt. Nach Absolvierung der Aufgaben wurde den Probanden unter anderem die Verwendung von *vereinfachenden Filterausdrücken* und *Pseudo-Filterausdrücken* vermittelt, welche auch die Filterung von zeitbezogenen Informationen ermöglicht.

Zwischen dem Vortrag und der Studie fand eine 15-minütige Pause statt, um den starken Unterschied zwischen den zeitlichen Abständen des Vortrags über SPARQL und der Evaluation sowie des Vortrags über SemwidgQL und der Evaluation etwas auszugleichen.

Für die Evaluierung wurden die Probanden randomisiert in zwei gleichgroße Gruppen aufgeteilt. Daraufhin mussten die Probanden insgesamt zwölf Aufgaben bearbeiten. In den ersten drei Aufgaben mussten sie abwechselnd vorgegebene SPARQL- oder SemwidgQL-Abfragen interpretieren. Die Abfragesprache, mit der sie beginnen sollten, war abhängig von ihrer Gruppe festgelegt. Die Webseite für die Evaluation hatte den gleichen Aufbau wie die Webseite zur Eingabe von SPARQL- und SemwidgQL-Abfragen während des Vortrags (vgl. Abbildung C.5). In den nachfolgenden neun Aufgaben mussten die Probanden vorgegebene Informationen mittels SPARQL und SemwidgQL abfragen. Jede Aufgabe musste mit beiden Abfragesprachen bearbeitet werden. Die Reihenfolge der Abfragesprachen wechselte mit jeder Aufgabe. Eine Übersicht über alle Aufgaben und ihre Musterlösungen befindet sich in Anhang C (vgl. Tabelle C.1 – Tabelle C.12). Den Probanden wurde kein Zeitlimit für die Bearbeitung der Aufgaben gesetzt. Sie konnten jederzeit Aufgaben abbrechen und zur nächsten Aufgabe übergehen. Nach jeder Aufgabe wurde ihnen zudem eine entsprechende Musterlösung präsentiert (vgl. Abbildung C.6).

Um zu gewährleisteten, dass die Abfragen, die im Rahmen der Aufgaben von den Probanden erstellt werden sollten, in ihrer Komplexität vergleichbar sind, mit in der Praxis genutzten Abfragen, wurden diese mit dem LSQ-Datensatz (z. B. Saleem et al., 2015) verglichen. Dazu wurde der Komplexitätswert mittel der SSE-basierten Metrik für die Abfragen des Datensatzes und der Studienaufgaben berechnet und miteinander verglichen. 90 % der Abfragen des LSQ-Datensatzes sind danach ähnlich komplex wie die Abfragen, die von den Probanden erstellt werden sollten.

Anschließend füllten die Probanden einen Fragebogen aus, in dem sie Alter, Geschlecht, Studiengang, Fachsemester, bereits besuchte Vorlesungen und zuvor gemachte Erfahrungen

mit Techniken aus dem Bereich Semantic Web und Linked Data angeben sollten. Ebenso sollten eine subjektive Selbsteinschätzung zu der jeweiligen Höhe der Aufmerksamkeit während des Vortrags zu SPARQL beziehungsweise SemwidgQL geben. Danach sollten sie jeweils sechs Eigenschaften von SPARQL und SemwidgQL anhand von fünfstufigen numerischen Rating-Skala bewerten. Der Minimalwert war jeweils negativ und der Maximalwert war jeweils positiv besetzt. Die Eigenschaften waren Erlernbarkeit (schwer erlernbar — leicht erlernbar), Intuitivität (nicht intuitiv — intuitiv), logischer Aufbau (unlogisch aufgebaut — logisch aufgebaut), Verständlichkeit (unverständlich — verständlich), Schreibaufwand (aufwendig zu schreiben — nicht aufwendig zu schreiben) und Eleganz (unelegant — elegant). Zuletzt wurden die Probanden explizit nach ihrer persönlich präferierten Abfragesprache und einer kurzen Begründung für ihre Entscheidung gefragt.

Eine Woche nach der Einführungsveranstaltung fand eine Wiederholung der Studie statt. Hierbei wurde die jeweilige Abfragesprache, mit denen die beiden Gruppen die Aufgaben begannen gewechselt. Abgesehen davon blieben die Aufgaben zur Interpretation und Formulierung von Abfragen gleich. Im anschließenden Fragebogen wurden nur noch die Bewertungen der Abfragesprachen anhand der Rating-Skalen beibehalten sowie die Frage bezüglich der persönlich präferierten Abfragesprache und der Begründung für diese Entscheidung. Hinzugekommen ist die Frage, in welchem Maße sich die Probanden in der Zwischenzeit weiter mit einer beiden Abfragesprachen befasst haben.

Datenerfassung

Die Erfassung der Daten zur Interpretation und Formulierung von Abfragen erfolgte über die eigens geschriebene Website, auf der die Probanden ihre Aufgaben bearbeiten sollten. Hierbei wurde bei der Formulierung der Antworten jeder Tastenanschlag aufgezeichnet und zusammen mit einem Zeitstempel in einer zentralen Datenbank gespeichert. Wurde eine Abfrage an den SPARQL-Server gestellt, wurde dies ebenfalls gespeichert. Zusätzlich wurde gespeichert, ob die Abfrage gültig war oder Fehler enthielt. Ebenso wurden die Zeitpunkte gespeichert, an denen die Probanden eine Aufgabe starteten oder beendeten, die Musterlösung angezeigt bekamen und eine Aufgabe als erfolgreich beendet oder als aufgegeben markierten. Die Erfassung der Daten des Fragebogens erfolgte über das Onlinebefragungsportal SoSci Survey[37].

2.4.4 Ergebnisse

Im folgenden Unterabschnitt sollen die Ergebnisse der Nutzerstudie vorgestellt werden. Zu Beginn wurde die Korrektheit der Antworten untersucht. Anschließend wurden die Ergebnisse hinsichtlich der im Studiendesign vorgestellten neun abhängigen Variablen evaluiert. Betrachtet wurden dabei, welchen Einfluss die genutzte Abfragesprache auf die durchschnittliche Leistung und Lerneffekte hatte. Ebenso wurde die Leistung der Probanden hinsichtlich der Komplexität der einzelnen Aufgaben hin analysiert. Im Rahmen der dabei genutzten statistischen Tests, wurden nur noch Daten betrachtet, die sich aus vollständig korrekten Antworten oder Antworten mit geringfügigen Fehlern ergaben. Daten aus falsch beantworteten Antworten oder abgebrochenen Aufgaben würden zweifellos die Ergebnisse verzerren. Abschließend wurden die subjektiven Einschätzungen der Probanden untersucht, welche sich aus der Auswertung der Fragebögen ergaben.

[37] `https://www.soscisurvey.de`

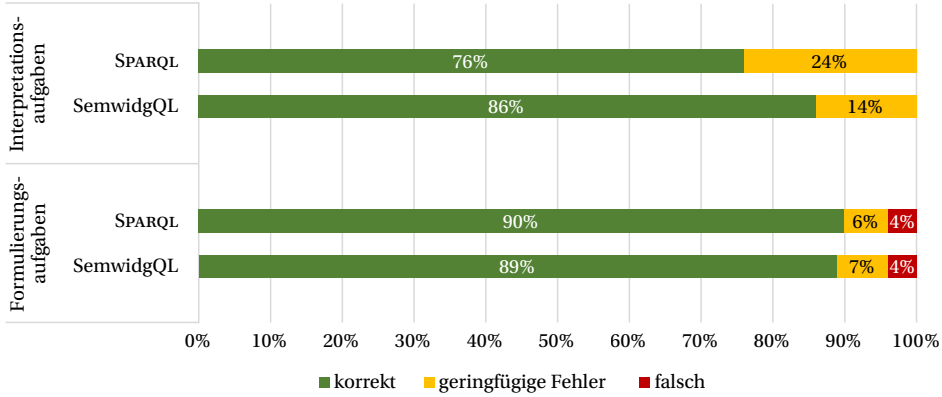


Abbildung 2.5: Korrektheit der Antworten.

Die Auswertung der quantitativen Daten erfolgte deskriptiv und vergleichend. Ergebnisse, die als statistisch signifikant bezeichnet werden, genügen mindestens einem fünfprozentigen Signifikanzniveau. Die Effektstärken werden anhand der Kategorisierung von Cohen (1992) beurteilt.

Korrektheit der Antworten

Wie auch bei Reisner et al. (1975), Welty & Stemple (1981) und Lassila et al. (2015) wurden die Antworten in die drei Kategorien *korrekte Antworten*, *Antworten mit geringfügigen Fehlern* und *falsche Antworten* unterteilt. Allerdings wurde darauf verzichtet, die Fehlertypen genauer zu bestimmen. Antworten mit geringfügigen Fehlern waren zwar syntaktisch korrekt und konnten vom SPARQL-Endpoint verarbeitet werden, enthielten allerdings Ungenauigkeiten. So wurden zum Beispiel Abfragen formuliert, die alle Triple-Pattern enthielten, um eine gewünschte Eigenschaft abzufragen, die entsprechende Variable wurde jedoch nicht dem SELECT-Ausdruck hinzugefügt. Diese Abfragen waren aber immer so ähnlich zur Musterlösung, dass eindeutig erkennbar war, dass die Probanden das Richtige meinten. Falsche Antworten waren syntaktisch inkorrekt, haben nicht die Anforderungen der Fragestellung erfüllt oder deren Bearbeitung wurden frühzeitig durch den Proband abgebrochen.

Insgesamt wurden pro Abfragesprache die Ergebnisse von 147 Aufgaben ausgewertet. Zu den Query-Interpretationsaufgaben gehörten 21 und zu den Query-Formulierungsaufgaben gehörten 126 Ergebnisse. Die Ergebnisse werden in Abbildung 2.5 grafisch dargestellt.

Werden die Query-Interpretationsaufgaben betrachtet, zeigt sich, dass die Probanden leicht bessere Ergebnisse bei der Interpretation von SemwidgQL-Abfragen erzielten, als bei der Interpretation von SPARQL-Abfragen. Bezüglich SemwidgQL waren 86 % der Antworten vollständig korrekt und 14 % der Antworten enthielten geringfügige Fehler. Bezüglich SPARQL waren 76 % der Antworten vollständig korrekt und 24 % der Antworten enthielten geringfügige Fehler. Weder für SemwidgQL, noch für SPARQL wurden falsche Antworten bei den Query-Interpretationsaufgaben gegeben.

Hinsichtlich der Query-Formulierungsaufgaben zeigten sich keine nennenswerten Unterschiede zwischen SPARQL und SemwidgQL. Bezüglich SPARQL waren 90 % der Antworten vollständig korrekt, 6 % der Antworten enthielten geringfügige Fehler und 4 % der Antworten waren

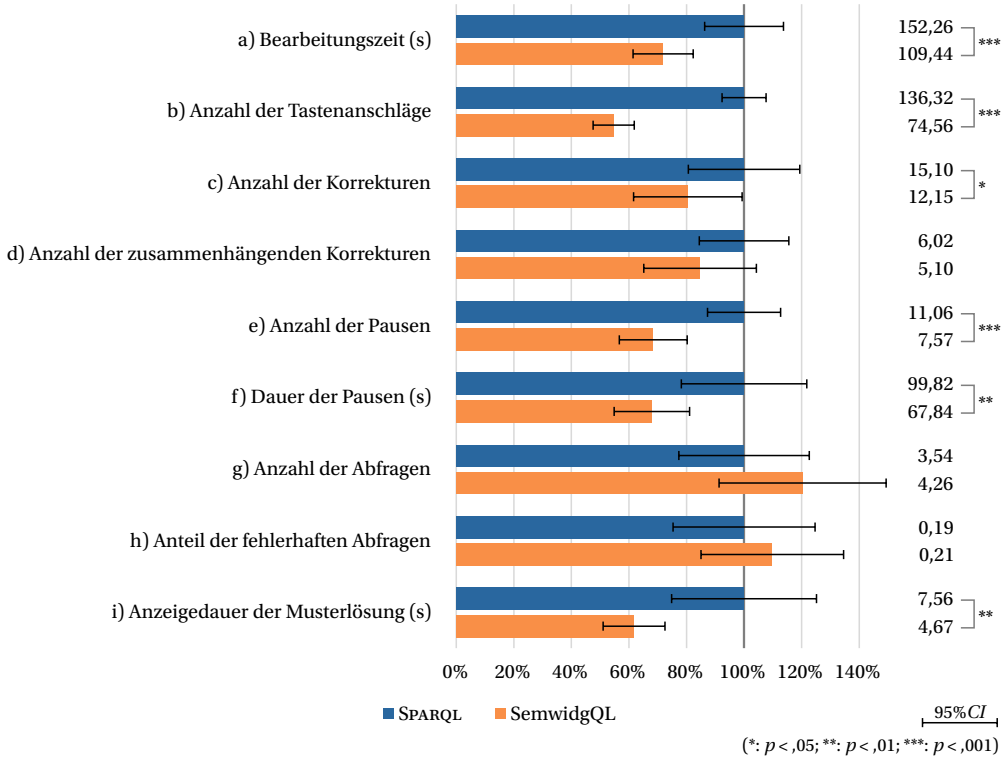


Abbildung 2.6: Durchschnittliche Leistungen der Probanden. Da die abhängigen Variablen unterschiedliche Maßstäbe und Einheiten aufweisen, werden die Werte – zur besseren Vergleichbarkeit – für SemwidgQL in Relation zu denen für SPARQL dargestellt. Die absoluten Durchschnittswerte werden am rechten Rand angezeigt.

falsch. Bezüglich SemwidgQL waren 89 % der Antworten vollständig korrekt, 7 % der Antworten enthielten geringfügige Fehler und 4 % der Antworten waren falsch.

Wird die Korrektheit der Antworten in Verbindung mit der *Anzeigedauer der Musterlösungen (i)* untersucht, zeigt sich, dass die Probanden die Musterlösungen bei korrekt beantwortete Fragen durchschnittlich 5,82 Sekunden, bei Antworten mit geringfügigen Fehlern 12,46 Sekunden und bei falsch beantworteten Fragen 12,82 Sekunden betrachtet haben. Dies unterstützt die Annahme, dass diese Variable als Indikator für das Vertrauen eines Probanden in die eigene Lösung dienen kann.

Analyse der durchschnittlichen Leistung

Die durchschnittlichen Leistungen der Probanden bezüglich der zuvor genannten neun abhängigen Variablen wurde mittels mehrerer t-Tests für abhängige Stichproben verglichen. Die im weiteren Verlauf beschriebenen Ergebnisse werden in Abbildung 2.6 und Tabelle 2.2 dargestellt. Die Werte für SemwidgQL waren bezüglich sechs der neun abhängigen Variablen (*a, b, c, e, f, i*) statistisch signifikant besser als die für SPARQL. Die *Anzahl der zusammenhängenden Korrekturen (d)* war zwar bei der Nutzung von SemwidgQL geringer als bei der Nutzung von SPARQL, der Unterschied war jedoch nicht statistisch signifikant. Die *Anzahl der Abfragen (g)* und der *Anteil*

Tabelle 2.2: Unterschiede zwischen SPARQL und SemwidgQL.

	SPARQL		SemwidgQL		t-Test			
	M	*SD*	*M*	*SD*	$t(124)$	p	r	
a) Bearbeitungszeit (s)	152,26	117,81	109,44	89,85	4,60	<,001	,382	***
b) Anzahl der Tastenanschläge	136,32	58,90	74,56	55,02	14,33	<,001	,790	***
c) Anzahl der Korrekturen	15,10	16,49	12,15	16,10	2,04	,044	,180	*
d) Anzahl der zusammenhängenden Korrekturen	6,02	5,27	5,10	6,63	1,72	,088	,152	
e) Anzahl der Pausen	11,06	7,95	7,57	7,36	5,82	<,001	,463	***
f) Dauer der Pausen (s)	99,82	122,89	67,84	73,91	2,99	,003	,259	**
g) Anzahl der Abfragen	3,54	4,51	4,26	5,78	−1,45	,150	,129	
h) Anteil der fehlerhaften Abfragen	0,19	0,27	0,21	0,27	−0,56	,578	,050	
i) Anzeigedauer der Musterlösung (s)	7,56	10,75	4,67	4,60	2,83	,005	,247	**

(*: $p <$,05; **: $p <$,01; ***: $p <$,001)

der fehlerhaften Abfragen (h) waren bei der Nutzung von SemwidgQL höher. Diese Unterschiede waren jedoch ebenfalls nicht statistisch signifikant.

Im Fall der *Anzahl der Tastenanschläge (b)* wurde ein sehr hoher Pearson-Korrelationskoeffizient r ermittelt, was auf einen starken Effekt schließen lässt. In den Fällen der *Bearbeitungszeit (a)* und der *Anzahl der Pausen (e)* wurde ein mittlerer Effekt festgestellt. Nur bei dem *Anteil der fehlerhaften Abfragen (h)* konnte kein Effekt festgestellt werden. Bei allen anderen abhängigen Variablen (*c, d, f, g, i*) wurde ein schwacher Effekt ermittelt.

Analyse von Lerneffekten

Die Ergebnisse der Studie wurde auf Lerneffekte hin untersucht. Dabei wurden ermittelt, inwiefern sich die Leistungen der Probanden im ersten und zweiten Durchlauf der Studie voneinander unterscheiden. Ebenso wurden die Ergebnisse bezüglich der unterschiedlichen Leistungen bei der Nutzung von SPARQL und SemwidgQL miteinander verglichen. Für beide Fälle wurden wieder mehrerer t-Tests für abhängige Stichproben bezüglich der neun abhängigen Variablen berechnet. In Tabelle 2.3 werden die vollständigen Ergebnisse bezüglich der Unterschiede zwischen SPARQL und SemwidgQL pro Durchlauf und in Tabelle 2.4 bezüglich der Unterschiede zwischen erstem und zweitem Durchlauf pro Abfragesprache aufgeführt. Abbildung 2.7 fasst die Resultate beider Untersuchungen in grafischer Form zusammen.

Bei der Nutzung von SemwidgQL haben die Probanden im ersten Durchlauf bezüglich vier der neun abhängigen Variablen (*a, b, e, i*) statistisch signifikant bessere Ergebnisse erzielt als bei der Nutzung von SPARQL. Die Effektstärke entspricht in allen vier Fällen einem mittleren Effekt. Bezüglich zwei weiterer abhängigen Variablen (*c, f*) waren die Ergebnisse bei der Nutzung von SemwidgQL tendenziell besser, jedoch nicht statistisch signifikant. Hinsichtlich der *Anzahl der Korrekturen (c)* zeigte sich kein Effekt, hinsichtlich der *Dauer der Pausen (f)* ein schwacher Effekt. Bei der Auswertung von drei abhängigen Variablen (*d, g, h*) zeigten die Probanden bei der Nutzung von SPARQL ein besseres Ergebnis als bei der Nutzung von SemwidgQL, bei keinem bis schwachen Effekt.

Im zweiten Durchlauf haben die Probanden bei der Nutzung von SemwidgQL bezüglich fünf der neun abhängigen Variablen (*a, b, c, d, e*) statistisch signifikant besser abgeschnitten,

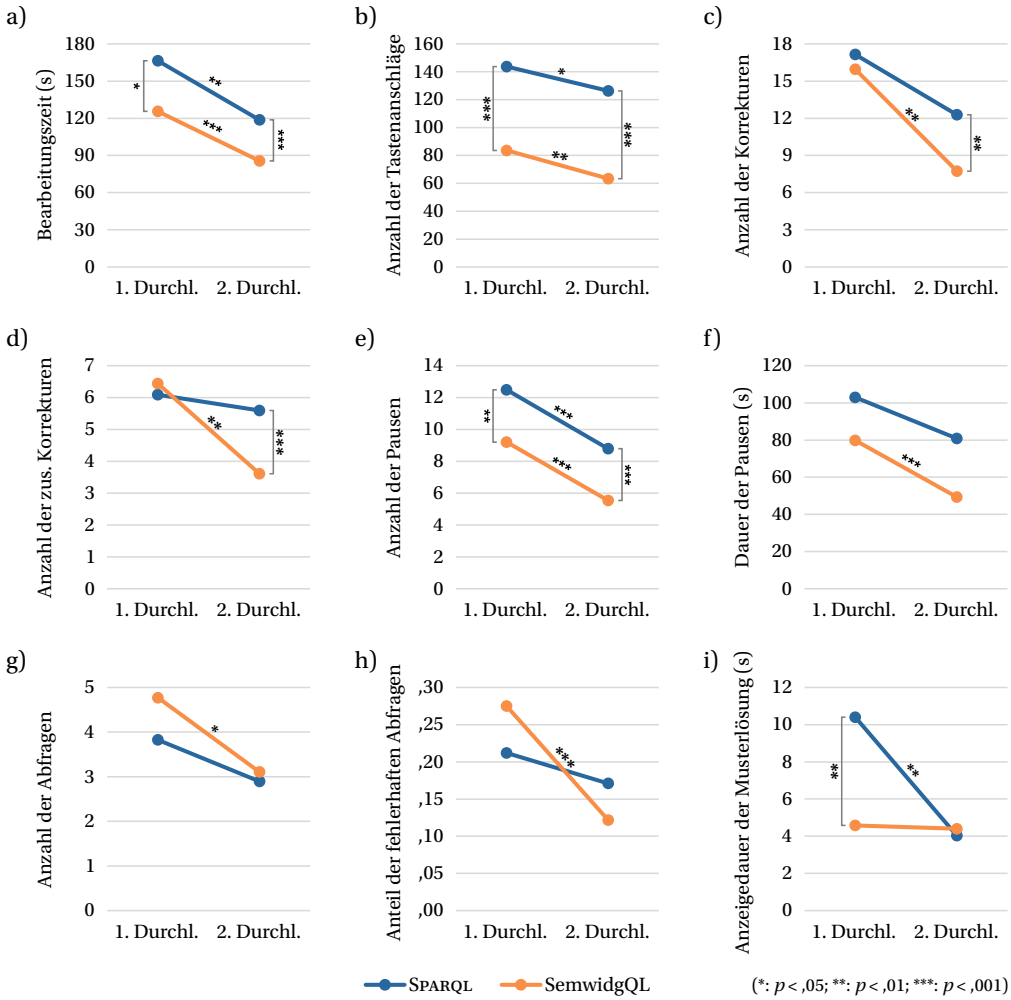


Abbildung 2.7: Unterschiede zwischen SPARQL und SemwidgQL pro Durchlauf, und Unterschiede zwischen dem ersten und zweiten Durchlauf pro Abfragesprache.

als bei der Nutzung von SPARQL. Die Effektstärken lagen in diesen Fällen im mittleren bis starken Bereich. Deskriptiv bessere Ergebnisse erzielten die Probanden bei der Nutzung von SemwidgQL bei zwei weiteren abhängigen Variablen (f, h) bei einem schwachen Effekt. Gering bessere Ergebnisse erreichten die Probanden bei der Nutzung von SPARQL bezüglich *Anzahl der Abfragen (g)* und der *Anzeigedauer der Musterlösung (i)*, welche jedoch statistisch nicht signifikant waren, noch einen Effekt aufwiesen.

Die Probanden haben sich bei der Nutzung von SemwidgQL hinsichtlich acht der neun abhängigen Variablen (*a, b, c, d, e, f, g, h*) statistisch signifikant verbessert. Hierbei wurde jeweils ein mittlerer Effekt gemessen. Bei der Nutzung von SPARQL zeigten die Probanden bezüglich vier abhängiger Variablen (*a, b, e, i*) statistisch signifikante Verbesserungen bei schwachem bis mittlerem Effekt. Durchschnittlich haben sich der Ergebnisse bei der Nutzung von SemwidgQL

Tabelle 2.3: Unterschiede zwischen SPARQL und SemwidgQL pro Durchlauf.

		SPARQL		SemwidgQL		t-Test			
	Durchlauf	*M*	*SD*	*M*	*SD*	*t*(56)	*p*	*r*	
a) Bearbeitungszeit (s)	1	166,33	140,71	125,51	103,54	2,60	,012	,328	*
	2	118,67	53,56	85,65	61,55	4,48	<,001	,513	***
b) Anzahl der Tastenanschläge	1	143,74	71,14	83,65	66,78	7,46	<,001	,706	***
	2	126,33	44,41	63,39	38,50	14,00	<,001	,882	***
c) Anzahl der Korrekturen	1	17,14	20,00	15,95	20,55	0,47	,641	,062	
	2	12,28	11,28	7,74	8,81	3,01	,004	,373	**
d) Anzahl der zus. Korrekturen	1	6,09	6,05	6,44	8,82	−0,36	,717	,049	
	2	5,60	3,65	3,61	3,33	4,01	<,001	,472	***
e) Anzahl der Pausen	1	12,49	9,52	9,21	8,89	3,13	,003	,386	**
	2	8,79	4,87	5,54	5,03	4,91	<,001	,549	***
f) Dauer der Pausen (s)	1	102,98	120,26	79,85	84,21	1,63	,109	,213	
	2	80,88	115,70	49,39	50,57	2,00	,051	,258	
g) Anzahl der Abfragen	1	3,82	5,25	4,77	6,53	−1,13	,264	,149	
	2	2,89	2,66	3,11	3,18	−0,46	,650	,061	
h) Anteil der fehlerhaften Abfragen	1	0,21	0,27	0,27	0,28	−1,23	,223	,163	
	2	0,17	0,26	0,12	0,22	1,12	,266	,148	
i) Anzeigedauer der Musterlösung (s)	1	10,40	14,60	4,58	4,63	2,94	,005	,366	**
	2	4,04	3,20	4,40	4,50	−0,51	,613	,068	

(*: $p <$,05; **: $p <$,01; ***: $p <$,001)

Tabelle 2.4: Unterschiede zwischen erstem und zweitem Durchlauf pro Abfragesprache.

		1. Durchlauf		2. Durchlauf		t-Test			
	Sprache[a]	*M*	*SD*	*M*	*SD*	t(56)	*p*	*r*	
a) Bearbeitungszeit (s)	$\mathcal{A}$	166,33	140,71	118,67	53,56	2,84	,006	,355	**
	$\mathcal{B}$	125,51	103,54	85,65	61,55	3,85	<,001	,457	***
b) Anzahl der Tastenanschläge	$\mathcal{A}$	143,74	71,14	126,33	44,41	2,54	,014	,321	*
	$\mathcal{B}$	83,65	66,78	63,39	38,50	2,68	,010	,338	**
c) Anzahl der Korrekturen	$\mathcal{A}$	17,14	20,00	12,28	11,28	1,96	,056	,253	
	$\mathcal{B}$	15,95	20,55	7,74	8,81	3,40	,001	,413	**
d) Anzahl der zus. Korrekturen	$\mathcal{A}$	6,09	6,05	5,60	3,65	0,71	,483	,094	
	$\mathcal{B}$	6,44	8,82	3,61	3,33	2,90	,005	,361	**
e) Anzahl der Pausen	$\mathcal{A}$	12,49	9,52	8,79	4,87	3,57	<,001	,431	***
	$\mathcal{B}$	9,21	8,89	5,54	5,03	4,00	<,001	,471	***
f) Dauer der Pausen (s)	$\mathcal{A}$	102,98	120,26	80,88	115,70	1,03	,309	,136	
	$\mathcal{B}$	79,85	84,21	49,39	50,57	3,58	<,001	,432	***
g) Anzahl der Abfragen	$\mathcal{A}$	3,82	5,25	2,89	2,66	1,42	,161	,186	
	$\mathcal{B}$	4,77	6,53	3,11	3,18	2,53	,014	,320	*
h) Anteil der fehlerhaften Abfragen	$\mathcal{A}$	0,21	0,27	0,17	0,26	0,90	,374	,119	
	$\mathcal{B}$	0,27	0,28	0,12	0,22	3,77	<,001	,450	***
i) Anzeigedauer der Musterlösung (s)	$\mathcal{A}$	10,40	14,60	4,04	3,20	3,33	,002	,406	**
	$\mathcal{B}$	4,58	4,63	4,40	4,50	0,30	,765	,040	

[a] ($\mathcal{A}$: SPARQL, $\mathcal{B}$: SemwidgQL)

(*: $p <$,05; **: $p <$,01; ***: $p <$,001)

Tabelle 2.5: Vergleich der Komplexitätsmaße bezüglich der SPARQL-Musterlösungen.

	Aufgabe								
	4	5	6	7	8	9	10	11	12
Halstead D	2,67	4,00	3,50	8,25	3,60	7,71	3,75	8,40	9,10
SSE-basierte Komplexität c	2	3	3	4	3	6	3	5	8

um 36 % verbessert. Bei der Nutzung von SPARQL haben sie sich durchschnittlich um 25 % verbessert. Die Probanden haben bei der Nutzung von SPARQL nie statistisch signifikant besser abgeschnitten als bei der Nutzung von SemwidgQL.

Komplexitätsbasierte Analyse der Studienergebnisse

Im Folgenden werden die Studienergebnisse hinsichtlich der in Abschnitt 2.4.1 vorgestellten Komplexitätsmetrik nach Halstead (1977) und der in Abschnitt 2.4.2 vorgestellten eigenen Komplexitätsmetrik, basierend auf der Notation in »SPARQL Syntax Expressions« (SSE), hin untersucht. Es ist davon auszugehen, dass die Komplexität einer Aufgabe ein Prädiktor für die gemessen Studienergebnisse bezüglich der neun abhängigen Variablen ist. Ziel ist es, zu ermitteln, wie gut oder schlecht die Probanden eine Reihe unterschiedlich komplexer Aufgaben bei der Nutzung der beiden Abfragesprache lösen können – nicht welche Abfragesprache komplexer oder weniger komplex ist. Um die Komplexität einer Aufgabe zu bestimmen, wurde stellvertretend die jeweilige SPARQL-Musterlösung gewählt, für welche mittels beider Verfahren ein konkreter Wert berechnet werden kann.

Die Komplexitätswerte für die SPARQL-Musterlösung werden in Tabelle 2.5 aufgeführt. Die errechneten Komplexitätswerte nach Halstead (D) und die der SSE-basierten Komplexitätsmetrik (c) zeigen bei Aufgaben ohne FILTER-Ausdrücke (4, 5, 6, 8, 10) keine großen Unterschiede, während die D-Werte bei Aufgaben mit FILTER-Ausdrücken (7, 9, 11, 12) deutlich höher sind, als die c-Werte.

Für alle abhängigen Variablen wurden jeweils lineare Regressionsanalysen mit D und c als Prädiktoren und SPARQL und SemwidgQL als Abfragesprachen berechnet. Die Bestimmtheitsmaße R^2 dieser Regressionsanalysen werden in Tabelle 2.6 aufgeführt. Diese sollen im hier vorliegenden Fall als Gütekriterien für der Qualität der jeweiligen Metrik fungieren. Basierend auf diesen Daten wurde ein Wilcoxon-Vorzeichen-Rang-Test durchgeführt, welcher die These unterstützt, dass die SSE-basierte Komplexitätsmetrik besser als die Halstead-Metrik dazu geeignet ist, die Komplexität der Aufgaben zu bestimmen. Der Median für c, $Mdn = ,85$, unterscheidet sich statistisch signifikant vom Median für D, $Mdn = ,53$ ($z = -3{,}03$; $p = ,002$). Hinsichtlich acht der neun abhängigen Variablen sind die Ergebnisse der SSE-basierten Komplexitätsmetrik näher an einem Optimalwert, als die der Halstead-Metrik. Nachfolgend werden daher nur noch die Ergebnisse betrachtet, die in Relation mit der SSE-basierten Komplexitätsmetrik berechnet wurden.

Die Ergebnisse der linearer Regressionsanalysen mit c als Prädiktor für alle neun abhängigen Variablen und SPARQL und SemwidgQL als Abfragesprachen werden in Tabelle 2.7 aufgeführt und in Abbildung 2.8 grafisch dargestellt. Die Ergebnisse deuten darauf hin, dass es einen statistisch signifikanten Zusammenhang zwischen der Komplexität c einer Aufgabe und acht

Tabelle 2.6: Vergleich der Bestimmtheitsmaße R^2, die sich aus den Regressionsanalysen ergeben.

		R^2								
		a)	b)	c)	d)	e)	f)	g)	h)	i)
Halstead D	SPARQL	,61	,70	,42	,47	,72	,54	,34	,57	,44
	SemwidgQL	,52	,54	,46	,55	,55	,52	,62	,36	,16
SSE-basierte Komplexität c	SPARQL	,93	,93	,84	,86	,96	,83	,72	,23	,73
	SemwidgQL	,89	,90	,78	,78	,92	,90	,89	,07	,51

Tabelle 2.7: Lineare Regressionsanalysen mit den Komplexitätswerten c als Prädiktoren.

	Sprache[a]	$F(1,7)$	p	R^2	f	$h_i(c)$[b]
a) Bearbeitungszeit (s)	$\mathcal{A}$	92,33	< ,001	,93	3,63	$46{,}19c - 31{,}83$
	$\mathcal{B}$	57,90	< ,001	,89	2,88	$40{,}24c - 47{,}17$
b) Anzahl der Tastenanschläge	$\mathcal{A}$	92,43	< ,001	,93	3,63	$23{,}90c + 41{,}98$
	$\mathcal{B}$	64,04	< ,001	,90	3,02	$21{,}32c - 9{,}57$
c) Anzahl der Korrekturen	$\mathcal{A}$	36,93	,001	,84	2,30	$4{,}00c - 1{,}30$
	$\mathcal{B}$	24,22	,002	,78	1,86	$4{,}33c - 4{,}95$
d) Anzahl der zus. Korrekturen	$\mathcal{A}$	42,98	< ,001	,86	2,48	$1{,}67c - 0{,}77$
	$\mathcal{B}$	25,06	,002	,78	1,89	$1{,}82c - 2{,}13$
e) Anzahl der Pausen	$\mathcal{A}$	156,63	< ,001	,96	4,73	$3{,}08c - 0{,}72$
	$\mathcal{B}$	81,27	< ,001	,92	3,41	$3{,}24c - 5{,}02$
f) Dauer der Pausen (s)	$\mathcal{A}$	34,19	,001	,83	2,21	$39{,}51c - 50{,}15$
	$\mathcal{B}$	60,02	< ,001	,90	2,93	$36{,}70c - 71{,}90$
g) Anzahl der Abfragen	$\mathcal{A}$	18,27	,004	,72	1,62	$2{,}54c - 5{,}95$
	$\mathcal{B}$	59,15	< ,001	,89	2,91	$2{,}18c - 3{,}41$
h) Anteil der fehlerhaften Abfragen	$\mathcal{A}$	2,05	,195	,23	0,54	$0{,}05c + 0{,}07$
	$\mathcal{B}$	0,52	,494	,07	0,27	$0{,}02c + 0{,}23$
i) Anzeigedauer der Musterlösung (s)	$\mathcal{A}$	19,27	,003	,73	1,66	$2{,}27c - 1{,}64$
	$\mathcal{B}$	7,41	,030	,51	1,03	$1{,}33c - 0{,}53$

[a] ($\mathcal{A}$: SPARQL; $\mathcal{B}$: SemwidgQL)

[b] (h: lineare Regressionsgleichung; i: gemessene Variable in Abhängigkeit der Sprache; c: Komplexität)

der neun abhängigen Variablen (*a, b, c, d, e, f, g, i*) gibt. Lediglich bei dem *Anteil der fehlerhaften Abfragen (h)* konnte kein Zusammenhang nachgewiesen werden. Die errechneten Bestimmtheitsmaße waren in fast allen Fällen sehr hoch. Nach der strengeren Klassifikation von Hair et al. (2011) weisen 13 der 18 Ergebnisse auf einen wesentlichen Einfluss ($R^2 \geq ,75$) der Prediktorvariable auf die Leistung der Probanden hin. Die Effektstärken f waren ebenfalls in fast allen Fällen sehr stark.

Die Regressionsgeraden, die sich aus den Messergebnissen bei der Nutzung von SemwidgQL ergaben, lagen in dem hier untersuchten Umfang der Aufgabenkomlexität c in sieben der neun abhängigen Variablen (*a, b, c, d, e, f, i*) vollständig unterhalb Regressionsgeraden bei der Nutzung von SPARQL. In vier dieser sieben Fälle steigen die SemwidgQL-Regressionsgeraden weniger steil an, als die SPARQL-Regressionsgeraden und würden sie somit auch bei komplexeren Aufgaben nicht schneiden. In den anderen drei Fällen schneiden die SemwidgQL-Regressionsgeraden die SPARQL-Regressionsgeraden bei höheren Komplexitätswerten. Bezüg-

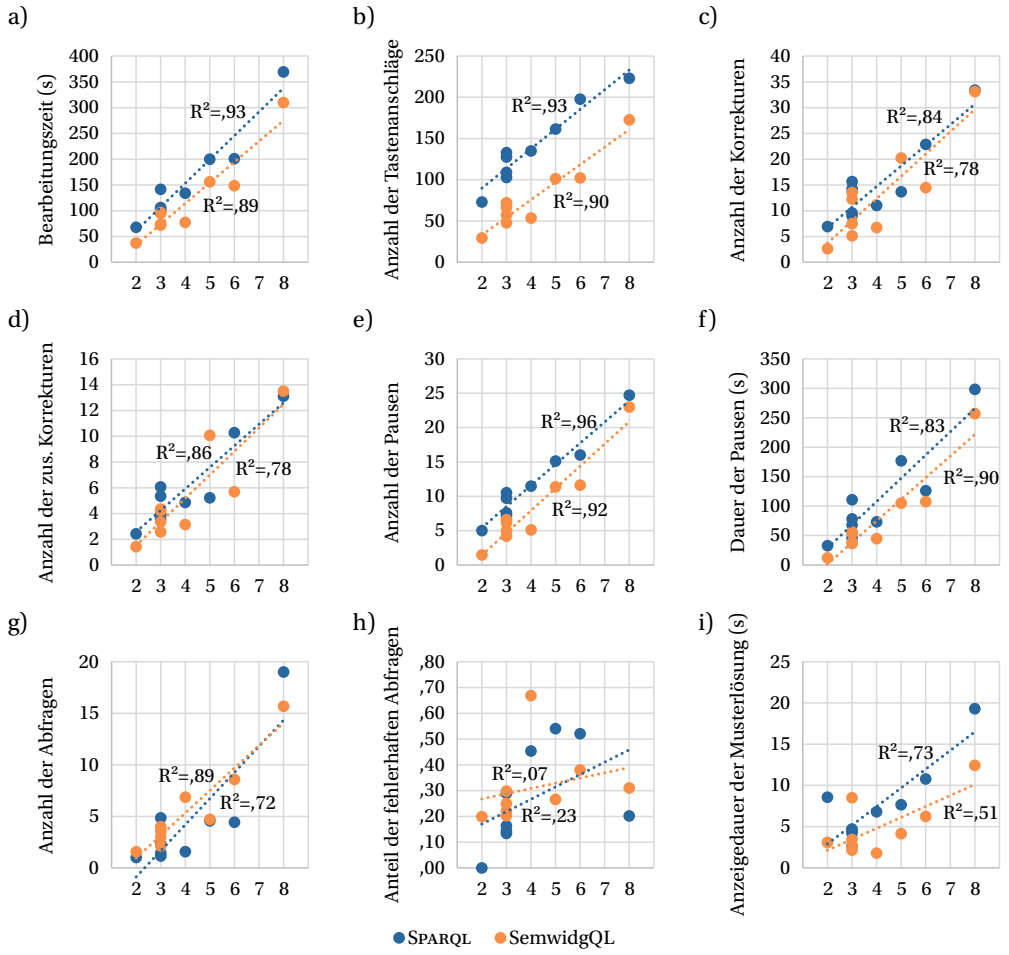


Abbildung 2.8: Lineare Regressionsanalysen mit der Aufgabenkomplexität als Prädiktor.

lich der beiden abhängigen Variablen (*g*, *h*), bei denen sie SemwidgQL-Regressionsgeraden nicht vollständig unter den SPARQL-Regressionsgeraden liegen, schneiden diese sich bei mittleren bei mittelhohen Komplexitätswerten.

Auswertung der subjektiven Einschätzungen der Probanden

Im folgenden Abschnitt werden die Ergebnisse ausgewertet, die sich aus den Fragebögen ergaben, welche von den Probanden jeweils am Ende der Studie ausgefüllt wurden. Bezüglich der subjektiv angegebenen Aufmerksamkeit während der beiden Vorträge zu SPARQL und SemwidgQL konnten keine statistischen Unterschiede festgestellt werden. Mit durchschnittlich 82 % der maximalen Aufmerksamkeit beim Vortrag zu SemwidgQL, lagen dieser Werte minimal unter dem durchschnittlichen Wert von 85 % beim Vortrag zu SPARQL.

Um die subjektiven Einschätzungen der Probanden hinsichtlich der sechs zu bewertenden Charakteristiken der beiden Abfragesprachen auszuwerten, wurden mehrere Wilcoxon-

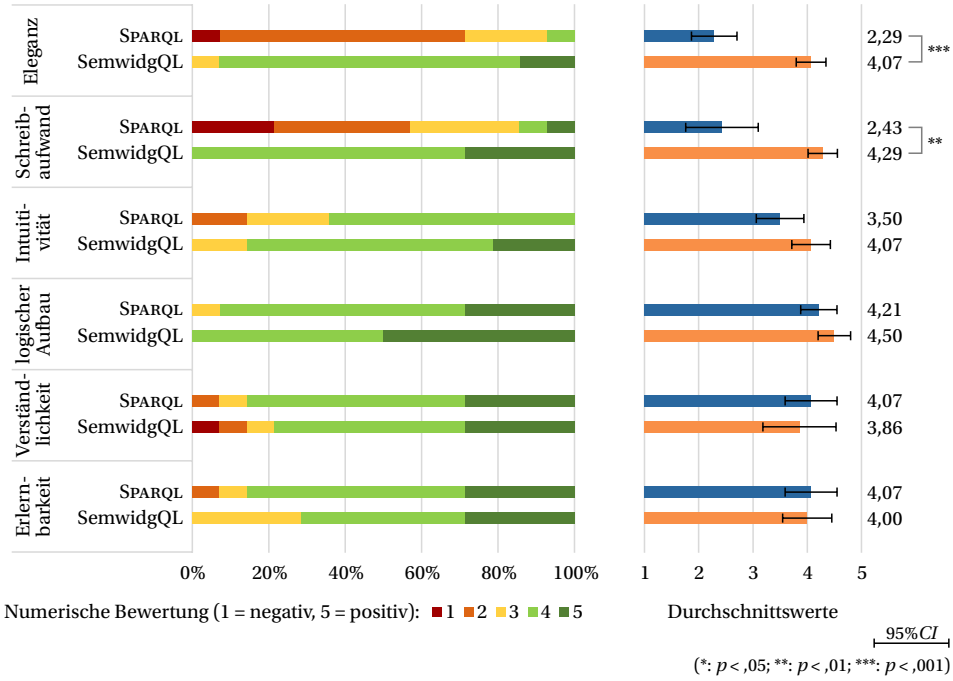


Abbildung 2.9: Subjektive Bewertung von SPARQL und SemwidgQL.

Vorzeichen-Rang-Test durchgeführt. Hinsichtlich der *Eleganz* wurde SemwidgQL, $Mdn = 4$, statistisch signifikant besser bewertet als SPARQL, $Mdn = 2$ ($z = -3{,}23$; $p < {,}001$). Ebenso wurde SemwidgQL hinsichtlich des *Schreibaufwands*, $Mdn = 4$, statistisch signifikant besser bewertet als SPARQL, $Mdn = 2$ ($z = -3{,}04$; $p = {,}002$). Bezüglich der *Intuitivität* wurde SemwidgQL, $Mdn = 4$, tendenziell besser bewertet als SPARQL, $Mdn = 4$ ($z = -1{,}73$; $p = {,}084$). Im Hinblick auf den *logischen Aufbau* (SemwidgQL $Mdn = 4{,}5$; SPARQL $Mdn = 4$; $z = -1{,}27$; $p = {,}206$), die *Verständlichkeit* (SemwidgQL $Mdn = 4$; SPARQL $Mdn = 4$; $z = -1{,}13$; $p = {,}257$) und die *Erlernbarkeit* (SemwidgQL $Mdn = 4$; SPARQL $Mdn = 4$; $z = -0{,}29$; $p = {,}773$) konnte kein statistischer Unterschied festgestellt werden. Die Ergebnisse wurde in Abbildung 2.9 grafisch aufbereitet.

Bei der Frage nach der persönlich präferierten Abfragesprache gaben die Probanden zusammen neun verschiedene Vorzüge bezüglich SemwidgQL mit insgesamt 30 individuellen Nennungen an. Insbesondere die Kürze der Abfragen, die Ähnlichkeit zu objektorientierten Programmiersprachen und die Einfachheit beim Schreiben wurden häufig (jeweils ≥ 5) genannt. Zu SPARQL wurden von den Probanden drei verschiedene Vorzüge mit insgesamt fünf Nennungen angegeben. Alle Nennungen sind in Abbildung 2.10 dargestellt.

In 79 % der Antworten wurde SemwidgQL als die bevorzugte Abfragesprache genannt. Dementsprechend wurde SPARQL in 21 % der Antworten als bevorzugte Abfragesprache genannt. Im zweiten Durchlauf hat ein Proband, welcher im ersten Durchlauf SPARQL als bevorzugte Abfragesprache genannt hat, seine Meinung geändert und ist zu SemwidgQL übergewechselt.

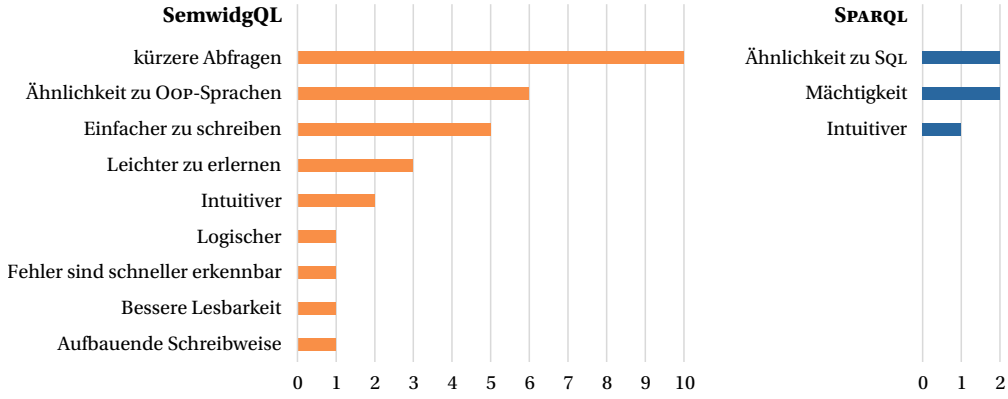


Abbildung 2.10: Kumulierte Anzahl der Nennung von Vorzügen hinsichtlich einer der beiden Abfragesprachen gegenüber der jeweils anderen.

2.4.5 Diskussion der Ergebnisse

Anhand der Ergebnisse der Nutzerstudie konnte gezeigt werden, dass die Probanden statistisch signifikant bessere Ergebnisse bezüglich der meisten der untersuchten abhängigen Variablen bei der Nutzung von SemwidgQL erzielten als bei der Nutzung von SPARQL. Insbesondere die Ergebnisse hinsichtlich der Bearbeitungszeit, der Anzahl der Korrekturen sowie die Anzahl und Dauer der genommenen Denkpausen deuten darauf hin, dass SemwidgQL einfacher zu nutzen ist als SPARQL. Lediglich bei der Anzahl der Abfragen und dem Anteil der fehlerhaften Abfragen wurden bei den Probanden höhere Werte gemessen. Eine höhere Anzahl von Abfragen muss jedoch nicht zwingend negativ gedeutet werden. Ebenso kann dies darauf hindeuten, dass SemwidgQL ein exploratives Vorgehen bei der Erstellung der Abfragen unterstützt. Durch den Wegfall der in SPARQL benötigten Schlüsselwörter (SELECT, WHERE, FILTER etc.) sowie die Nennung von Variablen für die Ausgabe ist der Overhead, den die Erstellung einer SemwidgQL-Abfrage mit sich bringt, deutlich geringer als bei der Erstellung einer SPARQL-Abfrage. Somit befanden sich die Probanden erheblich schneller in einer Situation, in der sie eine Abfrage an den SPARQL-Endpoint schicken konnten.

Es konnte allerdings kein Unterschied bezüglich der Korrektheit der bearbeiteten Aufgaben festgestellt werden. Dies lässt aber zumindest darauf schließen, dass den Probanden beide Abfragesprachen in einem gleich guten Maß gelehrt wurden und es zu keiner bewussten oder unbewussten Beeinflussung der Probanden durch den Studienleiter gekommen ist.

Nach der Einführungsveranstaltung haben die Probanden hinsichtlich sechs der neun untersuchten abhängigen Variablen bessere Leistungen bei der Nutzung von SemwidgQL gezeigt als bei der Nutzung von SPARQL. Im zweiten Durchlauf konnten sich die Probanden in fast allen Fällen statistisch signifikant verbessern. Hinsichtlich der Anzeigedauer der Musterlösung verblieb der ohnehin sehr niedrige Wert auf einem nahezu gleichen Niveau. Bei der Nutzung von SPARQL konnten sich die Probanden zwar auch durchgehend verbessern, jedoch waren diese Verbesserungen nur in vier von neun Fällen statistisch signifikant. Da die Probanden bereits im ersten Durchlauf bessere Leistung bei der Nutzung von SemwidgQL gezeigt und sich im zweiten Durchlauf im Vergleich zu SPARQL noch einmal deutlich stärker verbessern konnten, lässt sich annehmen, dass SemwidgQL einfacher zu lernen ist.

Die Ergebnisse der linearen Regressionsanalyse deuten darauf hin, dass die vergleichsweise besseren Ergebnisse bei der Nutzung von SemwidgQL nicht nur dadurch erzielt wurden, weil nur bereits einfache Aufgaben mit dieser Sprache noch weiter vereinfacht wurden, sondern dass sich auch komplexere Aufgaben mit SemwidgQL leichter lösen lassen als mit SPARQL. Einige der ermittelten Regressionsgeraden deuten an, dass sich auch noch komplexere Aufgaben, als die hier untersuchten, mit SemwidgQL leichter lösen lassen. Die Regressionsgeraden zu anderen abhängigen Variablen lassen aber auch darauf schließen, dass komplexere Aufgaben besser mit SPARQL zu lösen sind. Da die an einem Stück geschriebenen Pfadabfragen von SemwidgQL mit steigender Länge immer unübersichtlicher werden, ist allerdings davon auszugehen, dass SPARQL in diesen Fällen die geeignetere Abfragesprache ist. SemwidgQL war jedoch auch nie dazu gedacht SPARQL in diesem Bereich zu ersetzen.

Die guten Ergebnisse der objektiven Messungen werden durch die subjektiven Bewertungen seitens der Probanden weiter untermauert. So wurde eine erheblich größere Anzahl von Vorzügen für SemwidgQL genannt als für SPARQL. Besonders die Kürze der Abfragen und die Ähnlichkeit zu objektorientierten Programmiersprachen wurde von den Probanden häufig erwähnt. Zudem hat im zweiten Durchlauf ein Proband, welcher im ersten Durchlauf SPARQL als bevorzugte Abfragesprache genannt hat, seine Meinung geändert und bevorzuge nun SemwidgQL. Schlussendlich werden die Messergebnisse durch die explizite persönlich Präferenz der Probanden von SemwidgQL in 79 % der ausgefüllten Fragebögen unterstrichen.

Es muss jedoch darauf hingewiesen werden, dass die Anzahl der Probanden verhältnismäßig klein war, was sich auf die Aussagekraft der Studienergebnisse limitierend auswirkt.

Zusammenfassend lässt sich feststellen, dass die Ziele, die bei der Entwicklung von SemwidgQL im Vordergrund standen – nämlich Effektivität, Effizienz und leichte Erlernbarkeit – vollständig umgesetzt werden konnten. Die Probanden waren genauso effizient bei der Nutzung von SemwidgQL wie bei der Nutzung von SPARQL. Sie zeigten in fast allen Fällen statistisch signifikant bessere Leistungen bei der Nutzung von SemwidgQL als bei der Nutzung von SPARQL und waren somit effizienter. Diese Ergebnisse erzielten sie nicht nur bei einfachen Aufgaben, stattdessen waren die guten Ergebnisse unabhängig von der Komplexität der Aufgaben. Im Hinblick auf die Erlernbarkeit konnten sich die Probanden im zweiten Durchlauf bei der Nutzung von SemwidgQL in fast allen Bereichen statistisch signifikant verbessern, während sie sich bei der Nutzung von SPARQL meistens nicht statistisch signifikant verbessern konnten. Dies lässt darauf schließen, dass SemwidgQL leichter zu erlernen ist.

2.5 Empirische Evaluation der Ausdrucksfähigkeit

In der folgenden empirische Evaluation wurde die Ausdrucksfähigkeit von SemwidgQL untersucht. Als Datenbasis diente der LSQ-Datensatz mit mehreren Million protokollierten Abfragen von großen öffentlich zugänglichen SPARQL-Endpoints. Ziel der Evaluation war es zu überprüfen, wie groß der Anteil der in der Praxis genutzten SPARQL-Abfragen ist, der auch durch semantisch äquivalente SemwidgQL-Abfragen erzeugt werden kann. Da es momentan jedoch keine automatische Möglichkeit gibt, um zu überprüfen, ob eine semantisch äquivalente SemwidgQL-Abfrage zu einer gegebenen SPARQL-Abfrage existiert, musste diese Überprüfung manuell durchgeführt werden. Dies war bei der großen Anzahl der im LSQ-Datensatz enthaltenen Abfragen jedoch nicht praktikabel. Daher wurden die Abfragen in eine parametrisierte Form überführt, in der keine konkreten Resources, Variablennamen oder Literale mehr enthalten waren. Diese parametrisierten Abfragen konnten anschließend zusammengefasst werden, wodurch sich die große Anzahl der Abfragen auf eine kleine handhabbare Menge von Abfrage-Patterns reduzieren lies. Generell entspricht die Auswahl der SPARQL-Endpoints des LSQ-Datensatzes derjenigen, die von der angestrebten Nutzergruppe von SemwidgQL ebenfalls genutzt werden würde. Im speziellen wäre hier der DBpedia-Endpoint zu nennen.

> Die nachfolgenden Studienergebnisse und die daraus gezogenen Schlussfolgerungen wurden zum Großteil im Rahmen der »16th International Semantic Web Conference (ISWC 2017)« vorgestellt und in Stegemann & Ziegler (2017a) sowie Stegemann & Ziegler (2017b) veröffentlicht.

2.5.1 Analyse des Linked-SPARQL-Query-Datensatzes

Der Linked-SPARQL-Query-Datensatz (LSQ), veröffentlicht von Saleem et al. (2015), enthält etwa 1,75 Millionen einzigartige Abfragen[38], die insgesamt 5,68 Millionen Mal ausgeführt wurden. Die Abfragen stammen von vier verschiedenen SPARQL-Endpoints[39]. Der LSQ-Datensatz liegt im RDF-Format vor und enthält neben dem eigentlichen Text der Abfrage unter anderem das Ausführungsdatum, die Anzahl der zurückgelieferten Ergebnissets zum Zeitpunkt der Ausführung, gegebenenfalls aufgetretene Fehler und die Art der Abfrage. Der Datensatz wurde sowohl von den Autoren selbst wie auch schon von anderen (z. B. Han et al., 2016; Bonifati et al., 2017) mittels statistischer Tests untersucht. Dabei wurden unter anderem die Nutzung SPARQL-spezifischer Funktionen (UNION, DISTINCT, FILTER etc.) oder die verschiedenen Arten der Verbindungen der genutzten Triple-Patterns untereinander (*star, path, sink* etc.) untersucht. Eine Analyse auf Basis der Abfrage-Patterns, wie im Folgenden beschrieben, war jedoch bisher vollkommen neuartig.

Datenaufbereitung

Im Rahmen der Evaluation wurden alle Abfragen aus dem LSQ-Datensatz entfernt, die irgendeinen Syntax-Fehler enthielten, oder kein gültiges Ergebnis zum Zeitpunkt der Ausführung produzierten. Zu den Abfragen, die an den SPARQL-Endpoint des British Museum gestellt wurden,

[38] Stand: Juli 2017

[39] DBpedia (`http://dbpedia.org`),
LinkedGeoData (LGD, `http://linkedgeodata.org`),
Semantic Web Dog Food (SWDF, `http://data.semanticweb.org`)
und British Museum.

Listing 2.15: Beispielhafte Umwandlung einer SPARQL-Abfrage in eine parametrisierte Form.

```
1  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2  SELECT DISTINCT *
3    WHERE { ?value rdfs:label "Albert Einstein"@de . }
4  LIMIT 10
```

⇓

```
5  SELECT ?v0 WHERE { ?v0 <i0> "l0"@lang }
```

wurden keine Informationen darüber gespeichert, ob diese zum Zeitpunkt ihrer Ausführung ein gültiges Ergebnis zurückgeliefert haben. Um die Analyseergebnisse nicht zu verzerren, wurden daher alle Abfragen entfernt, die diesem SPARQL-Endpoint zuzuordnen waren. Da SemwidgQL ausschließlich SELECT-Abfragen erzeugen kann, wurden weiterhin alle ASK-, DESCRIBE- und CONSTRUCT-Abfragen entfernt. Übrig blieben 636 876 SELECT-Abfragen, die zusammen 1 526 804 Mal ausgeführt wurden. Diese SELECT-Abfragen repräsentieren 91,0 % aller gültigen Abfragen (ASK 4,5 %, DESCRIBE 3,8 % und CONSTRUCT 0,7 %). Während Saleem et al. den Datensatz im Ganzen untersucht haben, und dabei Informationen zu genutzten SPARQL-Konstrukten und Ausführungszeiten gesammelt haben, liegt der Fokus dieser Evaluation auf der Untersuchung von genutzten Abfrage-Patterns. Um die genutzten Abfrage-Patterns des LSQ-Datensatzes zu identifizieren, wurden die gesammelten SELECT-Abfragen in mehreren Schritten in eine parametrisierte Form überführt:

- Zuerst wurden alle Elemente des Abfrage-Textes entfernt, die keinen Einfluss auf das Pattern selbst haben. Dazu gehören die PREFIX-, FROM-, LIMIT-, OFFSET- und ORDER-BY-Elemente. Ebenfalls wurden alle DISTINCT-Schlüsselwörter und die entsprechende Klammerung entfernt.

- Im zweiten Schritt wurden alle IRIs, Variablenbezeichnungen, Literale und Sprachangaben jeder Abfrage in ein generisches Format überführt (IRIs: `<i#>`; Variablenbezeichnungen: `?v#`; Literale: `'l#'`; Sprachangaben: `@lang`. Hierbei entspricht »#« einem Zählwert, der anhand des ersten Vorkommens des entsprechenden Elements innerhalb der Abfrage ermittelt wird).

- Anschließend wurden Wildcards (`*`) in SELECT-Ausdrücken durch eine vollständige Liste der Variablenbezeichnungen aus den entsprechenden WHERE-Ausdrücken ersetzt.

- Im letzten Schritt wurden alle identisch parametrisierten Abfragen zusammengefasst und zusammen mit der Anzahl der Abfragen, die sie repräsentieren, und ihrer aggregierten Ausführungshäufigkeit gespeichert. Vollständig identische Abfrage-Patterns konnten automatisch zusammengefasst werden. Zusätzlich konnten einige weitere Patterns manuell zusammengefasst werden, die zwar nicht syntaktisch aber semantisch identisch waren. Dazu zählten zum Beispiel Abfragen mit den gleichen Triple-Patterns in einem WHERE-Ausdruck, die jedoch in unterschiedlicher Reihenfolge auftraten.

In Listing 2.15 wird diese Transformation anhand eines Beispiels verdeutlicht. Letztendlich konnte die Menge der insgesamt 636 876 SELECT-Abfragen auf nur noch 1619 Abfrage-Patterns reduziert werden.

	Anzahl der häufigsten Abfragen			
Anteil	Gesamt	DBpedia	LGD	SWDF
0,50	3	2	1	5
0,75	6	3	3	13
0,85	12	6	5	21
0,90	21	8	9	30
0,95	41	14	17	44
0,99	120	52	36	62

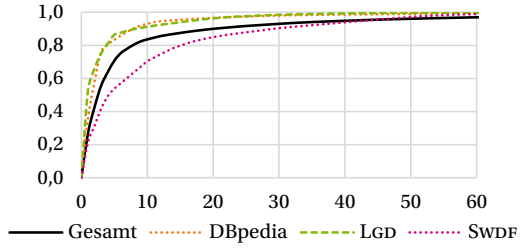


Abbildung 2.11: Angepasste kumulative Pareto-Verteilung.

	Anteil der insgesamt ausgeführten Abfragen			
Abfrage	Gesamt	DBpedia	LGD	SWDF
1	0,267	0,357	0,525	0,212
2	0,159	0,221	0,134	0,102
3	0,129	0,183	0,091	0,102
4	0,081	0,042	0,060	0,076
5	0,069	0,028	0,054	0,046
6	0,046	0,027	0,013	0,033
⋮	⋮	⋮	⋮	⋮
12	0,001	0,005	0,006	0,021

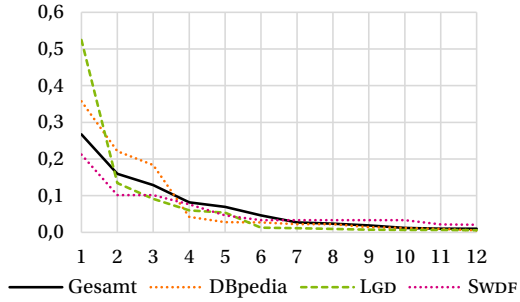


Abbildung 2.12: Häufigkeiten der Abfrage-Patterns für die am häufigsten ausgeführten Abfragen.

Allgemeine Analyse der Daten

Nachfolgend werden die 1619 Abfrage-Patterns näher betrachtet. Werden diese anhand der Häufigkeit ihrer Ausführungen sortiert, zeigt sich, dass die ersten 120 Abfrage-Patterns bereits 99 % aller ausgeführten Abfragen abbilden. Die ersten 41 Abfrage-Patterns repräsentieren 95 %, die ersten 21 Abfrage-Patterns repräsentieren 90 % und die ersten drei Abfrage-Patterns repräsentieren bereits mehr als 50 % aller ausgeführten Abfragen (vgl. Abbildung 2.11).

Werden die Daten für jeden Endpoint getrennt betrachtet, ergeben sich 1240 Abfrage-Patterns für den DBpedia-Endpoint, 289 für den LGD-Endpoint und 151 für den SWDF-Endpoint. Dabei zeigt sich, dass für verschiedene Endpoints verschiedene Abfrage-Patterns genutzt werden. Lediglich 17 Abfrage-Patterns wurden in allen drei untersuchten Endpoints genutzt, wobei diese jedoch 24,8 % der ausgeführten Abfragen abbilden. Nur 27 Abfrage-Patterns wurden in zwei verschiedenen Endpoints genutzt, und repräsentieren 9,5 % der ausgeführten Abfragen. Die restlichen 1575 Abfrage-Patterns wurden ausschließlich in einem einzigen Endpoint genutzt und repräsentieren dementsprechend 65,6 % der ausgeführten Abfragen.

Es ist hervorzuheben, dass für den LGD-Endpoint mehr als die Hälfte der Abfragen bereits durch das am häufigsten genutzte Abfrage-Pattern abgebildet wird. Für den DBpedia-Endpoint sind es die ersten beiden und für den SWDF-Endpoint die ersten fünf Abfrage-Patterns die mehr als die Hälfte aller gestellten Abfragen repräsentieren. Über 90 % aller gestellten Abfragen werden für den DBpedia-Endpoint von den ersten acht und für den LGD-Endpoint die ersten neun Abfrage-Patterns abgebildet. Im Gegensatz dazu werden für den SWDF-Endpoint mindestens 30 Abfrage-Patterns benötigt um einen gleichen Anteil der gestellten Abfragen darzustellen. Die

Kurven der Pareto-Verteilungen für den DBpedia- und LGD-Endpoint verlaufen entsprechend ähnlich (vgl. Abbildung 2.11), während die für den SWDF-Endpoint deutlich flacher verläuft. Die Häufigkeit der Abfrage-Patterns ist hier deutlich gleichmäßiger verteilt, als für den DBpedia- und LGD-Endpoint (vgl. Abbildung 2.12). Es ist jedoch anzumerken, dass 16,5 % aller vom SWDF-Endpoint ausgeführten Abfragen von einer PHP-Bibliothek[40] erzeugt wurden, die den Funktionsumfang eines SPARQL-Endpoints testet. Dies ist an den charakteristischen Abfragen der Bibliothek leicht zu erkennen.

Die zwölf am häufigsten genutzten Abfrage-Patterns des LSQ-Datensatzes repräsentieren bereits mehr als 85 % aller ausgeführten Abfragen. Eine visuelle Darstellung dieser Abfrage-Patterns findet sich in Abbildung 2.13. Die meisten dieser Abfrage-Patterns sind einfache Subjekt-Prädikat-Objekt-Relationen mit einer oder zwei Variablen. Diese werden in der Regel auch über das SELECT-Statement ausgegeben. FILTER-Ausdrücke werden nur selten, und auch nur auf sehr einfache Weise eingesetzt. In Abfrage-Pattern (g) wird die Variable für das Subjekt der Subjekt-Prädikat-Objekt-Relation anhand einer Menge von vordefinierten IRIs eingeschränkt. In den Abfrage-Patterns (d) und (i) werden jeweils die Werte für die zurückzuliefernden Variablen anhand der Sprache gefiltert. Im Abfrage-Pattern (k) wird dies nicht über einer FILTER-Ausdruckt, sondern über die Angabe eines Language-Tag realisiert. Abfrage-Pattern (k) ist das einzige der zwölf am häufigsten genutzten Patterns, welches ein Literal innerhalb eines Triple-Patterns nutzt. Die Abfrage-Patterns (b) und (k) besitzen mehrere Triple-Patterns, die sich das Subjekt teilen. Im ersten Fall ist dies ein IRI, im zweiten Fall eine Variable. Die Abfrage-Patterns (a), (b), (f) und (i) nutzen OPTIONAL-Relationen. Abfrage-Pattern (k) ist das einzige etwas komplexere Pattern in dieser Auswahl. Neben dem Einsatz von FILTER-Ausdrücken und dem OPTIONAL-Schlüsselwort, enthält es einen Pfad, der sich über zwei Triple-Patterns erstreckt, sowie ein Triple-Pattern, dessen Subjekt dem Prädikat einer vorherigen Relation entspricht.

In Tabelle 2.8 werden die zwölf am häufigsten genutzten Abfrage zusammen mit ihrem Rang innerhalb des gesamten Datensatzes sowie den Teildatensätzen der jeweiligen SPARQL-Endpoints dargestellt. Hierbei zeigt sich, dass an die verschiedenen SPARQL-Endpoints sehr unterschiedliche Abfragen gestellt werden. Die Abfrage-Patterns, die für die meisten Abfragen des DBpedia- und LGD-Endpoints verantwortlich waren, wurden an die jeweils anderen SPARQL-Endpoints nie gestellt. Vier der zwölf am häufigsten genutzten Abfrage-Patterns wurden an alle Endpoints gestellt, zwei wurden an zwei der Endpoints gestellt und die restlichen sechs Abfrage-Patterns wurden nur an einen Endpoint gestellt.

Wird der gesamte Datensatz betrachtet, so zeigt sich, dass in 14,5 % aller ausgeführten Abfragen die Ergebnisse anhand der Sprache gefiltert werden – entweder über einen FILTER-Ausdruck oder einen Language-Tag. FILTER-Ausdrücke, die die Sprache nicht einschränken, werden in 5,7 % genutzt. Vereinigungen von Ergebnismengen durch den UNION-Ausdruck kommen bei 7,5 % und Gruppierungen mittels GROUP BY bei lediglich 0,5 % aller ausgeführten Abfragen zum Einsatz. Aggregatfunktionen werden in 1,2 % aller ausgeführten Abfragen eingesetzt, wobei der Hauptanteil mit 0,9 % durch COUNT-Funktionen gebildet wird. Andere SPARQL-Funktionen wie Unterabfragen, BIND- oder HAVING-Ausdrücke werden nur in vernachlässigbar kleiner Anzahl genutzt.

Komplexitätsbasierte Analyse der Daten

Die 120 am häufigsten genutzten Abfragen des LSQ-Datensatzes wurden hinsichtlich ihrer Komplexität untersucht. Die Komplexitätswerte wurden sowohl anhand der Komplexitätsme-

[40] http://graphite.ecs.soton.ac.uk/sparqllib/

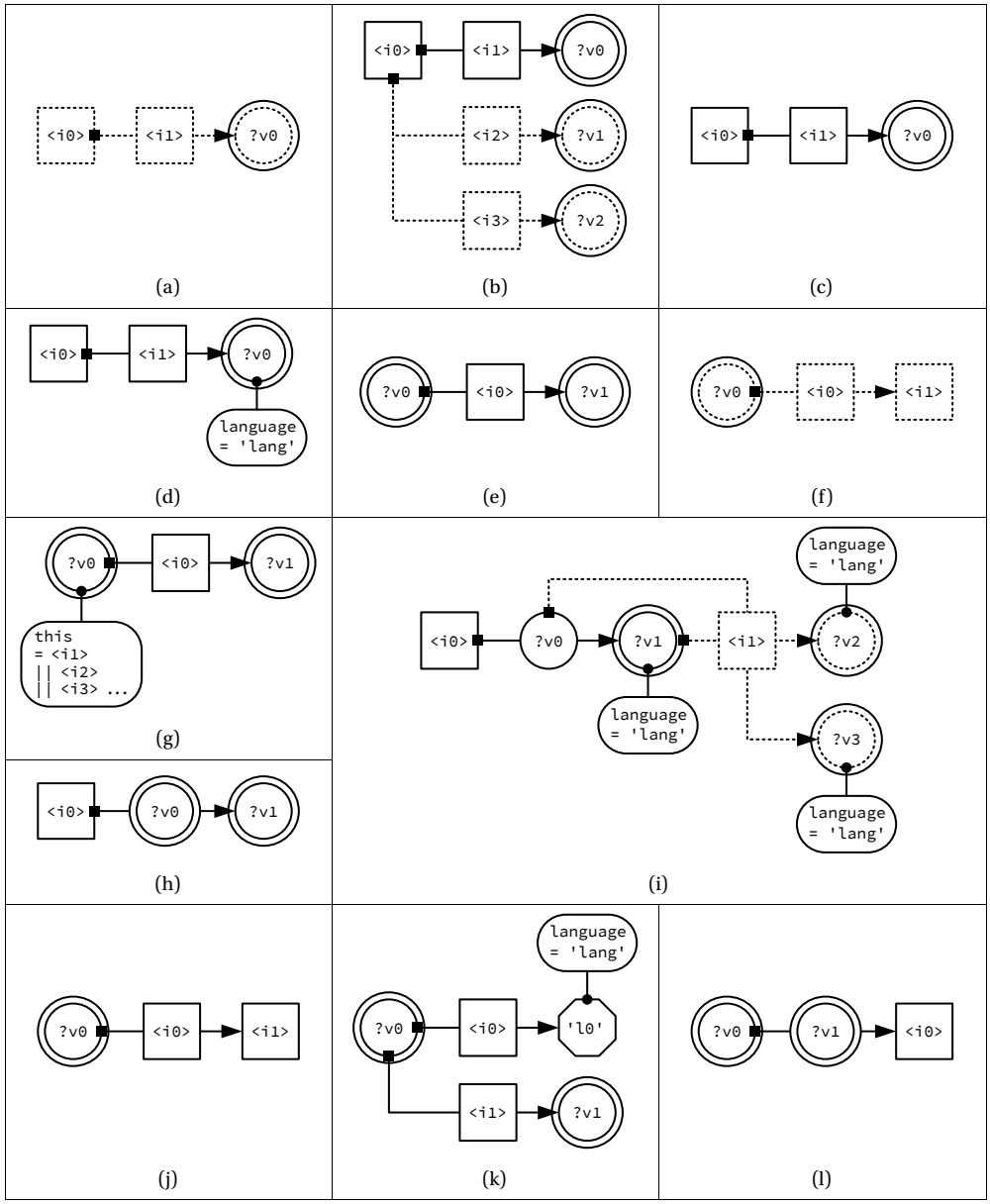


Abbildung 2.13: Visualisierung der zwölf am häufigsten vorkommenden Abfrage-Patterns des LSQ-Datensatzes. IRIs werden als Quadrate dargestellt, Variablen als Kreise und Literale als Achtecke. Variablen, die Teil des SELECT-Statements sind, werden durch einen doppelten Rahmen hervorgehoben. Ein Triple-Pattern wird durch einen Pfeil repräsentiert. Triple-Patterns, die OPTIONAL sind, werden mit gestrichelten Rahmen gezeichnet. FILTER-Ausdrücke werden innerhalb eines Rechtecks mit abgerundeten Kannten beschrieben und mit dem einzuschränkenden Element verbunden. Beispielsweise wird das Abfrage-Pattern `SELECT ?v0 ?v1 WHERE { ?v0 <i0> 'l0'@lang ; <i1> ?v1 }` durch (k) visualisiert.

Tabelle 2.8: Rangliste der zwölf am häufigsten genutzten Abfrage-Patterns des LSQ-Datensatzes. Diese decken über 85 % aller Abfragen ab.

	Gesamt	DBpedia	LGD	SWDF	Abfrage-Pattern
(a)	1	-	1	-	`SELECT ?v0 WHERE { OPTIONAL { <i0> <i1> ?v0 } }`
(b)	2	1	-	-	`SELECT ?v0 ?v1 ?v2 WHERE { <i0> <i1> ?v0 OPTIONAL { <i0> <i2> ?v1 ; <i3> ?v2 } }`
(c)	3	2	4	54	`SELECT ?v0 WHERE { <i0> <i1> ?v0 }`
(d)	4	3	128	-	`SELECT ?v0 WHERE { <i0> <i1> ?v0 FILTER langMatches(lang(?v0) , "l0") }`
(e)	5	31	2	20	`SELECT ?v0 ?v1 WHERE { ?v0 <i0> ?v1 }`
(f)	6	-	3	-	`SELECT ?v0 WHERE { OPTIONAL { ?v0 <i0> <i1> } }`
(g)	7	-	5	-	`SELECT ?v0 ?v1 WHERE { ?v0 <i0> ?v1 FILTER (?v0 = <i1> || ?v0 = <i2> || ?v0 = <i3> || ?v0 = <i4> || ?v0 = <i5>) }`
(h)	8	5	19	1	`SELECT ?v0 ?v1 WHERE { <i0> ?v0 ?v1 }`
(i)	9	4	-	-	`SELECT ?v3 ?v0 ?v1 ?v2 WHERE { <i0> ?v0 ?v1 OPTIONAL { ?v1 <i1> ?v2 } OPTIONAL { ?v0 <i1> ?v3 } FILTER ((langMatches(lang(?v3) , "l0") || !langMatches(lang(?v3), "*")) && (langMatches(lang(?v1), "l0") || !langMatches(lang(?v1), "*")) && (langMatches(lang(?v2), "l0") || !langMatches(lang(?v2), "*"))) }`
(j)	10	6	50	-	`SELECT ?v0 WHERE { ?v0 <i0> <i1> }`
(k)	11	7	80	-	`SELECT ?v0 ?v1 WHERE { ?v0 <i0> "l0"@lang ; <i1> ?v1 }`
(l)	12	8	-	-	`SELECT ?v0 ?v1 WHERE { ?v0 ?v1 <i0> }`

trik nach Halstead (1977) wie auch anhand der eigenen, in Abschnitt 2.4.2 eingeführten, SSE-basierten Komplexitätsmetrik bestimmt. Der Großteil der Abfragen des LSQ-Datensatzes kann als gering bis mittel komplex[41] angesehen werden.

Etwa ein Drittel (31 %) der gestellten Abfragen besitzt einen c-Wert zwischen 1 und 3, wodurch diese als gering komplex angesehen werden können. Nach der Halstead-Metrik betrifft dies sogar mehr als die Hälfte (56 %) der Abfragen. Über die Hälfte (58 %) der Abfragen besitzt einen c-Wert zwischen 4 und 6, wodurch diese als mittel komplex angesehen werden können. Nach der Halstead-Metrik betrifft dies etwa ein Viertel (24 %) der Abfragen. Eine hohe, sehr hohe oder außergewöhnlich hohe Komplexität nach der SSE-basierten Komplexitätsmetrik besitzen zusammen weniger als 9 % der gestellten Abfragen. Nach der Halstead-Metrik trifft dies auf 18 % der Abfragen zu. Die genauen Werte werden in Tabelle 2.9 dargestellt. Die allermeis-

[41] Die Bewertung der Komplexitätswerte basiert auf einer persönlichen Einschätzung. Geringe Komplexität: 1 – 3; Mittlere Komplexität: 4 – 6; Hohe Komplexität: 7 – 9; Sehr hohe Komplexität: 10 – 20; Außergewöhnlich hohe Komplexität: > 20

Tabelle 2.9: Übersicht über die berechneten Komplexitätswerte der 120 am häufigsten ausgeführten Abfrage-Patterns und deren Anteil an den insgesamt gestellten Abfragen des LSQ-Datensatzes, berechnet auf Basis zwei verschiedener Komplexitätsmetriken.

	SSE-basiertes c		Halstead D	
Komplexität	Anzahl[a]	Anteil[b]	Anzahl[a]	Anteil[b]
1–3	18	0,305	17	0,564
4–6	37	0,577	54	0,243
7–9	23	0,040	19	0,128
10–20	25	0,041	28	0,034
>20	17	0,027	2	0,020

[a] (absolute Anzahl, basierend auf den 120 am häufigsten genutzten Abfrage-Patterns)
[b] (relativer Anteil, bezogen auf sämtliche gestellten Abfragen des Datensatzes)

Listing 2.16: Zwei semantisch äquivalente Abfrage-Patterns. Während das erste Pattern mehrere kleinere Ergebnismengen miteinander zu einem Endergebnis vereint, erzeugt das zweite Pattern zuerst eine größere Ergebnismenge, die durch mehrere Filterausdrücke wieder eingeschränkt wird. Letztendlich erzeugen beide Abfrage-Patterns dasselbe Endergebnis.

```
1  SELECT ?v0 WHERE {
2    { ?v0 <u0> <u1> } UNION { ?v0 <u0> <u2> } UNION { ?v0 <u0> <u3> }
3  }
```

```
4  SELECT ?v0 WHERE {
5    ?v0 <u0> ?v1 . FILTER ( ?v1 = <u1> || ?v1 = <u2> || ?v1 = <u3> )
6  }
```

ten der zwölf am häufigsten genutzten Patterns lassen sich nach beiden Metriken als gering bis mittel komplex einstufen.

2.5.2 Analyse der Ausdrucksfähigkeit von SemwidgQL

Aufbauend auf der zuvor durchgeführten Analyse des LSQ-Datensatzes, wurde untersucht, welche SPARQL-Abfragen, die in der Praxis genutzt wurden, auch durch SemwidgQL ausgedrückt werden können. Dazu wurden die 120 am häufigsten genutzten Abfrage-Patterns, die zusammen über 99 % der insgesamt ausgeführten Abfragen abdecken, genauer analysiert. Von diesen 120 Patterns können 66, die bereits 91,3 % der insgesamt ausgeführten Abfragen repräsentieren, direkt und ohne Limitationen in SemwidgQL ausgedrückt werden. Im Gegensatz dazu können 15 dieser Patterns, welche jedoch nur 1,9 % der ausgeführten Abfragen repräsentieren, nicht in SemwidgQL ausgedrückt werden. Diese Abfrage-Patterns enthalten GROUP-BY-Ausdrücke oder Funktionsaufrufe wie `bound` oder `isLiteral`, welche von SemwidgQL nicht unterstützt werden. Die restlichen 39 Patterns, welche 5,8 % der ausgeführten Abfragen repräsentieren, enthalten UNION-Ausdrücke, für welche SemwidgQL kein Äquivalent bietet. Jedoch können einige dieser Abfragen auch ohne UNION-Ausdrücke das selbe Ergebnis erzeugen, indem diese durch FILTER-Ausdrücke ersetzt werden. Zum Beispiel sind die beiden Abfrage-Patterns aus Listing 2.16 semantisch äquivalent.

Tabelle 2.10: Übersetzung der 12 am häufigsten genutzten Abfrage-Patterns des LSQ-Datensatzes in SemwidgQL.

	SemwidgQL-Abfrage
(a)	`<i0>.<i1>(@optional = true)`
(b)	`<i0>.[<i1>, <i2>(@optional = true), <i3>(@optional = true)]`
(c)	`<i0>.<i1>`
(d)	`<i0>.<i1>(@lang = 'lang')`
(e)	`*.<i0>`
(f)	`<i1>.^<i0>(@optional = true)`
(g)	`*(@self = <i1> || @self = <i2> || @self = <i3> || @self = <i4> || @self = <i5>).<i0>`
(h)	`<i0>.*(@predicate = true)`
(i)	`<i0>.[*(@lang = 'lang').<i1>(@lang = 'lang' && @optional = true)), <i1>(@lang = 'lang' && @optional = true)]`
(j)	`<i1>.^<i0>`
(k)	`*.[<i0>(@self = 'l0' && @lang = 'lang'), <i1>]`
(l)	`<i0>.^*(@predicate = true)`

Sollte sich jedoch die Position der variierenden Resource innerhalb des UNION-Ausdrucks oder das Prädikat ändern, ist nicht sichergestellt, das eine UNION-freie Alternative zu der Abfrage existiert. Eine weitere Möglichkeit, neben der Nutzung von FILTER-Ausdrücken, bietet die Fähigkeit von SemwidgQL, mehrere Abfragen in einem einzelnen Ausdruck zu bündeln. Diese Abfragen werden in separate SPARQL-Abfragen übersetzt. Das Zusammenführen der Ergebnisse allerdings ist dem verarbeitenden Programm überlassen. Einige diese UNION-Abfragen können auf diese Weise in SemwidgQL realisiert werden.

Werden die Daten nicht im Gesamten, sondern Endpoint-spezifisch untersucht, zeigen sich zwischen diesen einige Unterschiede. Werden die Abfragen betrachtet, die an den DBpedia-Endpoint gestellten wurden, zeigt sich, dass sich prozentual etwas mehr ausgeführte Abfragen direkt in SemwidgQL ausdrücken lassen. Insgesamt lassen sich 29 Patterns, welche 92,6 % der ausgeführten Abfragen repräsentieren, direkt und ohne Einschränkung in SemwidgQL formulieren. 19 Patterns, welche 4,6 % der ausgeführten Abfragen repräsentieren, können nur teilweise beziehungsweise nach einer syntaktischen Umformulierung in SemwidgQL ausgedrückt werden. Lediglich 4 Patterns, welche nur 1,8 % der ausgeführten Abfragen repräsentieren, können gar nicht in SemwidgQL ausgedrückt werden.

Die Ergebnisse für die Abfragen des SWDF-Endpoints sind hinsichtlich ihrer Überführbarkeit in SemwidgQL hingegen deutlich schlechter. Zu den häufig genutzten Abfragen dieses Endpoints zählen die Abfragen der PHP-Bibliothek zur Überprüfung des Funktionsumfangs von SPARQL-Endpoints. Diese allein machen 16,5 % der ausgeführten Abfragen aus und können nicht in SemwidgQL ausgedrückt werden. Insgesamt können 21 Abfrage-Patterns, welche 31,5 % der Abfragen repräsentieren, nicht in SemwidgQL ausgedrückt werden. Immerhin 34 Patterns, welche 51,2 % der Abfragen repräsentieren, können noch direkt und 7 Patterns, welche 16,6 % der Abfragen repräsentieren, können teilweise in SemwidgQL abgebildet werden.

Für die Abfragen des LGD-Endpoints ergaben sich nahezu optimale Ergebnisse. Fast alle Patterns, welche zusammen 94,2 % der ausgeführten Abfragen repräsentieren, können direkt in SemwidgQL reproduziert werden. Weitere 12 Patterns, die die restlichen 4,8 % der Abfragen repräsentieren, können teilweise und nach einer syntaktischen Anpassung in SemwidgQL ausgedrückt werden.

Während einige selten genutzte Funktionen von SPARQL in SemwidgQL nicht unterstützt werden, ermöglicht es SemwidgQL andere häufig genutzte Funktionen vereinfacht zu formulieren. So wird die Filterung von Literalen nach einer bestimmten Sprache in 14,8 % der ausgeführten Abfragen vorgenommen, wofür in SemwidgQL ein vereinfachtes Filter-Schlüsselwort existiert. In Tabelle 2.10 werden die Übersetzungen der 12 am häufigsten ausgeführten Abfrage-Patterns des LSQ-Datensatzes in SemwidgQL dargestellt.

2.5.3 Diskussion der Ergebnisse

Die Pattern-basierte Analyse des LSQ-Datensatzes hat gezeigt, dass sich über 90 % der darin enthaltenen Abfragen direkt mit SemwidgQL abbilden lassen. Ein weiterer Teil kann zusätzlich nach einer syntaktischen Umstrukturierung der Abfragen mittels SemwidgQL ausgedrückt werden. Lediglich ein geringer Prozentsatz der Abfragen kann nicht als SemwidgQL-Abfrage formuliert werden.

Da Abfragen in SemwidgQL aufgrund der durchgängig geschriebenen Pfadstruktur ab einer gewissen Länge unweigerlich unübersichtlich werden, sind diese vorrangig für Abfragen mit geringer oder mittlerer Komplexität geeignet. Die Daten der komplexitätsbasierten Analyse haben gezeigt, dass in der Praxis insbesondere solche verhältnismäßig einfachen Abfragen an die SPARQL-Endpoints gestellt wurden. Es kann daher davon ausgegangen werden, dass SemwidgQL in den meisten Fällen, die für die angestrebte Zielgruppe relevanten Einsatzzwecke zufriedenstellend erfüllen kann.

2.6 Diskussion und Fazit

In diesem Kapitel wurde mit SemwidgQL eine Abfragesprache für RDF-Daten vorgestellt, die sich zu SPARQL transkompilieren lässt und sich somit zur Abfrage von Daten von einer Vielzahl öffentlicher SPARQL-Endpoints »out of the box« nutzen lässt.

Aufbauend auf einer Analyse des aktuellen Stands der Forschung im Bezug auf Abfragesprachen für den Linked-Data-Bereich (vgl. Abschnitt 2.1), wurde SemwidgQL als Pfadabfragesprache konzipiert, die einer Reihe von Anforderungen genügen sollte (vgl. Abschnitt 2.2.2), welche der Zielgruppe des Semwidg-Projekts einen leichten Einstieg in die Nutzung von Daten des Linked-Data-Bereichs ermöglichen sollte. Diese Zielgruppe wird hauptsächlich durch Programmierer und Webentwickler mit Programmierkenntnissen gebildet. Dementsprechend soll sich SemwidgQL wie eine objektorientierte Programmiersprache nutzen lassen, mit dessen Konzepten die Nutzer vertraut sind. Ebenso haben Effektivität, Effizienz und leichte Erlernbarkeit der Abfragesprache bei der Konzeption einen hohen Stellenwert eingenommen.

Neben recht einfachen Grundfunktionalitäten (vgl. Abschnitt 2.3.2), erweitern eine Reihen von weiteren Funktionen SemwidgQL (vgl. Abschnitt 2.3.3) und ermöglichen die Abfrage von Daten, für die sonst sehr komplexe SPARQL-Abfragen benötigt werden würden. Es wurden zwei Referenzimplementierungen von SemwidgQL in JavaScript und Java entwickelt, deren Quelltext auf der Semwidg-Projektseite[42] unter der MIT-Open-Source-Lizenz veröffentlicht wurden und dort allen Interessenten vollumfänglich zur Verfügung steht. Neben der Funktionsbeschreibung der Sprache[43] (vgl. Abschnitt 2.3) finden sich dort auch die EBNF-Regeln[44], welche die Syntax von SemwidgQL beschreiben, Pseudocode[45], der beschreibt, wie sich aus dem Syntaxbaum einer SemwidgQL-Abfrage eine SPARQL-Abfrage ableiten lässt und Übersetzungsbeispiele[46] von SemwidgQL nach SPARQL. All diese Informationen sollten es Nutzern ermöglichen SemwidgQL einzusetzen und gegebenenfalls sogar für weitere Programmiersprachen verfügbar zu machen. In dieser Hinsicht ist SemwidgQL momentan einzigartig im Linked-Data-Bereich. Es konnte keine weitere Abfragesprache gefunden werden, die ebenfalls so formal eindeutig spezifiziert wurde wie SemwidgQL und somit eine einfache Portierung für weitere Programmiersprachen erlaubt.

Eine empirische Nutzerstudie (vgl. Abschnitt 2.4), in der untersucht werden sollte, wie gut die drei Hauptziele Effektivität, Effizienz und leichte Erlernbarkeit bei der Entwicklung von SemwidgQL umgesetzt wurden, konnte zeigen, dass die Probanden bei der Nutzung von SemwidgQL genauso effektiv waren wie bei der Nutzung von SPARQL. Gleichzeitig waren sie deutlich effizienter bei der Nutzung von SemwidgQL als bei der Nutzung von SPARQL. Die Ergebnisse einer Wiederholung der Studie lassen zudem darauf schließen, dass SemwidgQL leichter zu erlernen ist als SPARQL. Nicht nur die objektiven Messwerte sprechen für SemwidgQL, sondern auch die subjektiven Bewertungen seitens der Probanden. Vier Fünftel der Probanden präferierten explizit SemwidgQL als Abfragesprache gegenüber SPARQL. Als Nebenprodukt der Analyse, wurde eine objektive Metrik zur Beschreibung der Komplexität von SPARQL-Abfragen entworfen, welches deutlich zufriedenstellendere Ergebnisse liefern konnte als bisher genutzte Metriken. Die Ergebnisse der Nutzerstudie und der neu entwickelten Metrik wurden im Rahmen des *Research Tracks* auf der »16th International Semantic Web Conference (ISWC 2017)« vorgestellt und in Stegemann & Ziegler (2017a) als Full Paper veröffentlicht. Die Acceptance

[42] https://semwidg.org/page/download-semwidgql
[43] https://semwidg.org/page/semwidgql
[44] https://semwidg.org/page/semwidgql-ebnf
[45] https://semwidg.org/page/semwidgql-pseudo-code
[46] https://semwidg.org/page/semwidgql-translation-examples

Rate des Research Tracks lag bei 22 % (44 von 197). Die Komplexitätsmetrik für Linked-Data-Abfragen sowie die empirische Nutzerstudie, in der zwei Abfragesprachen des Linked-Data-Bereichs miteinander verglichen wurden, sind in diesem Forschungsbereich jeweils ein Novum.

Eine zusätzliche Untersuchung der Ausdrucksfähigkeit von SemwidgQL (vgl. Abschnitt 2.5), basierend auf einer Pattern-basierten Analyse des LSQ-Datensatzes, konnte zeigen, dass sich über 90 % der darin enthalten 1,75 Millionen unterschiedlichen Abfragen auch in SemwidgQL abbilden lassen. Die Abfragen stammen aus den Log-Dateien von großen öffentlichen SPARQL-Endpoints wie DBpedia. Die Komplexität der Abfragen des Datensatzes wurden ebenfalls anhand der neu entwickelten Metrik untersucht. Dabei zeigte sich, dass diese vorwiegend eine geringe oder mittlere Komplexität aufweisen. Auch wenn die Daten des LSQ-Datensatzes nicht repräsentativ hinsichtlich des allgemeinen Einsatzes von SPARQL sind, wie zum Beispiel dem Einsatz im geschäftlichen Umfeld, bieten sie dennoch einen Einblick in die Nutzung der SPARQL-Endpoints, die für die Zielgruppe des Semwidg-Projekts besonders relevant sind. Die Ergebnisse der Untersuchung lassen darauf schließen, dass SemwidgQL in den meisten Fällen, die für die angestrebte Zielgruppe relevanten Einsatzzwecke bedienen kann. Die Ergebnisse der Pattern-basierten Analyse des LSQ-Datensatzes wurden im Rahmen der *Posters and Demos Session* auf der »16$^{\text{th}}$ International Semantic Web Conference (ISWC 2017)« vorgestellt und in Stegemann & Ziegler (2017a) veröffentlicht. Auch diese Pattern-basierten Analyse ist im hier vorliegenden Forschungsbereich ein Novum.

Trotz der guten Ergebnisse der beiden Studien, sollte SemwidgQL nicht als Ersatz für SPARQL verstanden werden. Auch wenn die Analyse des LSQ-Datensatzes zeigen konnte, dass sich die allermeisten der darin enthaltenen Abfragen mit SemwidgQL abbilden lassen, ermöglicht SemwidgQL dennoch nur die Formulierung eines Bruchteils der Abfragen, welche sich mit SPARQL formulieren lassen. Stattdessen soll SemwidgQL eine leichtgewichtige Abfragesprache sein, die die Einstiegshürden für die Nutzung des Semantic Webs und des Linked-Data-Bereichs senken soll. Die zuvor beschrieben Studien lassen darauf schließen, dass SemwidgQL diesen Anforderungen generell gewachsen ist.

KAPITEL 3

SemwidgJS – Eine Widget-Bibliothek und ein Präsentations-Framework für Linked Open Data

Zahlreiche Frameworks und Systeme erlauben die Einbindung von RDF-Daten in reguläre Webseiten. Ein Großteil dieser baut selbst wieder auf Protokollen, Formaten und Techniken des Linked-Data-Bereichs auf und ist somit für den Großteil der potenziellen Nutzer nur mit erhöhtem Einarbeitungsaufwand anwendbar. Andere Frameworks und Systeme sind so stark auf bestimmte Einsatzgebiete spezialisiert, dass eine anderweitige Nutzung kaum noch möglich ist. Wiederum andere stellen so hohe Anforderungen an die benötigte Serverinfrastruktur oder das dort eingesetzte Software-Ökosystem, dass sie mit den bisher genutzten Systemen nicht kompatibel sind. Eine Möglichkeit für Webseitenautoren ohne fortgeschrittenen Kenntnisse über die Techniken und Anwendungen des Linked-Data-Bereichs, um beliebige Daten aus SPARQL-Endpoints abzurufen, aufzubereiten und in Form ansprechender Präsentationen und Visualisierungen in Webseiten einzubinden, existierte bis zu Entwicklung von SemwidgJS nicht.

In diesem Kapitel wird mit SemwidgJS eine Widget-Bibliothek und zugleich ein Framework zur Präsentation und Visualisierung von Daten aus öffentlichen SPARQL-Endpoints vorgestellt. Widgets beschreiben vordefinierte Datenpräsentationen oder Visualisierungen, die sich anhand weniger Parameter an die Bedürfnisse der Webseitenautoren anpassen lassen. Statt dass die Datenabfrage, die Verarbeitung der Antwort des Datenbankservers und die Darstellung der Daten von den Autoren selbst programmiert werden müssen, müssen diese lediglich die entsprechenden Parameter an das Widget übergeben. Die SemwidgJS-Widget-Bibliothek stellt eine Vielzahl von Widgets für die Ein- und Ausgabe der Daten in Form üblicher Web-UI-Elemente, sowie zur Visualisierung von numerischen Daten in Form von Diagrammen bereit. Zugleich kann SemwidgJS als Framework zur Erzeugung benutzerdefinierter und an die anwendungsspezifischen Anforderungen angepasster Widgets genutzt werden. Die Widgets können untereinander Daten austauschen oder von außen manipuliert werden, sodass sich mit geringem Aufwand auch interaktive Webseiten erzeugen lassen.

SemwidgJS wurde auf der Semwidg-Projektseite[1] veröffentlicht, und steht dort allen Interessenten vollumfänglich zur Verfügung. Die JavaScript-Bibliothek, die sämtliche vordefinierten Widgets und die Möglichkeit zur Verarbeitung von SPARQL- und SemwidgQL-Abfragen ent-

[1] `https://semwidg.org/page/semwidgjs`

hält, kann dort heruntergeladen oder von dort aus auch direkt in eigene Webseiten eingebettet werden[2]. Weiterhin finden sich auf der Projektseite eine Dokumentation der Bibliothek[3], eine Übersicht der vordefinierten Widgets[4] sowie Quelltextbeispiele[5], die potenzielle Nutzer beim Einsatz von SemwidgJS unterstützen sollen.

Zu Beginn dieses Kapitels wird ein Überblick über den aktuellen Stand der Forschung hinsichtlich verschiedener Frameworks und Systeme zur Präsentation und Visualisierung von Daten aus dem Linked-Data-Bereich gegeben (vgl. Abschnitt 3.1). Anschließend werden die konzeptionellen Überlegungen zu SemwidgJS beschrieben (vgl. Abschnitt 3.2). Dazu wurden anhand der potenziellen Nutzergruppen und des zuvor analysierten Stands der Forschung verschiedene Anforderungen ermittelt. Aufbauend auf diesem Konzept wurde SemwidgJS entwickelt, dessen Implementierung nachfolgend beschrieben wird (vgl. Abschnitt 3.3). SemwidgJS kam im Rahmen des Eurostars-geförderten Projekts DESUMA zum Einsatz und konnte dort seine Nützlichkeit unter Beweis stellen. Das Projekt und der Einsatz von SemwidgJS in diesem wird dahingehend vorgestellt (vgl. Abschnitt 3.4). Abschließend werden die Ergebnisse dieses Kapitels diskutiert und ein Fazit gezogen (vgl. Abschnitt 3.5).

[2] https://semwidg.org/semwidg.js
[3] https://semwidg.org/page/semwidgjs-docs
[4] https://semwidg.org/page/semwidgjs-widgets
[5] https://semwidg.org/page/semwidgjs-example

3.1 Stand der Forschung

Nachfolgend wird ein Überblick über verschiedene Frameworks und Systeme zur Präsentation und Visualisierung von Daten aus dem Linked-Data-Bereich gegeben. Diese erlauben es Webentwicklern auf unterschiedliche Weise RDF-Daten einzulesen, zu verarbeiten und in eine für den Endnutzer verständliche Form zu überführen. In der Regel ist dies eine Webseite, die Werte enthalten soll, die unmittelbar aus der RDF-Datei entnommen werden oder Visualisierungen wie Diagramme oder Karten, die aus den RDF-Daten generiert werden.

Der hier gegebene Überblick beschränkt sich auf Frameworks und Systeme, die es Nutzern erlaubt, ihre Präsentationen nach ihren Vorstellungen frei anzupassen, soweit die Limitationen des Frameworks oder Systems dies ermöglichen. Systeme, die ausschließlich auf eine Visualisierungsmethode beschränkt sind (in der Regel Graphdarstellungen, z. B. STOOG, vgl. Artignan & Hascoët, 2010), werden nicht betrachtet.

Um die verschiedenen Systeme einzuordnen, soll im weiteren zwischen drei Arten von Verfahren zur Generierung von Präsentationen und Visualisierungen unterschieden werden, die im Folgenden definiert werden. Im Anschluss daran werden beispielhaft einige Arbeiten, die nach diesen Verfahren arbeiten, vorgestellt und analysiert.

Template-basierte Verfahren: Diese Verfahren lassen Entwickler ein Grundgerüst für die Datenpräsentation erstellen, in denen Platzhalter definiert werden können, welche anschließend durch Werte aus den RDF-Daten ersetzte werden. In den hier untersuchten Systemen besteht das Grundgerüst in der Regel aus HTML-Quelltext.

Transformationsbasierte Verfahren: Diese Verfahren erfordern, dass der Entwickler eine Reihe von Transformationsregeln definiert, die beschreiben, welche Daten auf welche Weise aufbereitet und präsentiert werden sollen. In diesen wird im Unterschied zu den Templates der Template-basierten Verfahren also kein Grundgerüst in der Ausgabesprache gespeichert. Häufig können innerhalb dieser Verfahren auch wieder Template-Elemente genutzt werden, mit denen sich definierte Transformationsregeln wiederverwenden lassen.

Algorithmische Verfahren: Diese Verfahren geben Entwicklern die größte Freiheit bei der Erstellung von Präsentationen und Visualisierungen, erfordern für die Nutzung jedoch auch fortgeschrittenere Programmierkenntnisse als die zuvor genannten Verfahren. Die RDF-Daten können mit der Programmiersprache des entsprechenden Frameworks beliebig verarbeitet werden. Das Framework selbst kümmert sich dabei um die Abfrage der Daten und stellt dem Entwickler Funktionen zur Verfügung, die ihm die Erstellung einer Webseite erleichtern.

3.1.1 Template-basierte Verfahren

VPOET (vgl. Rico et al., 2008) ist eine auf dem Fortuna Framework (vgl. Rico et al., 2010) basierende Anwendung, die es Webentwicklern erlaubt HTML-Templates in einer Wiki-Umgebung zu erstellen. Die Templates werden mit Daten aus RDF-Dokumenten gefüllt, deren URLs der Anwendung als Parameter übergeben werden. Um die darzustellenden Eigenschaften auszuwählen, können einige wenige Funktionen (von den Autoren »Macros« genannt) genutzt werden. Abfragen mittels einer vollwertigen Abfragesprache sind somit nicht nötig aber auch nicht möglich. Dies vereinfacht zwar die Erstellung einfacher Templates, verhindert aber auch, dass erfahrenere Nutzer bei Bedarf komplexere Templates erstellen können.

TAL4RDF (vgl. Champin, 2009) basiert auf der Template Attribute Language (TAL)[6]. TAL-Templates sind XML-Dokumente, die in andere XML-konforme Dokumente (HTML, SVG etc.) transformiert werden können. Über Umwege können aber auch beliebige Textdokumente erzeugen werden. TAL4RDF fügt TAL die Möglichkeit hinzu, Template-Platzhalter mit Daten aus RDF-Dokumenten zu belegen. Dazu wird eine minimalistische Pfadabfragesprache genutzt, die jedoch über keine Filtermöglichkeiten verfügt. Die Ergebnisse der Abfragen können in Variablen gespeichert und in nachfolgenden Abfragen wiederverwendet werden. Verschiedene TAL-Elemente ermöglichen die Nutzung bedingter Abfragen und Schleifen.

Auch wenn OWL-PL (vgl. Brophy, 2010) syntaktisch große Ähnlichkeiten zur Transformationssprache XSLT aufweist, werden in OWL-PL Templates definiert, die es erlauben, RDF-Daten in Webseiten darzustellen. Diese Templates werden in eine Ontologie eingebunden, die für die entsprechenden RDF-Daten das passende Template zurückliefert. Templates können entweder für alle Daten, Daten einer Klasse oder eine bestimmte Instanz entworfen werden. Auch in dieser Sprache besteht die Möglichkeit bedingte Abfragen und Schleifen zu nutzen.

Mit LESS (vgl. Auer et al., 2010) steht eine komplette Anwendung zur Erstellung von Templates für Webseiten, deren Verarbeitung und deren Einbinden in andere Medien sowie zu deren Austausch zwischen unterschiedlichen Nutzern zur Verfügung. Die Templates selbst werden in der eigens entwickelten deklarativen Sprache LeTL (LESS Template Language) beschrieben, welche auf der Smarty Template Language[7] basiert. Dabei können die zu verarbeitenden Daten direkt aus einem RDF-Dokument ausgelesen oder mittels SPARQL abgefragt werden. Template und Datenbankabfrage beziehungsweise URL des RDF-Dokuments werden in LESS unabhängig voneinander gespeichert. Innerhalb des Templates kann anhand der Variablenbezeichnungen des Abfrageergebnisses beziehungsweise der genutzten Eigenschaften des RDF-Dokuments direkt auf deren Werte zugegriffen werden. Wenn RDF-Dokumente als Datenquelle genutzt werden, können diese über eine einfache Pfadabfragesprache durchsucht werden. Erstellte Templates werden innerhalb der Anwendung gespeichert und können mit anderen Nutzern geteilt werden. Mittels JavaScript oder eines IFrames lassen sich die Templates in beliebige Webseiten einbetten. LESS verfügt zudem über einen Editor zur Erstellung von Templates (vgl. Abschnitt 4.1.1). Listing 3.1 zeigt ein Beispiel eines LESS-Templates, das für alle Nobelpreisträger für Physik, die in DBpedia gespeichert sind, eine HTML-Tabelle mit Namen, Kurzbeschreibung, Bild und Namen des Geburtsorts erzeugt.

Das Callimachus-Projekt (vgl. Battle et al., 2012) stellt eine Webplattform dar, die Templates auf Basis von RDFa[8] verarbeiten kann. Die Templates entsprechen bis auf einzelne Variablenbezeichnungen einem gültigen HTML-Dokument, welches mittels RDFa annotiert wurde. Variablen kommen immer dann zum Einsatz, wenn der darzustellende Datensatz für eine Eigenschaft mehrere Werte aufweisen kann, über die iteriert werden soll. Werden zusätzlich zu den RDFa-Annotationen statische Werte angegeben, dienen diese als Filterwerte. Aus dem Template wird eine SPARQL-Abfrage generiert und diese Ergebnisse werden in den Template-Code eingefügt und als HTML-Dokument ausgeliefert.

Ein großer Vorteil dieser Template-basierten Systeme zum Darstellung von RDF-Daten in HTML-Dokumenten ist es, dass das erstellte Template der erzeugten HTML-Ausgabe sehr ähnlich ist. Nutzer, die in der Lage sind, HTML-Quelltext zu erstellen, werden auch schnell lernen, diese Template-Sprachen zu verwenden. In den Systemen kann entweder eine einfache Pfadabfragesprache oder eine Menge von Funktionen zur Auswahl der darzustellenden Eigenschaf-

[6] https://zope.readthedocs.io/en/latest/zope2book/AppendixC.html
[7] https://www.smarty.net
[8] https://www.w3.org/TR/rdfa-core/

Listing 3.1: SPARQL-Abfrage und Template-Code eines LESS-Templates. Die Ergebnisse der oben gezeigten SPARQL-Abfrage, die an den DBpedia-Endpoint gestellt werden kann, werden anhand der Variablenbezeichnungen ausgewählt und in den unten dargestellten Template-Code eingefügt. Die Abfrage fragt Daten zu allen Nobelpreisträgern für Physik ab. Der Template-Code beschreibt eine HTML-Tabelle mit Platzhaltern für den Namen, eine Kurzbeschreibung, ein Bild und den Namen des Geburtsorts. Der Template-Code wird jeweils mit den Daten eines jeden Einzelergebnisses des SPARQL Result Sets befüllt, sodass in diesem Beispiel jeweils eine Tabelle mit Daten zu jedem Nobelpreisträger erzeugt wird (vgl. Abbildung 4.2).

```
1  SELECT DISTINCT ?label ?abstract ?thumbnail ?birthPlace ?placeLabel
2  WHERE {
3    ?physicists rdf:type yago:WikicatNobelLaureatesInPhysics .
4    ?physicists rdfs:label ?label .
5    ?physicists dbo:abstract ?abstract .
6    OPTIONAL { ?physicists dbo:thumbnail ?thumbnail. }
7    OPTIONAL { ?physicists dbo:birthPlace ?birthPlace .
8               ?birthPlace rdfs:label ?placeLabel . }
9    FILTER (lang(?label) = '' || langMatches(lang(?label), 'en'))
10   FILTER (lang(?abstract) = '' || langMatches(lang(?abstract), 'en'))
11   FILTER (lang(?placeLabel) = '' || langMatches(lang(?placeLabel), 'en'))
12 }
```

```
13 <table>
14   <tr>
15     <th colspan="2" style="background-color:#f2f2f4;">{$label}</th>
16   </tr>
17   <tr>
18     <td><div style="height: 150px; overflow: auto;">{$abstract}</div></td>
19     <td><img border="0" width="100" src="{$thumbnail}"></td>
20   </tr>
21   <tr>
22     <td>Birthplace:</td>
23     <td>{$placeLabel}</td>
24   </tr>
25 </table>
```

ten des RDF-Dokuments genutzt werden, sodass nicht einmal eine komplexe Abfragesprache erlernt werden muss. Dies schränkt jedoch auch die Mächtigkeit dieser Systeme ein. Daher bieten LESS und Callimachus zusätzlich noch an, SPARQL direkt für komplexere Abfragen verwenden zu können.

Bis auf LESS müssen alle Systeme direkt auf dem Server ausgeführt werden, welche die Webseiten ausliefern soll. Die von LESS erzeugten Templates können auf einem externen Server liegen und mittel JavaScript in beliebige Webseiten eingebunden werden, ohne direkten Zugriff auf den Server zu haben. Dies ist insbesondere für Nutzer relevant, denen gegebenenfalls ein solcher direkter Zugriff fehlt, da sie beispielsweise nur ein Blog über eine kostenlose Plattform wie WordPress[9] oder Blogger[10] betreiben, welche höchstens das Hochladen von HTML- oder eingeschränktem PHP-Quelltext erlaubt.

Allen Systemen ist jedoch gemein, dass ihnen die Fähigkeit zur algorithmischen Interpretation und Weiterverarbeitung der Daten fehlt. So können zum Beispiel nicht mehrere numeri-

[9] https://wordpress.com
[10] https://blogger.com

sche Daten zusammengerechnet oder ein Durchschnittswert ermittelt werden. Die Erstellung einer komplexen Präsentation der Daten ist somit nicht möglich.

3.1.2 Transformationsbasierte Verfahren

Xenon (vgl. Quan & Karger, 2004b) ist eine auf XSLT[11] basierende Transformationssprache für RDF-Daten. Um diese RDF-Daten besser verarbeiten zu können, wurden grundlegende Bereiche von XSLT angepasst. Dies betrifft unter anderem die Abfragesprache XPath[12], welche durch SPARQL ersetzt wurde. Zur Transformation der RDF-Daten nutzt Xenon die Konzepte von *Lenses* und *Views* (vgl. Quan & Karger, 2004a), von denen erstere beschreiben, welche Informationen dargestellt und letztere, auf welche Art (z. B. als Text, Bild oder Link) diese dargestellt werden sollen. Innerhalb der Views kann HTML-Quelltext enthalten sein, in den die RDF-Daten eingebettet werden sollen. Views können ineinander geschachtelt werden.

Fresnel (vgl. Pietriga et al., 2006) wurde als Nachfolger von Xenon entwickelt. Im Gegensatz zu Xenon sollten sich in Fresnel jedoch keine darstellungsspezifischen Elemente wie HTML-Tags mehr beschreiben lassen, welche die Nutzung der Datenpräsentation auf ein vorbestimmtes Darstellungsparadigma und Ausgabeformat beschränken. Stattdessen sollten durch Fresnel nur noch die Daten, die für eine Präsentation genutzt werden sollen, ermitteln und vorverarbeitet werden. Die eigentliche Darstellung sollte von einem Browser übernommen werden. Unterschiedliche Browser können so auch unterschiedliche Darstellungsparadigmen nutzen. Während zum Beispiel der eine Browser die durch Fresnel ausgewählten Daten als Tabelle anzeigt, kann ein anderer Browser diese Daten als Diagramm formatieren. Dementsprechend ist das Konzept der Views überflüssig geworden. Dieses wurde durch das Konzept der *Formats* ersetzt, welche Daten formatiert und gegebenenfalls mit zusätzlichen Informationen anreichern können. Diese Informationen können von einem Browser genutzt werden, um die Daten mit einer bestimmten CSS-Klasse zu verknüpfen oder, um dem Browser ein nicht verbindliches Darstellungsparadigma vorzuschlagen. Als Abfragesprachen dienen SPARQL und die *Fresnel Selector Language* (FSL, vgl. Abschnitt 2.1.2). Für beide Sprachen erlaubt Fresnel nur die Auswertung einer einzelnen Eigenschaft pro Abfrage. Listing 3.2 zeigt ein Fresnel-Stylesheet, welches mehrere Views und Formats kombiniert. Eine ganzen Reihe von Anwendungen zur Darstellung von RDF-Daten basiert auf Fresnel oder wurde nachträglich erweitert, um Fresnel nutzen zu können. Unter anderem kommt Fresnel in Marbles[13], IsaViz[14], Haystack (vgl. Quan et al., 2003) und Arago (vgl. Gassert & Harth, 2005) zur Anwendung. PRISSMA (vgl. Costabello & Gandon, 2014) erweitert Fresnel zusätzlich um Funktionen, die insbesondere für mobile Endgeräte relevant sind.

Mit X3S (vgl. Stegemann et al., 2011, 2012) wurde ein Verfahren geschaffen, das die Definition wiederverwendbare »Semantic Stylesheets« ermöglicht, die vornehmlich auf Basis bereits zuvor bestehender Techniken verarbeitet werden. Daten werden mittels SPARQL abgerufen, anschließend vorverarbeitet und mittels XSLT in HTML-Quelltext transformiert welcher anschließend mittels CSS formatiert wird. Für die Vorverarbeitung enthält ein X3S-Stylesheet eine hierarchische Strukturvorgabe, auf die die eigentliche flachen Ergebnislisten der SPARQL-Abfrage abgebildet werden, wodurch sich ein Teilbereich des RDF-Graphen als Baum rekonstruieren lässt. Die Definition der »Semantic Stylesheets« kann durch einen Editor (vgl. Abschnitt 4.1.1)

[11] https://www.w3.org/TR/xslt/
[12] https://www.w3.org/TR/xpath/
[13] https://mes.github.io/marbles/
[14] https://www.w3.org/2001/11/IsaViz/

Listing 3.2: Ein Fresnel-Stylesheet. Das Stylesheet zeigt die Namen der Nobelpreisträger in Physik und die Namen ihrer Geburtsorte an. Das Stylesheet besteht aus den zwei Lenses *NobelPrizeLens* und *BirthPlaceLabelLens*, welche für die Auswahl der anzuzeigenden Eigenschaften verantwortlich sind. Die Formats *PhysicistLabelFormat* und *BirthPlaceLabelFormat* definieren zusätzlich, wie die Eigenschaften angezeigt werden sollen.

```
:NobelPrizeLens a fresnel:Lens;
  fresnel:classLensDomain yago:WikicatNobelLaureatesInPhysics;
  fresnel:showProperties (
    rdfs:label
    [rdf:type fresnel:PropertyDescription;
     fresnel:property "dbo:birthPlace[dbpedia-owl:Settlement]"^^fresnel:
         ↪ fslSelector;
     fresnel:sublens :BirthPlaceLabelLens]
  );
  fresnel:group :NobelPrizeMainGroup.

:BirthPlaceLabelLens a fresnel:Lens;
  fresnel:classLensDomain dbpedia-owl:Settlement;
  fresnel:showProperties (
    rdfs:label
  )
  fresnel:group :BirthPlaceSubGroup.

:PhysicistLabelFormat a fresnel:Format;
  fresnel:propertyFormatDomain rdfs:label;
  fresnel:label "Name: ";
  fresnel:group :NobelPrizeMainGroup.

:BirthPlaceLabelFormat a fresnel:Format;
  fresnel:propertyFormatDomain rdfs:label;
  fresnel:label "Geboren in: ";
  fresnel:group :CitySubGroup.

:NobelPrizeMainGroup a fresnel:Group.
:BirthPlaceSubGroup a fresnel:Group.
```

auch von Nutzern ohne großes Hintergrundwissen bezüglich Webprogrammierung und Linked-Data erfolgen. Eine Erstellung eines X3S-Stylesheets ohne diesen Editor ist dagegen aufgrund der zahlreichen Verknüpfungen die zwischen den einzelnen Teilbereichen des Stylesheets bestehen jedoch kaum noch möglich.

STTL (vgl. Corby & Faron-Zucker, 2015) ist einer Erweiterung von SPARQL zur Transformation von RDF-Daten in jedes andere beliebige textbasierte Format. Anstelle des SELECT-Ausdrucks werden Transformationsregeln definiert, in denen die Variablen der Abfrage genutzt werden können. Diese können sich bei Bedarf gegenseitig oder auch rekursiv aufrufen, um so komplexe Ausgaben zu erzeugen. STTL-Abfragen können in reguläre SPARQL-Abfragen übersetzt werden und somit von jedem SPARQL-Endpoint verarbeitet werden. Innerhalb des SELECT-Ausdrucks wird in der SPARQL-Abfrage eine Zeichenkette zusammengesetzt, die die gewünschte Ausgabe enthält. STTL-Abfragen können dementsprechend nur Ausgabe erzeugen, die auch von SPARQL-Abfragen erzeugt werden können, vereinfachen jedoch deren Formulierung.

Die Erstellung von Transformationsregeln ist deutlich aufwendiger als die Erstellung von Templates. Der Nutzer muss in der Regel eine komplett neue Sprache erlernen und kann nicht

einfach seine Zielsprache verwenden und diese um einige Platzhalter und Datenbankabfragen erweitern. Die Erstellung von X3S-Templates ist sogar so komplex, dass sie manuell kaum noch durchgeführt werden kann, sondern einen Editor benötigt, der die Verknüpfung der einzelnen Abschnitte des X3S-Stylesheets untereinander für den Nutzer vornimmt. Ein großer Vorteil der transformationsbasierten Verfahren ist aber auch gerade, dass sie nicht so stark an eine Zielsprache gebunden sind, wie die Template-basierten Verfahren. Ein Fresnel-Stylesheets enthält keinerlei Ausgabeformat-spezifischen Informationen. Das Ausgabeformat wird stattdessen vom verarbeitenden Systeme beziehungsweise dessen Nutzer gewählt.

Xenon und Fresnel benötigen direkten Zugriff auf die Daten und werden daher für gewöhnlich serverseitig betrieben. Die Referenzimplementierung von X3S nutzt den Adobe Flash Player und ist somit direkt für den clientseitigen Einsatz konzipiert worden. Zudem könnte die Verarbeitung von X3S-Stylesheets auch problemlos von einem JavaScript-Programm übernommen werden. Für STTL existiert ein solches JavaScript-Programm bereits[15], sodass die Verarbeitung von STTL-Abfragen ebenfalls clientseitig geschehen kann.

Wie auch schon den Template-basierten Verfahren, fehlen den transformationsbasierten Verfahren die Fähigkeit zur algorithmischen Interpretation und Weiterverarbeitung der Daten. Eine Erstellungen von komplexen Präsentationen ist also auch mit ihnen nicht möglich.

3.1.3 Algorithmische Verfahren

Die Semantic Web Widget Library (vgl. Hollenbach, 2010) und Exhibit (vgl. Huynh et al., 2007) stellen Widgets zur Verfügung, mit denen sich clientseitig mittels JavaScript eingelesene RDF-Daten verarbeiten und präsentieren lassen. Einer vorgefertigten Widget-Klasse werden dabei bestimmte Parameter (z. B. eine Resource-URI oder die URI einer Eigenschaft) übergeben. Das Widget erstellt anhand dieser und den zuvor geladenen Daten eine HTML-Ausgabe. Die Bibliotheken enthalten unter anderem Widget-Klassen für einfache Texte und Bilder aber auch komplexe Karten und Zeittafeln. Beide Bibliotheken können allerdings nur RDF-Dokumente (im Fall von Exhibit auch andere strukturierte Datenformate) einlesen und verarbeiten, jedoch keine Abfragen an SPARQL-Endpoints stellen.

Dies ermöglicht hingegen die JavaScript-Bibliothek Sgvizler (vgl. Skjæveland, 2012), welche die Ergebnisse von SPARQL-Abfragen mittels der Google Chart Tools[16] visualisiert. Die übergebenen Parameter sind hier zum Teil nicht mehr einfache URIs sondern vollständige SPARQL-Abfragen. Die Widgets können entweder direkt im HTML-Quelltext der Webseite oder mittels JavaScript-Code instanziiert werden. Eine nachträgliche Manipulation der Parameter oder ein Austausch von Daten zwischen Widget-Instanzen ist nicht vorgesehen. Abbildung 3.3 zeigt die Instanziierung eines Sgvizler-Map-Widgets, welches auf einer Karte die Geburtsorte aller Nobelpreisträger für Physik anzeigt.

Einen Fokus auf den Datenaustausch zwischen Widgets legt die Serveranwendung Paggr (vgl. Nowack, 2009). Widgets sind hier Teil eines Dashboards. Daten können darin von einem Widget mittels Drag & Drop zu einem anderen Widget als Sucheingabe oder Filterwert übergeben werden. Die Datenabfrage wird in SPARQLScript beschrieben, die Ausgabe eines Widgets erfolgt mittels SPARQL Result Templates (vgl. Nowack, 2008). Nutzer der Plattform haben die Möglichkeit, benutzerdefinierte Widget-Typen zu erstellen.

SWP (SPARQL Web Pages, ursprünglich unter dem Namen UISPIN entwickelt)[17] ist ein deklaratives, RDF-basiertes Framework zur Erstellung von HTML-Seiten, deren darzustellenden

[15] https://github.com/vcharpenay/STTL.js
[16] https://developers.google.com/chart/
[17] https://uispin.org

Listing 3.3: Instanziierung eines Sgvizler-Map-Widgets. Mittels der data-*-Attribute wird der Widget-Instanz eine SPARQL-Abfrage, der entsprechende SPARQL-Endpoint und ein Widget-Typ übergeben. Das Widget erzeugt anhand dieser Daten eine Karte mit Markierungen an den Geburtsorten aller Nobelpreisträger für Physik.

```
<div id="physicistsMap"
  data-sgvizler-query="SELECT DISTINCT ?lat ?long ?label
    WHERE {
      ?physicists rdf:type yago:WikicatNobelLaureatesInPhysics .
      ?physicists rdfs:label ?label .
      ?physicists dbo:birthPlace ?city .
      ?city rdf:type dbo:Settlement .
      ?city geo:lat ?lat .
      ?city geo:long ?long .
      FILTER langMatches(lang(?label), 'en')
    }"
  data-sgvizler-endpoint="https://dbpedia.org/sparql"
  data-sgvizler-chart="google.visualization.Map">
</div>
```

Daten im RDF-Format vorliegen und ist Teil der TopBraid-Produktfamilie[18]. SPARQL Web Pages sind in ihrem Aufbau vergleichbar mit JavaServer Pages[19] und bieten dem Nutzer große Freiheiten bei der Programmierung, verlangt aber auch nach fortgeschrittenen Programmierkenntnissen. Da SPARQL Web Pages selbst wieder RDF-Dokumente darstellen, deren Aufbau extrem komplex ist und sich der HTML- und SPARQL-Quelltext in diesen aufgrund der RDF-Syntaxeinschränkungen nicht direkt abbilden lässt, ist die Programmierung nur unter Zuhilfenahme eines entsprechenden Editor zu bewältigen (vgl. Abschnitt 4.1.1). Zur Verarbeitung und Auslieferung einer SWP-Webseite wird eine spezielle Serversoftware von TopBraid benötigt. Diese verwaltet zudem standardmäßig die anzuzeigenden RDF-Daten. Das Ansprechen externer SPARQL-Endpoints ist jedoch auch möglich. Listing 3.4 zeigt einen Ausschnitt aus dem Quelltext einer SPARQL Web Page. Dieser Ausschnitt definiert den für den Endnutzer sichtbaren Teil der ausgelieferten HTML-Seite und erzeugt eine vergleichbare Anzeige wie das Sgvizler-Map-Widget aus Listing 3.3.

Häufig kommen bei den algorithmischen Verfahren Widgets zum Einsatz, welche eine vorgegebene Datenpräsentation erzeugen. Den Widgets werden dabei mindestens eine SPARQL-Abfrage oder eine Resource-URI als Parameter übergeben, die die anzuzeigenden Daten beschreiben. Über weitere Parameter lässt sich die Darstellung meistens noch weiter anpassen. Diese Widgets lassen sich daher schon von Nutzern ohne fortgeschrittene Programmierkenntnisse einsetzen. Nutzer mit fortgeschrittenen Programmierkenntnisse können zusätzlich benutzerdefinierte Widget-Klassen definieren, und so eigene Datenpräsentationen und -visualisierungen erzeugen. Da die Widget-Klassen mittels einer vollwertigen Programmiersprache definiert werden, können die Daten beliebig interpretiert und weiterverarbeitet werden.

Der Einsatz der Semantic Web Widget Library, Exhibit und Sgvizler ist für Webentwickler und Programmierer ohne großen Aufwand durchführbar, da diese einfach mittels JavaScript in nahezu jede bestehende Webseite einbinden lassen. Paggr und SWP benötigen hingegen spezielle Serveranwendungen.

[18] https://www.topquadrant.com
[19] https://www.oracle.com/java/technologies/jspt.html

Listing 3.4: Quelltextausschnitt einer SPARQL Web Page. Dieser Ausschnitt definiert den für den Endnutzer sichtbaren Teil der ausgelieferten HTML-Seite, welcher innerhalb der SPARQL Web Page unter der Eigenschaft `ui:prototype` gespeichert wird. In diesem Beispiel wird SWP-, JavaScript- und HTML-Quelltext kombiniert, um die Geburtsorte der Nobelpreisträger in Physik auf eine Karte darzustellen. Dazu wird hier die Open-Source-Bibliothek Leaflet genutzt, die außerhalb des hier gezeigten Ausschnitts in die Seite eingebunden wird. Innerhalb des JavaScript-Quelltextes wird eine SWP-Funktion deklariert, die eine SPARQL-Abfrage ausführt und deren Ergebnisliste durchläuft (Zeilen 8–22). Für jedes Ergebnis wird eine Zeile JavaScript-Quelltext generiert, in der die Ergebnisdaten in ein Array geschrieben werden (Zeile 20). An den Endnutzer wird nur der JavaScript- und HTML-Quelltext ausgeliefert.

```
1  <ui:group let:instanceSelectedEvent="org.physicists.instanceSelected">
2      <ui:setContext ui:varName="swaOnOpenResource" ui:varValue="{= swa:
           ↪ openViewFormGadgetWindow(?instanceSelectedEvent) }">
3        <physicists:ApplicationHeader/>
4        <script type="text/javascript">
5          var map = L.map('mapElem');
6          var markers = [];
7
8          <ui:forEach ui:resultSet="{#
9            SELECT DISTINCT ?label ?lat ?long
10           WHERE {
11             ?physicists rdf:type yago:WikicatNobelLaureatesInPhysics .
12             ?physicists rdfs:label ?label .
13             ?physicists dbo:birthPlace ?city .
14             ?city rdf:type dbo:Settlement .
15             ?city geo:lat ?lat .
16             ?city geo:long ?long .
17             FILTER langMatches(lang(?label), 'en')
18           }}">
19
20           markers.push(['{= ?label}', {= ?lat}, {= ?long}]);
21
22         </ui:forEach>
23
24         L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png').addTo(map);
25         map.setView([markers[0][1],markers[0][2]], 5);
26         for (var i = 0; i < markers.length; i++) {
27           L.marker([markers[i][1],markers[i][2]])
28             .bindPopup(markers[i][0]).addTo(map);
29         }
30       </script>
31       <div id="mapElem" style="height: 400px;"></div>
32     </ui:setContext>
33 </ui:group>
```

3.1.4 Zusammenfassung und Fazit

Algorithmische Verfahren bieten für Entwickler die größten Freiheiten und ermöglichen, dank des Einsatzes vollwertigen Programmiersprachen, eine beliebige algorithmischen Interpretation und Weiterverarbeitung der Daten. Um den damit verbundenen Arbeitsaufwand für Entwickler und die Komplexität der resultierenden Systeme zu verringern, bieten viele Frameworks Widget-Bibliotheken an, die vorgefertigte Datenpräsentationen und Visualisierungen enthalten. Entwickler müssen dann lediglich das gewünschte Widget instanziieren und die darzustel-

lenden Daten an diese Widget-Instanz übergeben. Diese Widget-Bibliotheken lassen sich in der Regel um selbst definierte Widgets erweitern. Verfahren, die auf Widgets verzichten, wie beispielsweise SWP, sind für Entwickler ohne fortgeschrittene Programmierkenntnisse hingegen kaum beherrschbar.

Eine große Stärke der Template-basierten Verfahren ist ihre Nähe zu der jeweils genutzten Ausgabesprache, welches in den hier dargestellten Systemen meist HTML ist. Nutzer, die in der Lage sind, HTML-Quelltext zu erstellen, werden daher auch schnell in die Lage sein, diese Template-Sprachen für ihre eigenen Projekte zu verwenden. Allerdings fehlt diesen Verfahren die Fähigkeit zur algorithmischen Interpretation und Weiterverarbeitung der Daten.

Die große Stärke der Template-basierten Verfahren kann allerdings auch als Schwäche gewertet werden, da es mit ihnen kaum möglich ist, Ausgaben in einem anderen als dem vorgesehenen Ausgabeformat zu erstellen. Die große Stärke der transformationsbasierte Verfahren ist daher ihre Unabhängigkeit von einer Ausgabesprache oder einem Ausgabeformat. Das Ausgabeformat wird vom verarbeitenden Systeme beziehungsweise dessen Nutzer gewählt. Allerdings ist der Einarbeitungsaufwand für Entwickler in ein transformationsbasierte Verfahren deutlich höher als in ein Widget-basiertes algorithmisches Verfahren oder eine Template-basierten Verfahren.

Zwar hat sich mit Fresnel ein transformationsbasiertes Verfahren als relativ weit verbreiteter Standard etabliert, doch wird dieser in der Regel nur intern innerhalb in sich abgeschlossenen spezialisierten Anwendungen genutzt. Diese Anwendungen lassen sich nur unter hohem Aufwand erweitern, falls die gewünschten Darstellungsformen nicht verfügbar sind. Zudem ist ein direkter Zugriff auf die anzuzeigenden Daten nötig, was dazu führt, dass clientseitig nur kleine Datenmengen, wie zum Beispiel in Form von RDF-Dokumenten, verarbeitet werden können oder bei größeren Datenmengen die Datenauswahl und deren Verarbeitung serverseitig zu geschehen hat.

Wie die transformationsbasiertes Verfahren, benötigen auch die meisten Template-basierten Verfahren einen direkten Zugriff auf die Rohdaten. Für die Präsentation von Daten aus der Linked-Open-Data-Cloud sind solche Verfahren daher nur ungenügend geeignet. Eine zufriedenstellende Nutzung der LOD-Cloud verlangt unweigerlich die Möglichkeit zur Datenabfrage mittels SPARQL und eine Interpretation und Weiterverarbeitung der resultierenden Ergebnisse. Insbesondere die clientseitig ausführbaren JavaScript-basierten Widget-Bibliotheken können in dieser Hinsicht überzeugen. Weder benötigen sie einen direkten Zugriff auf die Rohdaten, noch benötigen sie eine zugrundeliegende Serveranwendung.

3.2 Konzeptionelle Überlegungen

Nachfolgend werden die konzeptionellen Überlegungen zur Implementierung von SemwidgJS beschrieben. Dazu wird zuerst ein Überblick über die Ziele und Fähigkeiten der potenziellen Nutzer des Semwidg-Projekts (vgl. Abschnitt 3.2.1) gegeben. Anschließend werden die Anforderungen, welche sich aus diesen Zielen und Fähigkeiten sowie der zuvor erfolgten Analyse der verschiedenen Frameworks und Systeme zur Präsentation und Visualisierung von Daten aus dem Linked-Data-Bereich (vgl. Abschnitt 3.1) ergeben, abgeleitet.

3.2.1 Potenzielle Nutzer von SemwidgJS

Die potenziellen Anwender von SemwidgJS sind in erster Linie Programmierer und Webentwickler mit Programmierkenntnissen (vgl. Abschnitt 1.2.3), die Linked Data in Webseiten einbinden und diese Daten gegebenenfalls visualisieren möchten. Die Anwender sind mit objektorientierten Programmiersprachen oder mit im Web häufig verwendeten Script-Sprachen vertraut und haben die dahinterstehenden Konzepte verinnerlicht. Wenn die Nutzer nicht bereits mit JavaScript vertraut sind, sollten sie problemlos in der Lage sein, die grundlegenden Funktionen von JavaScript zu erlernen.

3.2.2 Anforderungen

Nachfolgend werden anhand der zuvor beschriebenen Bedürfnisse der potenziellen Nutzer (vgl. Abschnitt 3.2.1) und der Beurteilungen der untersuchten Verfahren zur Präsentation und Visualisierung von Daten aus dem Linked-Data-Bereich (vgl. Abschnitt 3.1) die Anforderungen an SemwidgJS abgeleitet.

Grundlegende Anforderungen: Die Hauptaufgabe von SemwidgJS besteht darin, Daten aus dem Linked-Data-Bereich in reguläre Webseiten einzubetten. Dazu muss SemwidgJS in der Lage sein, die entsprechenden Daten abzufragen, die zurückgelieferten Ergebnisse zu verarbeiten und aufzubereiten sowie ansprechende Präsentationen und Visualisierungen in einer HTML-konformen Weise zu erzeugen und in die Webseite zu integrieren. Wie die konzeptionellen Überlegungen zur Abfragesprache SemwidgQL bereits aufgezeigt haben (vgl. Abschnitt 2.2.2), lässt sich der größte Teil der Daten der in der LOD-Cloud verzeichneten Datenbanken mittels SPARQL abfragen. Dementsprechend sollte SemwidgJS diese SPARQL-Endpoints ansprechen können. Natürlich sollte SemwidgJS nicht nur SPARQL als Abfragesprache unterstützen, sondern auch SemwidgQL.

Algorithmisches Verfahren: Die Analyse des aktuellen Stands der Forschung hat aufgezeigt, dass algorithmische Verfahren Entwicklern die meisten Freiheiten bei der Darstellung der Daten bieten und die Mächtigkeit der genutzten Programmiersprache bei der korrekten Interpretation und Weiterverarbeitung der Daten teilweise zwingend benötigt wird. SemwidgJS sollte daher auf einer vollwertigen Programmiersprache aufbauen, die zusätzlich von den meisten Anwendern der Zielgruppe bereits beherrscht wird.

Leicht einbindbar in bestehende Systeme: Die meisten der zuvor untersuchten Systeme verlangten von den Anwendern die Installation auf einem Webserver. Häufig übernimmt dann auch dieses System die Auslieferung der Webseite, sodass die gesamte Webpräsenz auf das neue System umgestellt werden müsste. Dies dürfte in vielen Fällen nicht

möglich oder nicht lohnend sein, wenn beispielsweise bisherige Funktionen nicht mehr unterstützt werden können oder der Aufwand den Nutzen schlicht weit überwiegt.

Da vielen potenziellen Nutzern, wie zum Beispiel Nutzer von kostenlose Blogging-Plattform oder günstigen Webhosting-Angeboten ohne vollständigen Serverzugriff, die Möglichkeit zur Installation solcher Systeme nicht gegeben ist, sollte sich SemwidgJS auch in solchen Umgebungen nutzen lassen. Zusammen mit der vorherigen Anforderung (*Algorithmisches Verfahren*) spricht daher alles für JavaScript als die zu nutzende Programmiersprache für SemwidgJS. Eine JavaScript-Bibliothek lässt sich fast immer, unabhängig von den sonstigen Rahmenbedingungen, in eine Webseite einbinden. Die Verarbeitung der Daten kann clientseitig erfolgen und bedarf keiner weiteren Serversoftware.

Widgets: Da die Nutzung algorithmischer Verfahren deutlich komplexer sein kann, als die der anderen Verfahren, haben sich in diesem Bereich Widgets etabliert. Diese beschreiben vordefinierte Präsentationen oder Visualisierungen, die sich über meist nur wenige Parameter an die gestellten Bedürfnisse anpassen lassen. Dadurch wird die Komplexität und der Aufwand zur Nutzung algorithmischer Verfahren erheblich reduziert. Statt dass ein Nutzer die Präsentation der Daten selbst programmieren muss, muss er lediglich das geeignete Widget auswählen und mit den entsprechenden Parameterwerten instanziieren.

Die Deklaration der Widgets geschieht in der Regel innerhalb des HTML-Quelltextes der Webseite. Eine JavaScript-Bibliothek liest diese anschließend ein, verarbeitet sie und erzeugt an der Stelle, an der das Widget deklariert wurde die entsprechende Datenpräsentation oder Visualisierung. SemwidgJS sollte auf einem solchen Widget-System aufbauen und geeignete Widgets für eine Vielzahl von Anwendungsfällen mitliefern.

Erweiterbarkeit: Auch wenn SemwidgJS eine Vielzahl von Widgets vordefiniert, kann damit unmöglich jeder Anwendungsfall abgedeckt werden. Nutzer müssen daher die Möglichkeit haben, bestehende Widgets zu erweitern oder auch benutzerdefinierte Widgets erstellen zu können. Dabei sollte SemwidgJS sie durch die Bereitstellung entsprechender Funktionen unterstützen und so aufgebaut sein, dass diese Aufgaben möglichst einfach durchgeführt werden können.

Abfragesprachen: Als Abfragesprachen sollten sowohl SPARQL wie auch SemwidgQL unterstützt werden. Entsprechend müssen sich für beide Sprachen Verbindungen zu SPARQL-Endpoints und für SemwidgQL zusätzlich Resources und gegebenenfalls Prefixes konfigurieren lassen.

Wiederverwendbarkeit: Um redundante auszuführende Arbeiten zu vermeiden, sollten von Nutzern erstellte Widget-Kompositionen wiederverwendet werden können. Es sollte sowohl möglich sein, mehrere dieser Kompositionen auf der selben Seite instanziieren zu können wie auch eine Wiederverwendung über verschiedene Seiten hinweg. Dabei muss ein Mechanismus gegeben sein, der es erlaubt, bestimmte Parameter der Kompositionen nachträglich zu verändern, um beispielsweise die Ausgabesprache oder die zu Grunde liegende Resource auszutauschen, zu welcher gewisse Werte dargestellt werden sollen.

Interaktivität: Nur wenige der untersuchten Verfahren erlauben die Erstellung interaktiver Webseiten. Die meisten sind darauf ausgelegt statische Werte darzustellen. Um sich von den untersuchten Arbeiten weiter abzugrenzen, sollte SemwidgJS die Erstellung interaktiver Webseiten ermöglichen. Dazu sollten sowohl Widgets zur Eingabe wie auch Ausgabe

von Daten unterstützt werden. Zudem müssen Widgets auf die Veränderung von Daten innerhalb der Webseite reagieren können. Unabhängig davon, ob die Veränderungen durch ein Eingabe-Widget oder durch ein externes Programm oder Element erfolgt ist.

Dynamische Webseiten: Um die Verarbeitung von mit Zeitstempeln annotierten Daten zu vereinfachen, die beispielsweise bei der Speicherung von Sensordaten anfallen, wurden SemwidgQL einige spezielle Funktion hinzugefügt (vgl. Abschnitt 2.3). Da sich solche Sensordaten ständig ändern können, sollten sich mit SemwidgJS dynamische Webseiten erstellen lassen, welche aktualisierte Daten automatisch nachladen. Eine solche Funktionalität wird von keiner der zuvor untersuchen Arbeiten unterstützt und würde SemwidgJS erheblich von diesen abheben.

Die konkrete Implementierung dieser hier aufgeführten Anforderungen wird im nachfolgenden Abschnitt beschrieben.

3.3 Implementierung

Auf Basis der zuvor gemachten konzeptionellen Überlegungen (vgl. Abschnitt 3.2) wird in diesem Abschnitt die Implementierung von SemwidgJS beschrieben. Zu Beginn werden grundlegende Funktionsweisen erläutert und aufgezeigt, welche Elemente in SemwidgJS existieren und wie diese zusammenwirken. Anschließend werden die einzelnen Element detailliert vorgestellt. Ebenfalls wird auf die weiterführende Nutzung von SemwidgJS – jenseits der einfachen Anwendung als Widget-Bibliothek – als Framework mit der Möglichkeit zur Erzeugung benutzerdefinierter Widget-Klassen eingegangen.

3.3.1 Grundlagen

SemwidgJS wurde, wie es der Name bereits andeutet, als JavaScript-Bibliothek entworfen und kann in in nahezu jede beliebige Webseite eingebunden werden, mit dem Ziel diese mit Daten von SPARQL-Endpoints zu erweitern. Die Darstellung dieser Daten erfolgt mittels *Semantic Data Widgets* (vgl. Abschnitt 1.2.2), die innerhalb des Quelltextes der Webseite in Form von HTML-Elementen deklariert werden. Die Verarbeitung dieser Elemente durch die SemwidgJS-Bibliothek erfolgt automatisch, sobald die Seite geladen wurde. SemwidgJS kann jedoch nicht nur als Widget-Bibliothek eingesetzt werden, sondern auch als Framework zur Erzeugung benutzerdefinierter Widget-Klassen, die auf die anwendungsspezifischen Anforderungen hinsichtlich ihrer Funktionalität und darzustellender Daten optimiert wurden.

Um den zuvor aufgestellten Anforderungen (vgl. Abschnitt 3.2.2) gerecht zu werden, baut SemwidgJS auf drei Element-Klassen auf, deren genaue Beschreibung im weiteren Verlauf dieses Kapitels erfolgt:

Konfigurationselemente: Diese Elemente enthalten Informationen, die von Widgets und Templates, zusätzlich zu ihren lokalen Eigenschaften, zur Datenabfrage und -anzeige benötigt werden (vgl. Abschnitt 3.3.2). Dies umfasst beispielsweise Informationen zur Konfiguration von SPARQL-Endpoints und Resources, welche in SemwidgQL-Abfragen als bennante Resources (vgl. Abschnitt 2.3.1: *Benannte Resources*) verwendet werden können.

Widget-Instanzen: Diese Elemente sind Instanziierungen bestimmter Widget-Klassen. Die Instanziierung erfolgt im HTML-Quelltext und erfordert die Angabe einiger Parameter (vgl. Abschnitt 3.3.3). Standardmäßig ist beispielsweise die Angabe einer SPARQL- oder SemwidgQL-Abfrage erforderlich, die definiert, welche Daten dargestellt werden sollen.

Template-Definitionen und -Instanzen: Templates ermöglichen die Wiederverwendung von Kompositionen aus Widgets und anderem HTML-Inhalten. Eine Template-Definition enthält den HTML-Quelltext der Widget-Instanzen. Durch die Instanziierung dieser Template-Definitionen lassen sich diese Kompositionen an beliebiger Stelle und in beliebiger Häufigkeit wiederverwenden (vgl. Abschnitt 3.3.4).

HTML-Objekte innerhalb der SemwidgJS-Umgebung

HTML-Objekte sind JavaScript-Repräsentationen all jener Objekte, welche ein Entwickler, der SemwidgJS einsetzt, innerhalb des HTML-Quelltextes in Form eines HTML-Elements deklarieren und konfigurieren kann. Hierbei ist zwischen für den Endnutzer sichtbaren und unsichtbaren Elementen zu unterscheiden. Konfigurationselemente und Template-Definitionen sind unsichtbar, enthalten jedoch relevante Informationen, die unter anderem für die Datenabfrage

benötigt werden. Widget- und Template-Instanzen sind sichtbar und präsentieren und visualisieren die abgerufen Daten.

Nach dem Laden einer Webseite durchsucht SemwidgJS deren Document Object Model (DOM) und erstellt für jedes HTML-Element der zuvor genannten Objekt-Klassen eine entsprechende JavaScript-Repräsentation. Die Elemente werden mittels data-*-Attributen[20] konfiguriert. Um von SemwidgJS erkannt zu werden, muss ein HTML-Element ein Attribut besitzen, dessen Bezeichnung mit dem Präfix `data-sw-` beginnt, gefolgt von einem Bezeichner, welcher die Objekt-Klasse bestimmt (z. B. `config`) und einem Attributwert, welcher die genaue Unterklasse angibt (z. B. `endpoint`). Alle weiteren Eigenschaften werden durch (Unter-)Klassen-spezifische Attribute bestimmt, die mit dem Präfix `data-swp-` beginnen. Listing 3.5 zeigt beispielhaft die Deklaration eines Endpoint-Konfigurationselements für den DBpedia-SPARQL-Endpoint.

Das Framework überwacht das DOM der Webseite mittels der Mutation Observer API[21], um so nachträgliche Veränderungen an den HTML-Repräsentationen der Objekte zu erkennen. HTML-Objekte sind automatisch in das SemwidgJS-Event-System eingebunden, was ihnen ermöglicht eigene Events auszulösen und auf fremde Events zu reagieren.

Jedes HTML-Objekt verfügt mindestens über das Attribut `data-swp-id`, welches dieses identifiziert und anderen Elementen erlaubt, eine Verknüpfung zu diesem herzustellen. IDs müssen daher eindeutig sein und dürfen innerhalb einer Webseite nicht mehrfach vergeben werden. Sollte einem Element innerhalb des HTML-Quelltextes keine ID zugewiesen worden sein, erzeugt SemwidgJS automatisch einen eindutigen Wert. Die Werte des ID-Attributs dürfen nachträglich nicht mehr verändert werden. Die Funktionen, die die Manipulationen an den verschiedenen Objekten überwachen, verhindern auch das nachträgliche Ändern dieses Attributs, indem sie bei Manipulationsversuchen von außen, das Attribut auf den ursprünglichen Wert zurücksetzen.

Das Event-System

Ein auf Callback-Funktionen basierendes Event-System ermöglicht es sämtlichen SemwidgJS-Objekten, sich bei Objekten der SemwidgJS-Klassen zu registrieren, die von der Klasse `EventDispatcher` abgeleitet wurden, und sich so über spezifische Änderungen an diesen Objekten informieren zu lassen. Sobald eine dieser Änderungen vorgenommen wird, wird die bei der Registrierung übergebene Callback-Funktion ausgeführt, was dem anderen Objekt ermöglicht, entsprechend auf die Änderung zu reagieren. Beispielsweise werden alle zuvor beschriebenen SemwidgJS-HTML-Objekte von der `EventDispatcher`-Oberklasse abgeleitet.

Bidirektionale Synchronisierung zwischen dem HTML-DOM und der JavaScript-Engine

Das SemwidgJS-Framework sorgt für eine bidirektionale Synchronisierung von HTML-Elementen beziehungsweise HTML-Objekten zwischen ihren jeweiligen Repräsentationen im HTML-DOM und der JavaScript-Engine hinsichtlich ihrer SemwidgJS-bezogenen Attributwerte. Wird beispielsweise ein Attributwert der HTML-Repräsentation eines Resource-Konfigurationselements durch ein unabhängiges JavaScript-Programm manipuliert, so wird diese Änderung vom Framework erkannt und auf die Repräsentation innerhalb der JavaScript-Engine gespiegelt. Ebenso wird eine interne Änderung dieses Konfigurationselements innerhalb der JavaScript-Engine durch SemwidgJS auf dessen HTML-Repräsentation übertragen. In beiden Fällen wird

[20] `https://www.w3.org/TR/html5/dom.html#element-attrdef-global-data`
[21] `https://www.w3.org/TR/dom/#interface-mutationobserver`

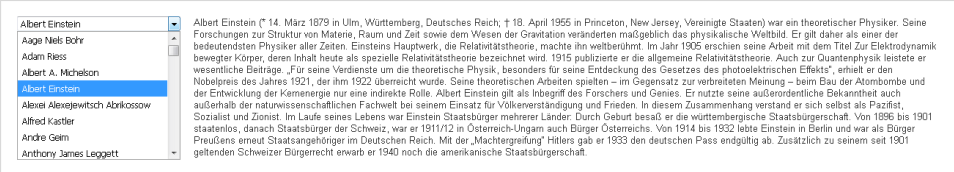


Abbildung 3.1: Eine einfache interaktive Webseite. Nutzer können aus einer Liste eine Person auswählen und bekommen den zu dieser dazugehörigen Beschreibungstext angezeigt. Abbildung 3.2 zeigt das Zusammenwirken der einzelnen sichtbaren Widgets sowie unsichtbaren Konfigurationselementen bei einer Nutzeraktion.

ein Event ausgelöst, das weitere mit diesem Konfigurationselement verknüpfte HTML-Objekte über diese Änderung informiert und ihnen die Möglichkeit gibt entsprechend darauf zu reagieren. Ein Widget würde zum Beispiel die Datenabfrage, die mit einer Resource verknüpft ist, neu ausführen und anschließend die Ausgabe aktualisieren, wenn ebenjene Resource verändert wurde.

Die bidirektionale Synchronisierung erlaubt somit eine einfache aktive Interaktion zwischen SemwidgJS und anderen unabhängigen JavaScript-Programmen, ohne dass eine der Parteien eine spezielle Programmierschnittstelle für eine solche Interaktion zur Verfügung stellen müsste. Andere JavaScript-Programme können gezielt Attributwerte von SemwidgJS-HTML-Objekten manipulieren, um eine bestimmte Aktion auszulösen. Ebenso kann SemwidgJS bestimmte HTML-Elemente manipulieren, auf deren Änderungen andere Programme reagieren können.

Kommunikation zwischen Elementen und Interaktive Webseiten

Nachfolgend wird ein Beispiel vorgestellt, dass eine einfache interaktive Webseite beschriebt, die aus lediglich vier SemwidgJS-Elementen erzeugt werden kann. Die Kommunikation der Elemente erfolgt über die zuvor beschriebene bidirektionale Synchronisierung. Die Webseite ermöglicht es einem Nutzer, aus einer Liste mit Nobelpreisträgern in Physik eine Person auszuwählen und zu dieser einen Beschreibungstext zu erhalten. Die Webseite besteht aus einem Endpoint-Konfigurationselement, welches den SPARQL-Endpoint definiert, aus dem alle Daten dieser Webseite abgerufen werden sollen, sowie einem Resource-Konfigurationselement, das mit dem Endpoint-Konfigurationselement verknüpft ist, einem Auswahl-Widget, welches die Liste der Nobelpreisträger enthält, und einem Text-Widget, welches den Beschreibungstext der Person anzeigen soll, dessen URI innerhalb des Resource-Konfigurationselement gespeichert ist. Abbildung 3.1 zeigt diese einfache interaktive Webseite.

Abbildung 3.2 zeigt das Zusammenwirken der einzelnen Widgets bei einer Nutzeraktion. Jedes Mal, wenn ein Nutzer eine Person aus der Liste auswählt (1), wird der URI dieser Person in das Resource-Konfigurationselement geschrieben (2). Das Text-Widget wird über diese Änderung informiert (3) und dazu veranlasst, den Beschreibungstext neu zu laden und dazu den nun geänderte Resource-URI zu nutzen (4, 5, 6). Nur das Auswahl- und das Text-Widget sind für den Nutzer sichtbar. Die beiden Konfigurationselemente werden nicht dargestellt, sind aber innerhalb des DOMs der Webseite präsent. Die Webseite lässt sich leicht erweitern, indem weitere Widgets hinzugefügt werden, die beispielsweise ein Porträtfoto, eine Karte mit dem Geburtsort oder auch ein Klimadiagramm des Heimatstadt der gewählten Person anzeigen.

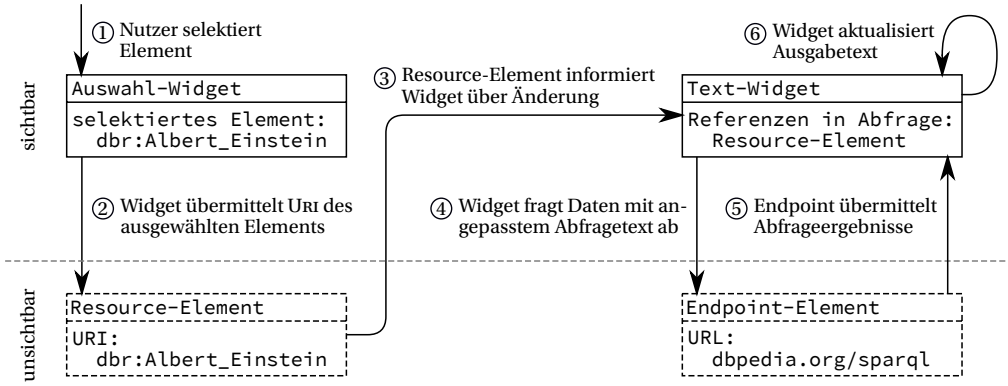


Abbildung 3.2: Zusammenwirken einzelner Widgets bei einer Nutzeraktion. Die Selektion eines Elements des Auswahl-Widgets überschreibt den Wert des Resource-Elements, welches das Text-Widget über diese Änderung informiert. Dieses aktualisiert daraufhin den Ausgabetext auf Basis einer neu gestellten Datenbankabfrage.

3.3.2 Konfigurationselemente

Konfigurationselemente werden mittels generischer, normalerweise nicht-sichtbarer HTML-Elemente (`div`, `span`) erzeugt und enthalten Daten, die von Widgets (vgl. Abschnitt 3.3.3) und Templates (vgl. Abschnitt 3.3.4) – zusätzlich zu ihren lokalen Eigenschaften – benötigt werden, um eine Anzeige zu erzeugen, um Daten zwischen SemwidgJS-Objekten auszutauschen, aber auch, um Datenabfragen zu verkürzen und deren Formulierung zu vereinfachen. Die Grundfunktionen der in SemwidgJS einsetzbaren Konfigurationselemente werden nachfolgend beschrieben. Eine detaillierte Auflistung aller erforderlichen und optionalen Parameter, deren Funktionen und ein Quelltextbeispiel für jedes Konfigurationselement ist in Anhang D.1 enthalten.

Endpoints

Ein Endpoint-Konfigurationselement enthält alle Informationen, die benötigt werden, um eine Verbindung zu einem SPARQL-Endpoint aufzubauen und die dort gespeicherten Daten abzufragen. Innerhalb einer Webseite können beliebig viele Endpoints definiert werden, die sich allerdings in ihrem ID-Wert voneinander unterscheiden müssen. Einer dieser Endpoints kann als Standard-Endpoint ausgewählt werden. Wird in einer Abfrage oder einem Widget kein spezifischer Endpoint angegeben, wird die Abfrage an den Standard-Endpoint gestellt. Bevor SemwidgJS eine Datenbankabfrage stellt, wird nach nachfolgen beschriebener Priorität der passende SPARQL-Endpoint ausgewählt:

1. Der Endpoint, der mit dem Haupt-Resource-Element der Abfrage (vgl. Abschnitt 3.3.2: *Resources*) verknüpft ist, falls es sich bei dieser Abfrage um eine SemwidgQL-Abfrage handelt.

2. Der Widget-Eigenschaft-spezifische Endpoint (vgl. Abschnitt 3.3.3: *Zusätzliche Query-Eigenschaften*), der der Widget-Eigenschaft untergeordnet ist, in welcher die Abfrage formuliert wurde.

Listing 3.5: HTML-Quelltext eines Endpoint-Konfigurationselements. Das Element enthält alle Informationen, die benötigt werden, um eine Verbindung zum DBpedia-SPARQL-Endpoint aufzubauen. Der letzte Parameter definiert diesen Endpoint als Standard-Endpoint für alle Abfragen, für die keine höher priorisierten Endpoint-Angaben vorliegen.

```
1 <div data-sw-config="endpoint"
2      data-swp-id="dbpedia"
3      data-swp-url="//dbpedia.org/sparql"
4      data-swp-default="true">
5 </div>
```

3. Der Widget-spezifische Endpoint (vgl. Abschnitt 3.3.3: *Widget-Eigenschaften*), der zu dem Widget gehört, in welcher die Abfrage formuliert wurde.

4. Der ausgewählte Standard-Endpoint der Webseite.

Eine Webseite, die Widgets mit Daten eines SPARQL-Endpoints darstellen soll, muss mindestens ein Endpoint-Konfigurationselement enthalten. Listing 3.5 zeigt beispielhaft den HTML-Quelltext für die Instanziierung eines Endpoint-Objekts, welches eine Verbindung zum DBpedia-SPARQL-Endpoint aufbaut. Weitere Informationen zum Endpoint-Konfigurationselement werden in Anhang D.1.1 beschrieben.

Resources

Ein Resource-Konfigurationselement erzeugt eine benannte Resource (vgl. Abschnitt 2.3.1: *Benannte Resources*), die für die Formulierung von SemwidgQL-Abfragen genutzt werden kann, anhand eines Namens und einem Resource-URI, und verknüpft dieses Element mit einem Endpoint-Konfigurationselement. Die Verknüpfung erfolgt durch Angabe des ID-Wertes des Endpoint-Konfigurationselements. Benannte Resources sind für die Formulierung von SemwidgQL-Abfragen nicht zwingend nötig, erleichtern die Verwendung von SemwidgJS jedoch erheblich.

Ist die benannte Resource das Haupt-Resource-Element einer SemwidgQL-Abfrage, so wird die Datenbankabfrage immer an den verknüpften SPARQL-Endpoint gestellt. Das Haupt-Resource-Element einer SemwidgQL-Abfrage ist die benannte Resource (vgl. Abschnitt 2.3.1: *Benannte Resources*), die nach der Umwandlung einer SemwidgQL-Abfrage in ihren Syntaxbaum an oberster Stelle steht. In der Regel sollte das die benannte Resource sein, mit der die SemwidgQL-Abfrage beginnt.

Erfolgt eine Änderung an einem Resource-Konfigurationselement, so werden über das Event-System alle interessierten SemwidgJS-Objekte über diese Änderung informiert. Sämtliche Widgets, die diese Resource enthalten, stellen eine neue Abfrage an den entsprechenden SPARQL-Endpoint mit gegebenenfalls geändertem Abfragetext. Dieses Verhalten kann dazu genutzt werden, um interaktive Webseiten zu erstellen. Listing 3.6 zeigt die Deklaration eines Resource-Konfigurationselements, welches eine benannte Resource mit dem Bezeichner `physicists` beschreibt und mit dem, im vorherigen Abschnitt erstellten DBpedia-Endpoint-Konfigurationselement verknüpft ist. Bei der Verarbeitung einer SemwidgQL-Abfrage, die diese benannte Resource nutzt, wird der Bezeichner durch den Resource-URI von Albert Einstein ersetzt. Weitere Informationen zum Resource-Konfigurationselement werden in Anhang D.1.2 beschrieben.

Listing 3.6: HTML-Quelltext eines Resource-Konfigurationselements. Das Element erzeugt eine benannte Resource mit dem Bezeichner `physicists` und ist mit dem, im vorherigen Abschnitt erstellten DBpedia-Endpoint-Konfigurationselement verknüpft. Bei der Verarbeitung einer SemwidgQL-Abfrage, die diese benannte Resource nutzt, wird der Bezeichner durch den Resource-URI von Albert Einstein ersetzt.

```
1 <div data-sw-config="resource"
2      data-swp-id="physicists"
3      data-swp-uri="http://dbpedia.org/resource/Albert_Einstein"
4      data-swp-endpoint="dbpedia">
5 </div>
```

Namespaces

Namespace-Konfigurationselemente beschreiben Namespaces beziehungsweise Präfixe, die es Anwendern erlauben, präfigierte URIs für Resources oder Eigenschaften innerhalb von SemwidgQL-Abfragen zu nutzen (vgl. Abschnitt 2.3.1: *Präfixdefinition*). Da diese nicht direkt in der Abfrage definiert werden können, wie es beispielsweise in SPARQL der Fall ist, müssen Präfixe über Namespace-Konfigurationselemente deklariert werden. Allerdings können sie so auch von verschiedenen Abfragen wiederverwendet werden, während sie in SPARQL-Abfragen jedes mal erneut angefügt werden müssen.

Der SemwidgJS-Bibliothek wurde eine Liste mit mehreren hundert häufig genutzten Präfixen beigefügt, die automatisch an den Transcompiler übergeben wird. Dies führt dazu, dass in vielen Fällen gar keine Namespace-Konfigurationselemente durch den Anwender mehr erzeugt werden müssen. Weitere Informationen zum Namespace-Konfigurationselement werden in Anhang D.1.3 beschrieben.

Mappings

Mappings dienen als globale Variablen, die in jeden Parameterwert eines SemwidgJS-Widgets eingebettet werden können, unabhängig davon, ob dieser Parameterwert statisch ist oder eine Datenbankabfrage beschreibt. Ein Mapping-Konfigurationselemente enthält einen Mapping-Bezeichner und einen Datenwert.

Damit SemwidgJS erkennen kann, dass ein Parameterwert ein Mapping enthält, muss der Mapping-Bezeichner von doppelte Dollarzeichen umschlossen werden. Bei der Verarbeitung eines Widgets werden zuerst alle Mapping-Bezeichner durch die mit ihnen verknüpften Werte ersetzt. Listing 3.7 zeigt, wie ein Mapping-Konfigurationselement in Verbindung mit einem Widget genutzt werden kann. Das Widget enthält eine SemwidgQL-Abfrage mit einem Filterausdruck, dessen Wert durch ein Mapping-Element bestimmt wird. Bei der Verarbeitung des Widgets wird der Mapping-Bezeichner durch den Datenwert des Mappings ersetzt. In diesem Beispiel wird das Mapping dazu genutzt, die Sprache für einen abzufragenden Beschreibungstext zu bestimmen.

Wie auch bei Resource-Elementen reagieren alle anderen Elemente auf Änderungen an Mapping-Konfigurationselementen, wenn sie mit diesen verknüpft sind. Wird beispielsweise der Datenwert des Mappings aus vorherigem Beispiel geändert, wird eine neue Datenabfrage erzeugt, die den Beschreibungstext in einer anderen Sprache abfragt und das Widget wird daraufhin aktualisiert. Weitere Informationen zum Mapping-Konfigurationselement werden in Anhang D.1.4 beschrieben.

Listing 3.7: HTML-Quelltext eines Mapping-Konfigurationselements in Verbindung mit einem Widget. Bei der Verarbeitung des Widgets wird die Zeichenkette `$$language$$` durch den Datenwert des Mappings ersetzt, sodass der Beschreibungstext für die entsprechende Resource nur in englischer Sprache abgefragt wird.

```
<div data-sw-config="mapping"
     data-swp-id="language"
     data-swp-value="en">
</div>
```

```
<div data-sw-widget="SwText"
     data-swp-id="physicistDescription"
     data-swp-value="{physicists.rdfs:comment(@lang = '$$language$$')}">
</div>
```

Proxies

Damit SemwidgJS mit einem externen SPARQL-Endpoint kommunizieren kann, ist es erforderlich, dass dieser das CORS-Protokoll (Cross-Origin Resource Sharing)[22] unterstützt, um somit die Same-Origin-Policy des Browsers umgehen zu dürfen. Da eine Vielzahl der SPARQL-Endpoints der LOD-Cloud CORS nicht implementieren, erlauben Proxy-Konfigurationselemente die Abfrage über einen anderen Server zu leiten, welcher das CORS-Protokoll unterstützt. Wird ein Endpoint-Konfigurationselemente mit einem Proxy-Konfigurationselemente verknüpft, wird automatisch jede Abfrage, die an diesen Endpoint gerichtet ist, über den Proxy umgeleitet. Weitere Informationen zum Proxy-Konfigurationselement werden in Anhang D.1.5 beschrieben.

3.3.3 Widgets

Widgets stellen die Quintessenz von SemwidgJS dar und sind für die Präsentation der Daten beliebiger SPARQL-Endpoint verantwortlich. Dabei ist formal zwischen den Widget-Klassen und Widget-Instanzen zu unterscheiden. Widget-Instanzen sind konkrete Ausprägungen einer Widget-Klasse und verfügen über bestimmte Parameterwerte, wie zum Beispiel Datenbankabfragen, sowie über eine sichtbare Darstellung der Ein- oder Ausgabedaten. Der Einfachheit halber werden im weiteren Verlauf sowohl Widget-Klassen wie auch Widget-Instanzen als Widgets bezeichnet. Welche der beiden dabei genau gemeint ist, ergibt sich aus dem jeweiligen Kontext.

Im Gegensatz zu den Konfigurationselementen ist die Auswahl der HTML-Elemente zur Instanziierung von Widgets nicht auf nicht-sichtbare Elemente (`div`, `span`) begrenzt. Stattdessen können bei Bedarf auch andere HTML-Elemente genutzt werden. Ein Widget erbt in diesem Fall die Eigenschaften dieses Elements. Dies geschieht automatisch, da der generierte HTML-Quelltext, welcher für die Ausgabe des Widgets verantwortlich ist, in das entsprechende HTML-Element eingebettet wird.

Listing 3.8 zeigt den HTML-Quelltext eines Text-Widgets, wie er im DOM eines Browsers nach der Verarbeitung des Widgets stehen würde. Das Widget nutzt die Konfigurationselemente, die in den zuvor gezeigten Beispielen (vgl. Listings 3.5, 3.6 und 3.7) aufgeführt wurden. Das Widget enthält eine SemwidgQL-Abfrage, welche das englische Label der Resource zu Albert

[22] `https://www.w3.org/TR/cors/`

Listing 3.8: Ein verarbeitetes SemwidgJS-Widget. Das Widget enthält eine SemwidgQL-Abfrage, welche das englische Label der Resource zu Albert Einstein abfragt, und das erste Ergebnis dieser Abfrage als Text in den inneren Teil des Widget-Elements einfügt.

```
1  <h1 data-sw-widget="SwText"
2      data-swp-value="{physicists.rdfs:label(@lang = '$$language$$')}">
3
4    <!-- created by SemwidgJS -->
5    <span>Albert Einstein</span>
6
7  </h1>
```

Einstein abfragt, und das erste Ergebnis dieser Abfrage als Text in den inneren Teil des Widget-Elements einfügt. Da der Text innerhalb eines neutralen Containers (`span`), welches in einem Überschriftenelement (`h1`) geschachtelt ist, ausgegeben wird, erbt es automatisch die visuellen Eigenschaften des umgebenden Elements und wird vom Browser als Überschrift dargestellt.

Widget-Eigenschaften

Neben der ID, die jedes SemwidgJS-Element besitzt, verfügen alle Widgets über eine ganze Reihe weitere Standardeigenschaften. Zusätzlich kann jedes Widget über weitere Widget-spezifischen Eigenschaften verfügen. Die relevantesten der Standardeigenschaften werden im weiteren Verlauf dieses Abschnitts genauer beschrieben. Eine vollständige Übersicht über alle Standardeigenschaften befindet sich in Anhang D.2.1.

`endpoint`: Diese Eigenschaft ermöglicht die Benennung eines Widget-spezifischen Endpoints, der für alle Abfragen eines Widgets genutzt wird, für den Fall, dass kein Resource-spezifischer oder Widget-Parameter-spezifischer Endpoint festgelegt wurde.

`limit`: Da SemwidgQL-Abfragen keine Limitierung der Anzahl der Abfrageergebnisse zulassen, kann dies nachträglich über diese Eigenschaft erfolgen. Nachdem der SemwidgQL-zu-SPARQL-Transcompiler die Abfrage erzeugt hat, wird der hier festgelegte Wert an die SPARQL-Abfrage angehängt.

`style`: Diese Eigenschaft kann CSS-Informationen enthalten, die bei der Verarbeitung an das Widget übergeben werden. In den meisten Fällen werden diese Informationen in das Style-Attribut der erzeugten HTML-Ausgabe eingebettet.

`value`: Standardmäßig werden anhand dieser Eigenschaft die Werte mittels eine Datenbankabfrage beschrieben, die das Widget anzeigen soll. In den meisten Fällen ist es ausreichend lediglich diese Eigenschaft zu nutzen, um die gewünschte Datenpräsentation zu erzeugen (vgl. Listing 3.8).

Eine Eigenschaft besitzt neben ihrem Namen, einen Eigenschaftentyp, welcher im nachfolgenden Abschnitt näher betrachtet wird, und einen Kommentar, der Aufschluss über den Einsatzzweck der Eigenschaft geben soll. Dieser Kommentar wird beispielsweise von SemwidgED als Tooltip für das entsprechende Eingabefeld angezeigt (vgl. Abschnitt 4.3.3).

Eigenschaftentypen

In den meisten Fällen können die Attributwerte von Widgets Datenbankabfragen in Form von SemwidgQL- oder SPARQL-Abfragen enthalten. Attribute können jedoch auch bestimmte andere Eigenschaftentypen voraussetzen. Welchen Eigenschaftentyp ein Attributwert enthalten soll, wird in der Widget-Klassendeklaration festgelegt. Insgesamt wird in SemwidgJS zwischen vier Eigenschaftentypen unterschieden.

`fixed`: Die Werte von Eigenschaften dieses Typs können nachträglich nicht mehr verändert werden. Dieser Eigenschaftentyp wird beispielsweise für die IDs (`data-swp-id`) von sämtlichen SemwidgJS-Elementen genutzt.

`static`: Die Werte von Eigenschaften dieses Typs werden nur als statische Zeichenketten interpretiert. Der Wert kann jedoch während der Laufzeit geändert werden und andere Elemente werden über diese Änderung informiert. Dieser Eigenschaftentyp wird beispielsweise für die Verknüpfung über IDs zwischen den verschiedenen SemwidgJS-Elementen genutzt.

`selector`: Selektoren können von interaktiven Elementen genutzt werden, die es ermöglichen, die Daten von anderen Widgets zu manipulieren. Die Werte von Eigenschaften dieses Typs beschreiben, welches konkrete Element und welche Eigenschaft manipuliert werden soll.

Ein Auswahl-Widget kann beispielsweise mehrere Sprachcodes enthalten und den ausgewählten Code in den Wert des Mapping-Konfigurationselements aus Listing 3.7 schreiben (selektiert durch `mapping.value`). Die SemwidgJS-Bibliothek stellt einige Funktionen für diesen Eigenschaftentyp bereit, die die Manipulation von weiteren Widgets vereinfachen.

`query`: Die Werte von Eigenschaften dieses Typs können SemwidgQL- und SPARQL-Abfragen, aber auch statische Zeichenketten oder eine beliebige Kombination aus diesen enthalten. Dies ist der Standardtyp der meisten Eigenschaften. Um zu kennzeichnen, dass Teile eines Attributwerts als SemwidgQL- oder SPARQL-Abfrage zu interpretieren sind, müssen diese in einfache geschweifte Klammern (SemwidgQL) oder in doppelte geschweifte Klammern (SPARQL) eingeschlossen werden.

Jede Eigenschaft des Typs `query` erzeugt automatisch zusätzliche weitere Eigenschaften, die im nachfolgenden Abschnitt beschrieben werden.

Zusätzliche Query-Eigenschaften

Sämtliche Eigenschaften des Typs `query` erzeugen automatisch eine Reihe weiterer Eigenschaften. Die wichtigsten dieser Eigenschaften werden im weiteren Verlauf beschrieben. Eine Übersicht über alle zusätzlichen Query-Eigenschaften befindet sich in Anhang D.2.3. Diese Eigenschaften verwenden den selben Eigenschaftennamen wie die übergeordnete Eigenschaft, welche diese erzeugt hat, zuzüglich eines Suffixes.

`*-endpoint`: Diese Eigenschaft ermöglicht die Benennung eines Widget-Eigenschaft-spezifischen Endpoints, der für alle Abfragen der übergeordneten Eigenschaft genutzt wird, für den Fall, dass kein Resource-spezifischer Endpoint festgelegt wurde.

`*-order`: Da SemwidgQL-Abfragen keine Sortierung der Ergebnisliste zulässt, lässt sich eine Sortierung nachträglich anhand dieser Eigenschaft festlegen. Die Syntax für den Wert der Eigenschaft entspricht weitestgehend der Syntax von SQL, in der zuerst die Variable und dann dessen Sortierreihenfolge (aufsteigen `asc` oder absteigend `desc`) angegeben werden. Wird keine Sortierreihenfolge angegeben, wird automatisch aufsteigend sortiert. Mittels Kommata getrennt, können mehrere Variablen zur Sortierung hintereinander angegeben werden.

Die Syntax in SemwidgJS unterscheidet sich von der Syntax in SQL jedoch in der Benennung der zu sortierenden Variablen. Da die Variablennamen innerhalb der SPARQL-Abfrage automatisch vom SemwidgQL-zu-SPARQL-Transcompiler generiert werden, sind sie dem Nutzer auch nicht bekannt. Daher lassen sich die Variablen über ihren Index innerhalb der SPARQL-Ergebnisliste auswählen. Zur einfacheren Anwendung wurden hier die beiden Schlüsselwörter `first` und `last` eingeführt. Zusätzlich lassen sich vereinfachte arithmetische Ausdrücke (`+-*/`) einfügen, die auch mit den beiden Schlüsselwörtern kombiniert werden können.

Zu beachten ist jedoch, dass die Sortierung lokal durchgeführt wird und daher nur die Ergebnisse in die gewünschte Reihenfolge gebracht werden können, die auch lokal vorliegen. Da einige SPARQL-Endpoints die maximale Länge der Ergebnisliste begrenzen, kann sich daher das Ergebnis der nachträglich sortierten Liste von einer serverseitig sortierten Liste unterscheiden. Dies ist insbesondere dann relevant, wenn die Sortierung dafür genutzt werden soll, um den minimalen oder maximalen Wert in der Liste auszugeben. Der tatsächliche Extremwert wurde bei einer sehr langen Ergebnisliste gegebenenfalls gar nicht zurückgegeben.

Listing 3.9 zeigt die Nutzung der *-order-Eigenschaft anhand eines Beispiels. Darin werden in einem Listen-Widget die Geburtstorte Albert Einsteins aus einer erweiterten Version des Beispielgraphen (vgl. Abbildung 2.1) aufgelistet. Die Geburtsorte werden nach ihrem Gründungsjahr aufsteigend sortiert, sodass der zuerst gegründete an oberster Stelle steht.

`*-selector`: Sämtliche SemwidgJS-Widgets besitzen Zugriff auf eine Hilfsfunktion, die ihnen die Auswahl des anzuzeigenden Wert aus der Ergebnisliste vereinfacht. Standardmäßig wählt diese Funktion immer das erste Ergebnis der Ergebnisliste und daraus die Eigenschaft mit dem höchsten Index – also die Variable, die in der (übersetzten) SPARQL-Abfrage als letztes im SELECT-Ausdruck aufgeführt wurde. Enthält die Eigenschaft nicht nur eine Abfrage, sondern eine Kombination aus Abfragen und statischem Text, so werden diese, dem zuvor beschriebenen Verhalten entsprechend, automatisch zu einer einzigen Zeichenkette zusammengeführt und zurückgeliefert.

Soll dieses Verhalten geändert werden, kann über diese Eigenschaft eine Auswahlfunktion definiert werden. Dieser wird die SPARQL-Ergebnisliste in vorverarbeiteter Form übergeben und muss einen String zurückliefern. Der Quelltext der Auswahlfunktion kann entweder direkt innerhalb dieser Eigenschaft definiert werden oder es kann auf eine global definierte JavaScript-Funktion verwiesen werden.

`*-transform`: Soll das Ergebnis vor der Anzeige manipuliert werden, kann mittels dieser Eigenschaft eine Transformationsfunktion definiert werden. Eine solche Funktion kann beispielsweise dafür genutzt werden einen abgefragten Datumswert im ISO-Format (z. B. »1984-07-12«) in eine lokalisierte Form zu überführen (»12.07.1984«). Auch hier kann der

Listing 3.9: Ein verarbeitetes SemwidgJS-Widget mit nachträglicher Sortierung der Ergebnisse. Das Widget enthält eine SemwidgQL-Abfrage, welche zu den Geburtsorten Albert Einsteins das Gründungsjahr und die deutschsprachige Bezeichnung abfragt. Die Ergebnisliste wird nachträglich anhand der vorletzten Variablen (`last - 1` ⇒ `dbo:foundingYear`) aufsteigend sortiert, sodass Ulm (ca. 854) vor dem Deutschen Kaiserreich (1871) gelistet wird.

```
<div data-sw-widget="SwSimpleList"
    data-swp-value="{dbr:Albert_Einstein.dbo:birthPlace.[dbo:foundingYear,
    ↪ rdfs:label(@lang = 'de')]}"
    data-swp-value-order="last - 1 asc">

  <!-- created by SemwidgJS -->
  <ul>
    <li>Ulm</li>
    <li>Deutsches Kaiserreich</li>
  </ul>

</div>
```

Listing 3.10: Ein SemwidgJS-Widget mit Aktualisierungsintervall. Das Widget zeigt den aktuellen Messwert eines Luftfeuchtigkeitssensor an und aktualisiert diesen jede Minute zur fünften Sekunde.

```
<div data-sw-widget="SwText"
     data-swp-refresh="every 1 minute on the 5th second"
     data-swp-value="Humidity: {sensor.^prop:sensor(prop:type = 'Humidity'
     ↪ ).[prop:measuredAt(@timestart = 'now - 15 min'), prop:value]} %RH"
     data-swp-value-order="last - 1 desc">
</div>
```

Quelltext der Funktion entweder direkt innerhalb dieser Eigenschaft definiert werden oder es kann auf eine global definierte JavaScript-Funktion verwiesen werden.

Zeitgesteuerte Aktualisierung der angezeigten Daten

SemwidgQL enthält mit den (Pseudo-)Filter-Schlüsselwörtern `@timestart`, `@timeend` (vgl. Abschnitt 2.3.3: *Filter-Schlüsselwörter*) und `@timeinterval` (vgl. Abschnitt 2.3.3: *Pseudo-Filter-Schlüsselwörter*) einige Funktionen, die die Abfrage von zeit- und datumsbezogenen Daten stark vereinfacht. Insbesondere die Abfrage von durch Sensoren generierten Daten sollte durch diese Schlüsselwörter erleichtert werden. Da in diesen Fällen speziell die Betrachtung von Live-Daten relevant ist, und SPARQL-Endpoints über keine standardisierten Funktionen verfügen, die über die Aktualisierung von Daten informieren, besteht in SemwidgJS die Möglichkeit, Widgets zeitgesteuert zu aktualisieren, um so stets die aktuellen Daten des Endpoints anzeigen zu können.

Auf Widget- oder Eigenschaftenebene können Aktualisierungsintervalle festgelegt werden. SemwidgJS greift dazu auf die JavaScript-Bibliothek *Later.js*[23] zurück, die aus einfachen natürlich-sprachlichen Angaben[24] komplexe Aktualisierungszeitpläne erzeugen kann. Listing 3.10

[23] `https://bunkat.github.io/later/`
[24] `https://bunkat.github.io/later/parsers#text`

zeigt ein Widget, das den aktuellen Messwert eines Luftfeuchtigkeitssensor anzeigt. Das Widget wird jede Minute zur fünften Sekunde aktualisiert. Sollten sich mehrere Widgets auf einer Webseite befinden, die regelmäßig ihre Daten aktualisieren, ist es ratsam, die Aktualisierungen zeitlich versetzt zu stellen, um den SPARQL-Endpoint so zu entlasten. Im gezeigten Beispiel erfolgt die Aktualisierung daher nicht direkt zu Beginn einer neuen Minute, sondern um fünf Minuten versetzt.

Zwischenspeicherung von Abfrageergebnissen

Während sich die Daten von Sensoren ständig ändern, sind die meisten Daten in SPARQL-Endpoints in der Regel statisch und ändern sich nicht mehr oder nur selten. Eine erneute Abfrage der Daten ist im Laufe einer Browsersitzung meist nicht nötig und es kann auf zwischengespeicherte Daten zurückgegriffen werden. SemwidgJS besitzt zwei Methoden zur Zwischenspeicherung von Daten. Wenn es der Browser unterstützt, nutzt SemwidgJS den persistenten lokalen Speicher, in dem sich Daten in einer einfachen Key-Value-Datenbank ablegen lassen. In diesem Fall greift SemwidgJS standardmäßig eine Woche lang auf diese Daten zurück, bis es die Daten aktualisiert. Die Speicherdauer lässt sich auf Widget- oder Eigenschaftenebene konfigurieren. Sollte SemwidgJS keinen Zugriff auf den persistenten lokalen Speicher des Browsers haben, werden die Daten im Arbeitsspeicher vorgehalten. Die Nutzung des zuvor beschriebenen Aktualisierungsintervalls setzt die Zwischenspeicherung automatisch außer Kraft.

Erstellung benutzerdefinierter Widget-Klassen

Zusätzlich zu den vordefinierten Widget-Klassen der Widget-Bibliothek, lässt sich SemwidgJS als Framework zur Erstellung benutzerdefinierter Widget-Klassen nutzen. Benutzerdefinierte Widget-Klassen werden in JavaScript programmiert und können anschließend wie die vordefinierten Klassen genutzt werden. Obwohl eine benutzerdefinierte Widget-Klasse programmiert werden muss, sind nur rudimentäre JavaScript-Kenntnisse für diese Aufgabe nötig, da das Framework die meisten Aufgaben übernimmt und eine Vielzahl von Hilfsfunktionen zur Verfügung stellt. Die Erstellung einer benutzerdefinierten Widget-Klasse besteht im Wesentlichen aus vier Schritten:

1. Bestimmen der Widget-Eigenschaften, über die die neue Klasse zusätzlich zu den Standardeigenschaften (vgl. Abschnitt 3.3.3: *Widget-Eigenschaften*) verfügen soll,
2. Erben der Funktionen der Basis-Widget-Klasse,
3. Überschreiben der Funktion, die für die Erstellung der HTML-Ausgabe zuständig ist,
4. Registrieren der neuen Widget-Klasse in der Widget-Bibliothek, damit SemwidgJS diese verarbeiten kann.

Alle vordefinierten Widget-Klassen wurden ebenfalls auf diese Weise programmiert. Die benutzerdefinierte Klasse kann anschließend entweder als wiederverwendbare JavaScript-Datei in die Webseite eingebunden oder ihr Quelltext kann direkt in die Seite eingefügt werden. Listing 3.11 zeigt die beispielhaft Definition einer benutzerdefinierten Widget-Klasse, an deren Quelltext im weiten Verlauf die zuvor aufgeführten Schritte genauer beschrieben werden. Die hier vorgestellte Klasse erlaubt die Instanziierung von Widgets, die ein Bild zuzüglich einer Bildunterschrift darstellen können.

Listing 3.11: JavaScript-Quelltext einer benutzerdefinierten Widget-Klasse. Die Programmierung der Klasse erfolgt in vier einfachen Schritten. Die Hauptaufgabe eines Widget-Autors besteht in der Erzeugung des HTML-Quelltextes, wofür ihm eine Reihe von Hilfsfunktionen zur Verfügung stehen. Die anderen Schritte sind in allen Widget-Klassen nahezu gleich und können dementsprechend mit nur kleinen Änderungen von diesen Übernommen werden.

1. Bestimmen der zusätzlichen Widget-Eigenschaften:

```
1  SemwidgJS.Custom.ImageWithCaption = function() {
2    this._addProperties([
3      ['alt', 'query', 'Specifies an alternate text for this image.'],
4      ['caption', 'query', 'Caption text for this image.']
5    ]);
6  };
```

2. Erben der Funktionen der Basis-Widget-Klasse:

```
7  SemwidgJS.Custom.ImageWithCaption.prototype = new SemwidgJS.SwWidget();
```

3. Überschreiben der Funktion zur Erstellung der HTML-Ausgabe:

```
8  SemwidgJS.Custom.ImageWithCaption.prototype._buildHtml = function() {
9
10   var html = '';
11
12   html += '<figure>'
13
14   html += '<img ';
15   html += this._getDefaultHtmlAttributes();
16   if (!SemwidgJS.Util.String.isEmpty(this.getValue())) {
17     html += 'src="' + this.getValueDefaultValue() + '" ';
18   }
19   if (!SemwidgJS.Util.String.isEmpty(this.getAlt())) {
20     html += 'alt="' + this.getAltDefaultValue() + '" ';
21   }
22   html += '>';
23
24   html += '<figcaption>'+ this.getCaptionDefaultValue() +'</figcaption>';
25
26   html += '</figure>'
27
28   this._setHtml(html);
29 };
```

4. Registrieren des neuen Widgets in der Widget-Bibliothek:

```
30 SemwidgJS.WidgetProvider.registerWidgetType('SemwidgJS.Custom.
       ↪ ImageWithCaption');
```

Im ersten Schritt wird die neue Widget-Klasse innerhalb des `SemwidgJS.Custom`-Namespaces erzeugt, der speziell für diesen Anwendungszweck vorgesehen ist. Es ist zwar nicht nötig diesen Namespace zu nutzen, es verhindert aber mögliche Namenskollisionen mit anderen Funktionen. Der nachfolgende Name kann frei gewählt werden. Innerhalb der Funktion werden in diesem Beispiel zwei zusätzliche Eigenschaften definiert, die später genutzt werden können. Die eine für einen Beschreibungstext des Bildes, die andere für die Bildunterschrift.

Für die Bild-URL soll die bereits vordefinierte Standardeigenschaft `value` genutzt werden. Um die Eigenschaften zu bestimmen, müssen ein Eigenschaftenname, der Eigenschaftentyp und ein kurzer Kommentartext, der Aufschluss über den Einsatzzweck eine Eigenschaft liefern soll, angegeben werden.

Noch würde die Instanziierung der neuen Widget-Klasse zu einer Fehlermeldung führen, da die zuvor genutzte Funktion zum Hinzufügen der zusätzlichen Eigenschaften, noch gar nicht für diesen Typ verfügbar ist. Diese Funktion erhält die Klasse erst durch das Erben der Funktionen der Basis-Widget-Klasse im zweiten Schritt.

Im dritten Schritt wird durch Überschreiben der `_buildHtml`-Funktion definiert, wie die HTML-Ausgabe erzeugt werden soll. Dabei kann der Autor der Widget-Klasse auf eine Vielzahl von Hilfsfunktionen zurückgreifen, die die neue Widget-Klasse zum einen von der Basis-Widget-Klasse erbt und die zum anderen speziell als Akzessoren (vgl. Abschnitt 3.3.3: *Automatisch erstellte Akzessoren*) von SemwidgJS für diese Klasse auf Basis der vorhandenen Widget-Eigenschaften generiert wurden. Die HTML-Ausgabe wird dann als Zeichenkette in mehreren Teilschritten zusammengesetzt. In diesem Beispiel wird die Hilfsfunktion `_getDefaultHtmlAttributes()` der Basis-Widget-Klasse genutzt, um die Werte der Eigenschaften `style` und `styleclass` vorformatiert in das Bild-Element einzubetten. Anschließend werden die Standardwerte für die Eigenschaften `value` und `alt` als Bild-URL beziehungsweise als alternativer Text, falls das Bild nicht angezeigt werden kann, dem Bild-Element hinzugefügt. Zuvor wird jeweils geprüft, ob für die Eigenschaft auch ein Wert vorhanden ist, um das Einfügen leere Attribute in den HTML-Quelltext zu vermeiden. Zuletzt wird der Standardwerte für die Eigenschaft als Bildunterschrift genutzt und die zusammengesetzte Zeichenkette wird als HTML-Ausgabe gespeichert.

Im letzten Schritt wird die neue Widget-Klasse an den SemwidgJS-Widget-Provider übergeben. Danach kann sie wie jede andere vordefinierte Widget-Klasse auch, anhand ihres Namens im HTML-Quelltext instanziiert werden. Dabei kann entweder der komplette Name angegeben werden (`SemwidgJS.Custom.ImageWithCaption`) oder es kann der verkürzte Name genutzt werden. Diese ist der Teil hinter dem letzten Punkt des Namens der Klasse (`ImageWithCaption`). Wird nur der verkürze Name genutzt, ist es möglich vordefinierte Widget-Klassen zu überschreiben, indem der gleiche Name genutzt wird. Soll sichergestellt werden, dass die ursprüngliche Version genutzt wird, kann der vollständige Name verwendet werden.

Wie an diesem Beispiel zu erkennen ist, werden für die Erstellung einer benutzerdefinierten Widget-Klasse nur rudimentäre JavaScript- und einige HTML-Kenntnisse benötigt. Der Quelltext aus dem ersten, zweiten und vierten Schritt kann bei der Erstellung einer weiteren Widget-Klasse größtenteils übernommen werden. Lediglich die Widget-Eigenschaften und der Name der Klasse müsste angepasst werden. Dank der Hilfsfunktionen, die SemwidgJS zur Verfügung stellt besteht die Hauptarbeit für einen Widget-Autor in der Erzeugung des HTML-Quelltextes, in den er an den richtigen Stellen die Rückgabewerte dieser Funktionen einsetzen muss.

Listing 3.12 zeigt die Instanziierung des benutzerdefinierten Widgets inklusive des generierten HTML-Quelltextes. Dazu wird erneut auf die im Laufe diese Kapitels bereits genutzten Endpoint- und Resource-Konfigurationselemente zurückgegriffen.

Automatische Erzeugung von Akzessoren

Für jede Widget-Eigenschaft (vgl. Abschnitt 3.3.3: *Widget-Eigenschaften*) werden eine Reihe von Akzessoren (Getter- und Setter-Funktionen) erzeugt. Welche Funktionen genau erzeugt werden, hängt dabei teilweise vom Eigenschaftentyp ab (vgl. Abschnitt 3.3.3: *Eigenschaftentypen*).

Listing 3.12: Instanziierung der benutzerdefinierten Widget-Klasse zuzüglich der erzeugten Ausgabe. Dazu wird auf die bereits zuvor genutzten Endpoint- und Resource-Konfigurationselemente zurückgegriffen

```
<div data-sw-widget="ImageWithCaption"
     data-swp-id="ImageWithCaptionExample"
     data-swp-style="height: 200px;"
     data-swp-value="{physicists.dbpedia-owl:thumbnail}"
     data-swp-alt="An Image of {physicists.rdfs:label(@lang = 'en')}"
     data-swp-caption="An image of {physicists.rdfs:label(@lang = 'en')}">

  <!-- created by SemwidgJS -->
  <figure>

    <img style="height: 200px;"
         src="http://en.wikipedia.org/wiki/Special:FilePath/
        ↪ Einstein_1921_by_F_Schmutzer_-_restoration.jpg?width=300"
         alt="An Image of Albert Einstein">

    <figcaption>An image of Albert Einstein</figcaption>

  </figure>

</div>
```

Für jede Eigengenschaft werden Getter- und Setter-Funktionen generiert, die den unverarbeiteten Wert dieser Eigenschaft zurückliefern beziehungsweise einen neuen Wert setzen können. Durch die Nutzung der Setter-Funktionen ist sichergestellt, dass alle relevanten Events ausgelöst werden, die über das Event-System (vgl. Abschnitt 3.3.1) weitergereicht werden.

Der Name eines Akzessoren entspricht dem Namen der Eigenschaft zuzüglich des Präfixes `get`, welches für alle Getter-Funktionen genutzt wird, beziehungsweise `set`, für alle Setter-Funktionen. Bindestriche in Eigenschaftennamen werden entfernt und es wird die Upper-Camel-Case-Notation genutzt. Weitere Funktionen fügen dem Eigenschaftennamen zusätzlich ein Suffix an. So fügt die automatisch erzeugte Getter-Funktion, die den Kommentartext einer Eigenschaft zurückliefert, das Suffix `PropertyDescription` an den Eigenschaftennamen an (z. B. wird für die Eigenschaft `value` u. a. die Funktion `getValuePropertyDescription()` erzeugt).

Zusätzlich werden für jede Query-Eigenschaft drei weitere Getter-Funktionen erstellt, die Autoren neuer Widget-Klassen bei der Erstellung der HTML-Ausgabe unterstützen können:

`*DefaultValue`: Diese Funktion liefert einen Standardwert für eine Eigenschaft in Form einer einzelnen Zeichenkette zurück, die direkt in die HTML-Ausgabe eingebettet werden kann, ohne dass der Autor sich weitere Gedanken über das Zusammenfügen von Abfrageergebnissen und statischen Eingaben machen muss. Die Funktion kann bei Bedarf über die Angabe einer Auswahlfunktion (vgl. Abschnitt 3.3.3: *Zusätzliche Query-Eigenschafen*) überschrieben werden.

`*JoinedValues`: Diese Funktion liefert das kartesische Produkt $(X_1 \times \cdots \times X_n)$ aus allen einzelnen Bestandteilen (X_i) einer verarbeiteten Eigenschaft zurück. Die Daten werden allerdings nicht als Zeichenkette, sondern in Form einer verschachtelten Liste zurückgeliefert. Ebenso wird kein Standardwert für das Ergebnis einer Abfrage ausgewählt, sondern es

werden alle Ergebnisse einer Abfrage zurückgeliefert, sodass die Auswahl eines passenden Wertes im Aufgabenbereich des Widget-Autors liegt. Mit dieser Funktion lassen sich leicht Listen und Tabellen aus den abgefragten Daten generieren. Da diese Funktion extrem große Ergebnisse produzieren kann, wenn eine Eigenschaft mehrere Abfragen mit vielen Ergebnissen enthält ($\prod |X_i|$), kann ihr ein Parameterwert für die maximale Länge der inneren Listen übergeben werden.

`*AllValues`: Diese Funktion liefert eine verschachtelte Liste mit den einzelnen Bestandteilen einer verarbeiteten Eigenschaft zurück. Im Gegensatz zu der `*JoinedValues`-Funktion müssen die einzelnen Listen nicht die selbe Länge aufweisen. Ansonsten sind die Ergebnisse dieser beiden Funktionen gleich aufgebaut.

Wiederverwendbarkeit von Abfragen und Widgets

Die bisherige Systeme ermöglichen in der Regel keine Wiederverwendung der abgerufenen Rohdaten durch andere Nutzer. Die Daten werden für gewöhnlich nach der Abfrage aufbereitet und anschließend auf unterschiedliche Weise dargestellt. Die Visualisierung beziehungsweise Datenpräsentation bildet somit eine Sackgasse für die abgerufenen Daten. Der Nutzer hat nur noch Zugriff auf diese Darstellung. Bei clientseitig ausgeführten Systemen wie Sgvizler (vgl. Skjæveland, 2012) können Nutzer noch die verwendeten Abfragen aus dem Quelltext der Webseite extrahieren und anschließend die Rohdaten eigenständig abfragen. Für alle serverseitig ausgeführten Systeme besteht eine solche Möglichkeit nicht.

Damit interessierte Nutzer nicht gezwungen sind, den Quelltext von SemwidgJS-Widgets nach für sie relevanten Abfragen zu durchsuchen, wurde eine Funktion implementiert, die diese Arbeit überflüssig macht. Über einen Kontextmenüeintrag, über das jedes Widget automatisch verfügt, kann sich ein Nutzer sämtliche SemwidgQL-Abfragen, deren Übersetzung nach SPARQL und die Rohdaten, die diese Abfragen zurückgeliefert haben, anzeigen lassen. Weiterhin lässt sich der vollständige Quelltext eines Widgets zusammen mit dem aller relevanten Konfigurationselemente anzeigen, sodass Widgets direkt in andere Webseiten übernommen werden können.

3.3.4 Templates

Templates ermöglichen die Wiederverwendung von Kompositionen aus Widgets und anderem HTML-Inhalten. Eine Template-Definition enthält HTML-Quelltext, welche wiederum Widgets enthalten kann. Teile dieser Inhalte können durch Platzhalter ersetzt werden. Durch Instanziierung dieser Template-Definitionen mit entsprechenden Parameterwerten für die Platzhalter können Template-Instanzen erstellt werden. Auf einer Webseite können beliebig viele Instanzen einer Template-Definition erzeugt werden. Weitere Informationen zu Template-Definitionen und Template-Instanzen sind in Anhang D.3 beschrieben.

Template-Definitionen

Wie auch die Konfigurationselemente, werde Template-Definitionen mittels generischer HTML-Elemente (`div`, `span`) erzeugt, enthalten im Gegensatz zu diesen aber nicht nur Attribute, sondern auch einen inneren HTML-Text. Innerhalb eines Parameter-Attributs wird eine kommaseparierte Liste mit Parameternamen angegeben, die später bei der Instanziierung des Templates genutzt werden können.

Listing 3.13: Eine Template-Definition. Als Attribute werden eine ID und zwei Parameter (`lang` und `person`) definiert. Innerhalb des Elements werden drei Widgets deklariert. Die Start-Resource der SemwidgQL-Abfrage und der Wert für den Sprachfilter wurden durch Platzhalter ersetzt, deren Bezeichnungen den zuvor definierten beiden Parametern entsprechen.

```
1  <div data-sw-config="template"
2       data-swp-id="PersonTemplate"
3       data-swp-parameter="lang, person"
4       style="display: none;">
5
6    <h1  data-sw-widget="SwText"
7         data-swp-value="{%%person%%.rdfs:label(@lang = '%%lang%%')}">
8    </h1>
9
10   <div data-sw-widget="SwText"
11        data-swp-value="{%%person%%.rdfs:comment(@lang = '%%lang%%')}">
12   </div>
13
14   <div data-sw-widget="SwImage"
15        data-swp-value="{%%person%%.dbpedia-owl:thumbnail}">
16   </div>
17
18 </div>
```

Der Inhalt des Definitionselements kann aus beliebigen HTML-Text bestehen und natürlich auch Widgets enthalten. Teile des Inhalts können durch Platzhalter ersetzt werden, deren Bezeichner den zuvor definierten Parameternamen entsprechen und von doppelten Prozentzeichen umschlossen sein müssen. Diese werden auf die selbe Weise verarbeitet, wie die Platzhalter der Mapping-Konfigurationselemente. Während Mappings jedoch global verfügbar sind, sind diese Platzhalter nur innerhalb des Templates nutzbar.

Eine Template-Definition muss nicht zwingend direkt innerhalb einer Webseite vorhanden sein. Um den Grad der Wiederverwendbarkeit von Widget-Kompositionen zu erhöhen, ist es ebenfalls möglich, Template-Definitionen in externe Dokumente zu verlagern und diese nachträglich mittels JavaScript in die Webseite einzubetten. Dies muss entweder vor dem Laden der SemwidgJS-Bibliothek geschehen oder SemwidgJS muss nachträglich über den Vorgang informiert werden, um daraufhin den zusätzlichen Quelltext einzulesen und zu verarbeiten.

Listing 3.13 zeigt ein Beispiel einer Template-Definition. Als Attribute werden eine ID, anhand derer Instanzen dieses Templates erzeugt werden können, und zwei Parameter (`lang` und `person`) definiert. Zusätzlich wird das HTML-Element, welches die Template-Definition bildet unsichtbar geschaltet. Zwar wird die Sichtbarkeit von Template-Definitionselementen bei der Verarbeitung durch SemwidgJS automatisch ausgeschaltet, dennoch ist es ratsam diese Style-Eigenschaft bereits im Voraus zu setzen, um zu verhindern, dass das Element sichtbar ist, bis die SemwidgJS-Bibliothek geladen wurde.

Innerhalb des Elements werden drei Widgets deklariert, die ein Label, einen kurzen Beschreibungstext und ein Bild zu einer Resource anzeigen sollen. Die Start-Resource der SemwidgQL-Abfrage wurde durch einen Platzhalter (`%%person%%`) ersetzt. Ebenso wie der Wert für den Sprachfilter (`%%lang%%`). Widgets innerhalb von Template-Definitionen werden nicht instanziiert, sondern werden wie gewöhnlicher Text behandelt. Die Instanziierung der hier gezeigten Template-Definitionen wird im nächsten Abschnitt erläutert.

Listing 3.14: Instanziierung eines Templates mit verschiedenen Werten. Das in Listing 3.13 vorgestellte Template wird mit verschiedenen Werten für die dort definierten Parameter `person` und `lang` instanziiert.

```
<div data-sw-template="PersonTemplate"
     data-swp-lang="de"
     data-swp-person="dbr:Albert_Einstein">
</div>

<div data-sw-template="PersonTemplate"
     data-swp-lang="en"
     data-swp-person="dbr:Stephen_Hawking">
</div>

<div data-sw-template="PersonTemplate"
     data-swp-lang="fr"
     data-swp-person="dbr:Marie_Curie">
</div>
```

Template-Instanzen

Template-Instanzen werden ähnlich wie Widgets deklariert. Anstatt auf einer Widget-Klasse, basieren sie jedoch auf einer Template-Definition, welche anhand ihrer ID ausgewählt wird. Die Attribute, die bei der Deklaration eine Template-Instanz angegeben werden können, entsprechen den Parameternamen, die bei der Erstellung der entsprechenden Template-Definition angegeben wurden. Listing 3.14 zeigt die Instanziierung des zuvor definierten Templates aus Listing 3.13 mit verschiedenen Werten für den Sprachfilter (`lang`) und die darzustellende Person (`person`). Während der Verarbeitung fügt SemwidgJS dann den inneren Text der entsprechenden Template-Definition in die Instanz-Elemente ein und ersetzt dabei die Platzhalter durch die passenden Parameterwerte. Anschließend werden die eingefügten Widgets ganz normal verarbeitet.

3.3.5 Fazit

Bei der Implementierung von SemwidgJS konnten sämtliche zuvor gestellten Anforderungen (vgl. Abschnitt 3.2.2) umgesetzt werden. SemwidgJS wurde als JavaScript-Bibliothek entwickelt, die sich problemlos in nahezu jede beliebige Webseite einbinden lassen sollte, und mittels Semantic Data Widgets Daten aus SPARQL-Endpoints darstellen und visualisieren kann. Widgets können direkt oder über zusätzliche Elemente miteinander kommunizieren, was die Erstellung interaktiver Webseiten ermöglicht. Mittels der bidirektionalen Kommunikation zwischen den JavaScript-Objekten und HTML-Elementen von SemwidgJS, kann zusätzlich auch mit externen JavaScript-Anwendungen kommuniziert werden, ohne dass eine der beiden Seiten eine spezielle Programmierschnittstelle für eine solche Interaktion zur Verfügung stellen müsste. Templates erlauben zusätzlich die Wiederverwendung von Kompositionen aus Widgets aber auch regulären HTML-Inhalten.

SemwidgJS enthält 17 vordefinierte Widgets, die eine Reihe standardmäßig genutzter Elemente zur Datenpräsentation sowie grafische Bedienelemente in Webseiten erzeugen. Ebenso sind weitere Widgets zur Datenvisualisierung enthalten. Eine vollständige Übersicht über diese Widgets befindet sich in Anhang D.2.4. Bei Bedarf lässt sich SemwidgJS auch als Framework zur Erstellung benutzerdefinierter Widget-Klassen nutzen.

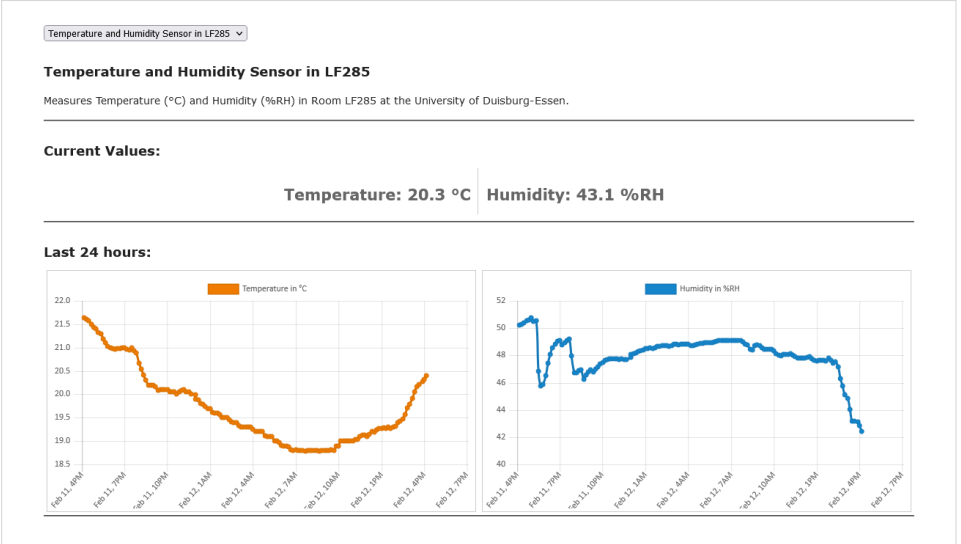


Abbildung 3.3: Ein Teilausschnitt eines Beispiels von der Semwidg-Projektwebseite mit zeitbezogenen Sensordaten. Mittels des oberen Widgets wird der Sensorstandort ausgewählt. Darunter werden Informationen zu diesem Standort angezeigt. In der Mitte werden die aktuellen Messwerte als einfacher Text angezeigt. Im unteren Bereich werden die Messwerte der letzten 24 Stunden in einem Diagramm dargestellt. Die angezeigten Daten werden zyklisch aktualisiert.

Die SemwidgJS-Bibliothek wurde auf der Semwidg-Projektseite[25] unter der MIT-Open-Source-Lizenz veröffentlicht. Zudem befindet sich dort eine Dokumentation der Bibliothek, eine Übersicht der vordefinierten Widgets sowie Quelltextbeispiele, die potenzielle Nutzer beim Einsatz von SemwidgJS unterstützen sollen.

Abbildung 3.3 zeigt einen Teilausschnitt eines Beispiels[26] von der Semwidg-Projektwebseite mit zeitbezogenen Sensordaten. Ein Auswahl-Widget lässt den Nutzer einen Standort aus einer Liste mit allen verfügbaren Sensorstandorten wählen. Darunter werden die aktuellen Messwerte als Text dargestellt. Darunter werden die Messwerte der letzten 24 Stunden in einem Diagramm dargestellt. Die angezeigten Daten werden zyklisch aktualisiert. Auf der Beispielwebseite werden darüber hinaus die Messwerte der letzten sieben Tage sowie der Sensorstandort auf eine Karte angezeigt.

[25] `https://semwidg.org/page/semwidgjs`
[26] `https://semwidg.org/page/semwidgjs-sensor-example`

3.4 Produktiveinsatz

SemwidgJS kam im Rahmen des Eurostars-geförderten Projekts DESUMA (Decision Support Marketplace)[27] zum Einsatz. Unter anderem wurden in diesem Projekt Nutzer-generierte Produktrezensionen aus Onlineshops hinsichtlich beschriebener Produkteigenschaften und der diesbezüglichen Meinungen und Einstellungen der Nutzer analysiert, aufbereitet und in einem SPARQL-Endpoint zur weiteren Verwendung abgelegt. Ein Verwendungszweck war die Unterstützung von Entwicklern von Empfehlungssystemen in Form von Online-Produktberatern, welche in einigen Onlineshops eingesetzt werden, und Kunden bei der Suche nach den für sie bestgeeigneten Produkten in unübersichtlichen und unbekannten Produktdomänen beraten sollen. Der Projektpartner Smart Information Systems (mittlerweile umbenannt in Zuvoo[28]) ist einer der weltweit führenden Anbieter für Online-Produktberatersoftware, die von großen Online-Versandhändlern wie Amazon[29] und Produktherstellern wie Canon[30] oder Bose[31] eingesetzt wird. Anhand der analysierten Produktrezensionen können sich Entwickler einen Überblick über jene Produkteigenschaften verschaffen, die für Kunden vermutlich von besonderer Relevanz sind und ihren entwickelten Produktberater dahingehend optimieren.

3.4.1 Umsetzung

Auf Basis von SemwidgJS wurde ein Webseite – der DESUMA Review Browser – erstellt, welche es Entwicklern ermöglicht, den Datenbestand der analysierten Produktrezensionen zu explorieren, zu filtern und Detailinformationen betrachten zu können. Ein Teilausschnitt dieser Webseite ist in Abbildung 3.4 zu sehen. Die Webseite enthält drei Widgets und einige Konfigurationselemente zu Konfiguration des genutzten SPARQL-Endpoints, zur Definition von Namespaces sowie einige Mapping-Konfigurationselemente die der Kommunikation zwischen den einzelnen Widgets dienen. Für zwei der Widgets wurden Anwendungsfall-spezifische Widget-Klassen implementiert.

Über ein Standard-Auswahl-Widget lassen sich die verschiedenen Produktkategorien auswählen, zu denen Analyseergebnisse in der Datenbank vorliegen. Die Auswahl der Produktkategorie beeinflusst die Datenbankabfrage eines zweiten Widgets. Dieses stellt alle Produkteigenschaften, welche zur ausgewählten Produktkategorie gehören und im Rahmen der Textanalyse mit einer bestimmten Wahrscheinlichkeit als korrekt bewertet wurden, sortiert nach der Häufigkeit der Nennungen innerhalb aller untersuchten Produktrezensionen, als Radiobuttons dar. Die Eigenschaften, welche eine niedrigere Korrektheitswahrscheinlichkeit aufwiesen, werden darunter, aber immer noch innerhalb des selben Widgets, in einem Dropdown-Listenfeld dargestellt. Die Auswahl der Produkteigenschaft beeinflusst wiederum die Datenbankabfrage eines dritten Widgets.

Dieses Widget stellt die Produktrezensionen dar, welche zu der gewählten Produktkategorie gehören und in denen die gewählte Produkteigenschaft erwähnt wurde. Sowohl die gewählte Produkteigenschaft wie auch alle anderen erkannten Eigenschaften werden innerhalb des Texts markiert. Zusätzlich werden die enthaltenen Meinungsäußerungen, positiv oder negativ, farblich hervorgehoben. Der dargestellte Textausschnitt beschränkt sich auf die Sätze, die

[27] https://interactivesystems.info/projects/desuma
[28] https://zoovu.com
[29] https://www.amazon.de/Miele-Produktfinder/b?node=10572773031
[30] https://www.canon.co.uk/lenses/find-the-perfect-lens/
[31] https://www.bose.de/de_de/landing_pages/headphone-assistant.html

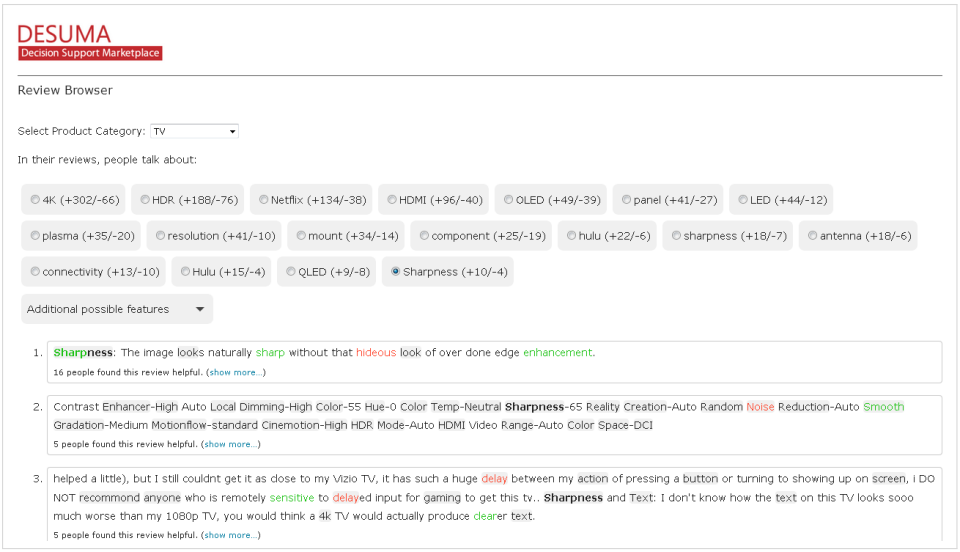


Abbildung 3.4: Ein Teilausschnitt des DESUMA Review Browsers. Über das erste Widget wird die Produktkategorie ausgewählt. Innerhalb des zweiten Widgets wird die Produkteigenschaft gewählt, die näher betrachtet werden soll. Das dritte Widget stellt alle Produktrezensionen dar, die zu der gewählten Produktkategorie gehören und in denen die gewählte Produkteigenschaft erwähnt wurde. Genannte Produkteigenschaften und enthaltenen Meinungsäußerungen werden hervorgehoben.

die gewählte Produkteigenschaft enthalten und soll einen schnellen Überblick über die Eigenschaften und die damit verknüpften Meinungen verschaffen. Um auch den vollen Kontext der Aussagen verstehen zu können, kann bei Bedarf die vollständige Produktrezension angezeigt werden.

3.4.2 Fazit

Die Erstellung der Webseite zur Betrachtung der Analyseergebnisse inklusive der Programmierung der Anwendungsfall-spezifische Widget-Klassen erfolgte innerhalb weniger Stunden. Damit konnten die vorhandenen Ressourcen stärker auf die anspruchsvolleren Projektarbeiten konzentriert werden. Die Einbindung in die bestehende Software zur Erstellung von Produktberaten verlief problemlos. Ebenso kann die Webseite aber auch getrennt von dieser Software genutzt werden, was die Wiederverwendbarkeit in anderen Anwendungskontexten ermöglicht. Für SemwidgJS konnte damit gezeigt werden, dass es auch außerhalb des akademischen Bereichs für den Produktiveinsatz geeignet ist.

3.5 Diskussion und Fazit

In diesem Kapitel wurde mit SemwidgJS eine Widget-Bibliothek und ein Framework zur Präsentation und Visualisierung von in öffentlich zugänglichen SPARQL-Endpoints abgespeicherten Daten vorgestellt. SemwidgJS lässt sich ohne große Umstände in nahezu jede Webseite einbinden und ermöglicht deren Autoren die Durchführung eines umfassenden Arbeitsprozesses, bestehend aus Abfrage, Verarbeitung und Darstellung der Daten, ohne dass diese spezielle Kenntnisse im Linked-Data-Bereich ausweisen müssten.

SemwidgJS ist eine JavaScript-Bibliothek, die Semantic Data Widgets (vgl. Abschnitt 1.2.2) zur Durchführung des oben genannten Arbeitsprozesses nutzt. Diese Widgets werden innerhalb des Quelltextes der Webseite in Form von HTML-Elementen deklariert. Neben der Funktion als Widget-Bibliothek, kann SemwidgJS auch als Framework zur Erzeugung selbstdefinierter Widget-Klassen genutzt werden, die auf die anwendungsspezifischen Anforderungen hinsichtlich ihrer Funktionalität und Datendarstellung hin optimiert wurden. Die Widget-Bibliothek umfasst bereits 17 vordefinierte Widget-Klassen zur Ein- und Ausgabe der Daten in Form üblicher Web-UI-Elemente, sowie zur Visualisierung von numerischen Daten in Form von Diagrammen (vgl. Anhang D.2.4). Kompositionen aus Widgets und regulären HTML-Elementen, die diese Widgets umgeben, können mittels Templates wiederverwendet und angepasst werden.

Die Anforderungen (vgl. Abschnitt 3.2.2), die an die Implementierung von SemwidgJS (vgl. Abschnitt 3.3) gestellt wurden, beruhen auf der Analyse des aktuellen Forschungsstands in diesem Bereich (vgl. Abschnitt 3.1) und den Anforderungen, die die potenziellen Nutzer (vgl. Abschnitt 3.2.1) von SemwidgJS an diese Anwendung stellen. Die Hauptanforderung an SemwidgJS war es, Programmierern und Web-Entwicklern, die keine fortgeschritten Kenntnisse im Linked-Data-Bereich vorweisen können, Daten aus SPARQL-Endpoints in reguläre Webseiten einbetten lassen zu können. Das Endprodukt dieser Arbeit kam im Rahmen des Eurostars-geförderten Projekts DESUMA (Decision Support Marketplace)[32] zum Einsatz und konnte dort seine Nützlichkeit beweisen (vgl. Abschnitt 3.4).

Verglichen mit den zuvor untersuchten Frameworks und Systemen zur Präsentation und Visualisierung von Daten aus dem Linked-Data-Bereich (vgl. Abschnitt 3.1) ist SemwidgJS das einzige System, welches gleichzeitig ein Widget-basiertes Verfahren zur Darstellung von Daten aus SPARQL-Endpoints bereitstellt, welches sowohl die Datenpräsentation mittels üblicher Web-UI-Elemente wie auch die Visualisierung von numerischen Daten in Form von Diagrammen ermöglicht, die Erstellung von interaktiven Webseiten unterstützt, und neben SPARQL eine zusätzliche vereinfachte und leicht zu erlernende Abfragesprache – hier SemwidgQL – verarbeiten kann, wodurch sich der Linked-Data-Bereich von weiteren Nutzergruppen erschließen lassen sollte. Zudem ist SemwidgJS das einzige System, dass eine einfache Wiederverwendung der genutzten Abfragen und abgerufenen Rohdaten erlaubt.

Nichtsdestotrotz weißt SemwidgJS einige Einschränkungen auf, die die Nutzung in bestimmten Einsatzszenarios erschwert oder auch ausschließt. SemwidgJS besitzt aktuell[33] keine Möglichkeit zum Log-in in passwortgeschützte nicht öffentliche SPARQL-Endpoints und ist somit auf die Nutzung öffentlicher SPARQL-Endpoints beschränkt. Es ist allerdings möglich, dass ein Webseitenentwickler das Log-in über eine separate Funktion bereitstellt und SemwidgJS somit den Zugriff auf den entsprechenden Endpoint erlaubt. Allerdings hätte der Endnutzer in

[32] `https://interactivesystems.info/projects/desuma`
[33] Stand: Oktober 2022

diesem Fall auch Zugriff auf eine Vielzahl anderer Daten, die ihm der Betreiber des geschützten Endpoints vielleicht verwehren möchte. Auf Seiten des Endpoints wäre ein sehr komplexes Rechtemanagement nötig, um den Zugriff auf die für einen Nutzer erlaubt einsehbaren Daten einzuschränken. Die meisten Firmen, die Daten in SPARQL-Endpoints gespeichert haben, und diese Daten innerhalb von Webseiten darstellen möchten – ob nur intern oder auch extern zugänglich – werden vermutlich auf eine serverseitige Verarbeitung der Daten setzen, was das Rechtemanagement deutlich einfacher gestaltet.

Ein weiteres Problem, welches alle Systeme betrifft, die auf Daten der LOD-Cloud zugreifen, ist die teilweise unbefriedigende Verarbeitungsgeschwindigkeit der Abfragen und Verlässlichkeit der SPARQL-Endpoints. Für SemwidgJS wurde daher ein Caching-System implementiert, welches einmal erhaltene Abfrageergebnisse zwischenspeichert und deren Wiederverwendung ermöglicht. Allerdings kann dieses System nicht funktionieren, wenn der SPARQL-Endpoint beim ersten Zugriff nicht verfügbar ist. Serverseitige Systeme, welche die Abfrageergebnisse global für alle Nutzer zwischenspeichern und nicht nur auf einen lokalen Nutzer beschränkt sind, können in hierbei effektiver und effizienter arbeiten.

Die SemwidgJS-Bibliothek wurde auf der Semwidg-Projektseite[34] unter der MIT-Open-Source-Lizenz veröffentlicht. Neben einer Dokumentation der Bibliothek[35], befinden sich dort ebenfalls eine Übersicht der vordefinierten Widgets[36] sowie Quelltextbeispiele[37], die potenzielle Nutzer beim Einsatz von SemwidgJS unterstützen sollen.

[34] https://semwidg.org/page/semwidgjs
[35] https://semwidg.org/page/semwidgjs-docs
[36] https://semwidg.org/page/semwidgjs-widgets
[37] https://semwidg.org/page/semwidgjs-example

KAPITEL 4

SemwidgED – Ein Editor für das Einbinden von Semantic Data Widgets in Webseiten

In diesem Kapitel wird mit SemwidgED ein Editor für das Einbinden von SemwidgJS-Widgets in reguläre Webseiten vorgestellt. Dieser richtet sich sowohl an erfahrene Nutzer des Semwidg-Projekts wie auch neue Nutzer, die schnell und auf einfache Weise erproben möchten, ob ein Einsatz von SemwidgJS in ihren eigenen Projekten in Betracht kommt. SemwidgJS und SemwidgQL bieten eine Fülle von Funktionen, die insbesondere neue Nutzer überfordern können. SemwidgED erleichtert den Einstieg, indem er den Nutzer durch den Prozess der Widget-Einbindung führt, ohne erfahrene Nutzer in ihrer Freiheit einzuschränken.

SemwidgED wurde als Online-WYSIWYG-Editor entwickelt und lässt sich in viele Content-Management-Systeme einbinden, sodass deren Nutzer nicht nur statischen HTML-Text, sondern auch dynamische Semantic Data Widgets in ihre Webseiten einbinden können. Neben dem Einbinden von Widgets unterstützt der Editor den Nutzer bei der Erstellung und Bearbeitung von Konfigurationselementen, der Generierung und Instanziierung von Templates sowie der Formulierung von SemwidgQL-Abfragen. Ein Hauptaugenmerk bei der Entwicklung lag auf ebendieser Unterstützung bei der Abfrageformulierung. Für fortgeschrittene Anwender wurde eine textbasierte Eingabeunterstützung implementiert, welche eine möglichst schnelle und effiziente Abfrageerstellung ermöglichen soll. Eine formularbasierte Eingabe richtet sich an weniger erfahrene Anwender, welche im Rahmen dieser Variante stärker angeleitet werden, wodurch Fehler bei der Formulierung der Abfragen vermieden werden sollen. Die Gebrauchstauglichkeit des Editors wurde in einer empirischen Nutzerstudie evaluiert und bestätigt.

Der Editor und sein Quelltext wurden auf der Semwidg-Projektseite[1] veröffentlicht. Zusätzlich wurde ein ausführliches Tutorial erstellt, sowohl in textueller Form wie auch als kommentiertes Video. Eine Demo-Version des Editors kann auf der Webseite direkt getestet werden.

Zu Beginn dieses Kapitels wird der aktuelle Stand der Forschung hinsichtlich der Editoren für Linked-Data-Präsentationen und für die Erstellung von SPARQL-Abfragen beschrieben (vgl. Abschnitt 4.1). Anschließend wird ein Konzept für einen eigenen Editor vorgestellt, der das Einbinden von Semantic Data Widgets in Webseiten ermöglichen soll (vgl. Abschnitt 4.2). Dazu wurden anhand der potenziellen Nutzergruppen und einem Anwendungsszenario verschiedene Anforderungen ermittelt. Aufbauend auf diesem Konzept wurde der Editor entwickelt,

[1] `https://semwidg.org/page/semwidged`

dessen Implementierung nachfolgend beschrieben wird (vgl. Abschnitt 4.3). Eine empirische Nutzerstudie, in welcher die Gebrauchstauglichkeit von SemwidgED evaluiert wurde, wird anschließend vorgestellt (vgl. Abschnitt 4.4). Abschließend werden die Ergebnisse dieses Kapitels diskutiert und ein Fazit gezogen (vgl. Abschnitt 4.5).

4.1 Stand der Forschung

Editoren im Linked-Data-Bereich können unterschiedliche Funktionen erfüllen und ihre Nutzer bei unterschiedlichen Aufgaben unterstützen. In diesem Abschnitt wird daher ein Überblick über relevante Arbeiten gegeben, in welchen Editoren für Link-Data-Präsentationen und Editoren für die Erstellung von SPARQL-Abfragen beschrieben werden. Viele der bereits beschriebenen Systeme (vgl. Abschnitt 3.1) verfügen über einen integrierten Editor für die Erstellung von HTML-Seiten oder Datenbankabfragen. Für die Betrachtung der Editoren für Link-Data-Präsentationen wurden stellvertretend fünf dieser Systeme mit unterschiedlichen Konzepten und unterschiedlichem Unterstützungsumfang ausgewählt, die es Nutzern ermöglichen Daten auszuwählen und damit ein HTML-Dokument zu gestalten. Bei der darauffolgenden systematischen Untersuchung der Editoren für die Erstellung von SPARQL-Abfragen konnten drei Kategorien abgeleitet werden, in welche diese Editoren unterteilt werden können. Jede dieser Kategorien bietet gewisse Stärken und Schwächen bezüglich der Freiheit bei der Formulierung der Abfragen und des Umfangs, in dem der Editor den Nutzer unterstützen kann.

4.1.1 Editoren für Linked-Data-Präsentationen sowie artverwandte Systeme

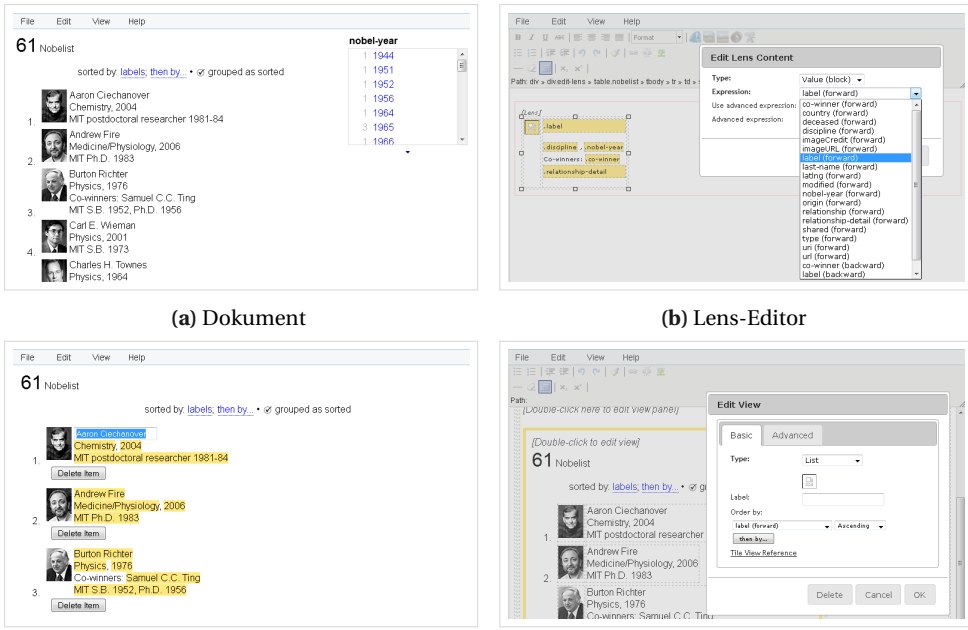
Im weiteren Verlauf werden mit TopBraid Composer, dem LESS Template Editor (vgl. Auer et al., 2010), dem X3S-Editor (vgl. Stegemann et al., 2012) und Uduvudu (vgl. Luggen et al., 2015) verschieden Editoren für die Präsentation von Linked Data vorgestellt. Zusätzlich wird mit Dido (vgl. Karger et al., 2009) ein artverwandtes System vorgestellt, welches zwar keine Daten im RDF-Format nutzt, aber Konzepte von Xenon (vgl. Quan & Karger, 2004b) und Fresnel (vgl. Pietriga et al., 2006) umsetzt. Da diese über keine Editoren verfügen, wird Dido hier stellvertretend betrachtet. Die hier vorgestellten Editoren gehören zu Systemen, die bereits in Abschnitt 3.1 beschrieben wurden. Bei der Analyse der verschiedenen Systeme wird zudem ihre Eignung hinsichtlich der in Abschnitt 1.2.3 beschriebenen potenziellen Nutzer des Semwidg-Projekts untersucht.

TopBraid Composer

TopBraid Composer[2] ist eine Modellierungs- und Entwicklungsumgebung für Ontologien, RDF-Daten und Linked-Data-Anwendungen. Zusätzlich enthält das System einen Server, der diese Anwendungen publizieren kann.

TopBraid Composer verfügt über eine auf Eclipse[3] basierende IDE, welche die Erstellung von SPARQL Web Pages (SWP)[4] ermöglicht. Diese sind in ihrem Aufbau vergleichbar mit den in der Vergangenheit weit verbreiteten JavaServer Pages[5] zur Erstellung von Webanwendungen. SWP-Autoren können HTML-Quelltext (andere Formate wie z. B. XML sind ebenfalls möglich) mit Daten von SPARQL-Endpoints kombinieren. Die Darstellung der Daten kann entweder beliebig frei programmiert werden oder es können vordefinierte »Gadgets« verwendet werden, die das Ergebnis einer SPARQL-Abfrage als Parameter entgegennehmen. Über weitere Parameter lassen sich diese Gadgets konfigurieren. Die IDE verfügt über eine Autocomplete-Funktion für verfügbare Gadgets und deren Parameter.

[2] https://www.topquadrant.com/products/topbraid-composer/
[3] https://www.eclipse.org
[4] https://uispin.org
[5] https://www.oracle.com/java/technologies/jspt.html



(a) Dokument **(b)** Lens-Editor

(c) Data-Editor **(d)** View-Editor

Abbildung 4.1: Ein Dido-Dokument verfügt über vier verschiedene Ansichten. Die grundlegende Dokument-Ansicht und die drei Ansichten für die einzelnen Editoren *Lens*, *Data* und *View*. Über den Lens-Editor können die anzuzeigenden Daten ausgewählt werden. Der Data-Editor ist eine erweiterte Ansicht der Dokumentansicht und ermöglicht es, Daten direkt zu manipulieren. Über den View-Editor kann definiert werden, wie die Daten angezeigt werden sollen.

Im Rahmen einer eigenen Testung der Anwendung, erwies sich die Erstellung von SPARQL Web Pages, im Vergleich zu den anderen hier vorgestellten Verfahren, als sehr aufwendig und erforderte entsprechende Fach- und Programmierkenntnisse. Die Anbindung eines externen SPARQL-Endpoints war vergleichsweise umständlich, da es von den TopBraid-Entwicklern in erster Linie vorgesehen ist, nur in TopBraid Composer integrierte Daten zu publizieren, statt auf externe Quellen zuzugreifen.

Die erstellten SPARQL Web Pages müssen von einer speziellen Serveranwendung verarbeitet werden und lassen sich nicht ohne zusätzlichen Aufwand in bestehende Webseiten einbinden. Dafür hat der SWP-Autor große Freiheiten bei der Gestaltung der Seiten und auch komplexe Linked-Data-Anwendungen lassen sich erstellen. Anwender von TopBraid Composer müssen sowohl über gute Programmierkenntnisse wie auch über hohe Fachkenntnisse im Linked-Data-Bereich verfügen, um die von ihm gewünschten Daten abzurufen, zu verarbeiten und anzuzeigen.

Dido

Das von Karger et al. (2009) entwickelte Dido stellt ein aktives HTML-Dokument dar und enthält neben den anzuzeigenden Daten auch einen Editor, der es erlaubt, diese Daten zu manipulieren, neue Daten hinzuzufügen oder diese zu löschen. Ebenso kann die Darstellung dieser Daten innerhalb des Editors, welcher in JavaScript implementiert wurde, bearbeitet werden.

Abbildung 4.1a zeigt die Dokument-Ansicht einer Dido-Datei in einem Webbrowser. Für die Erstellung der Ansicht wird das Exhibit-Framework (vgl. Huynh et al., 2007) genutzt. Die Daten können nach verschiedenen Kriterien sortiert und gefiltert werden. Innerhalb dieser Ansicht können, wie es in Abbildung 4.1c zu sehen ist, die Daten auch direkt manipuliert und ergänzt werden. Ebenso können komplett neue Elemente hinzugefügt werden. Über den Lens-Editor, wie ihn Abbildung 4.1b zeigt, können die anzuzeigenden Daten ausgewählt werden. In der gleichen Ansicht können die Daten auch in einem HTML-Editor formatiert werden, welcher auf dem WYSIWYG-Editor TinyMCE[6] basiert. Das WYSIWYG-Prinzip beschränkt sich jedoch auf die Formatierung der ausgewählten Eigenschaften. Welche Werte für die ausgewählten Eigenschaften später angezeigt werden, kann der Nutzer so nicht erkennen, sondern muss erst vom Lens-Editor zurück in die Dokument-Ansicht wechseln. Mittels des View-Editors, dargestellt in Abbildung 4.1d, können Nutzer bestimmen, auf welche Weise die Daten angezeigt werden sollen. Dazu gehören die klassischen Darstellungsformen wie die Listen- und Tabellenansicht, aber auch die Anzeige auf einem Zeitstrahl oder in einer Karte.

Im Gegensatz zu den anderen Systemen, welche in diesem Kapitel betrachtet werden, arbeitet Dido nicht mit Daten im RDF-Format, sondern nutzt eine flache Liste von Objekten, bestehend aus einer Menge von Key-Value-Paaren, welche innerhalb des Dokuments im JSON-Format[7] gespeichert werden. Eine Verknüpfung der einzelnen Objekte – ein Grundprinzip von Linked Data – ist nicht vorgesehen. Dementsprechend richtet sich Dido in erster Linie an die Gruppe der Anfänger. Selbstverständlich kann Dido somit auch von allen anderen Nutzergruppen genutzt werden. Doch um für die Gruppe der Programmierer von größerem Interesse zu sein, fehlt es Dido an der Möglichkeit der stärkeren Individualisierbarkeit der Datendarstellung. Da die Daten im JSON- statt im RDF-Format abgelegt werden, werden auch die Linked-Data-Experten kein größeres Interesse an dieser Anwendung haben.

LESS Template Editor

LESS (vgl. Auer et al., 2010) ist ein System zur Darstellung von Linked Data mithilfe von Templates, die sich mittels JavaScript oder als iFrame in beliebige Webseiten einbinden lassen.

Der LESS Template Editor (vgl. Abbildung 4.2) ermöglicht die Erstellung von Templates für Daten auf Basis eines online verfügbaren RDF-Dokuments oder auf Basis einer SPARQL-Abfrage. Die URL des RDF-Dokuments oder die SPARQL-Abfrage lassen sich bei der späteren Nutzung des fertiggestellten Templates austauschen. Das Template wird dann auf jede Resource des RDF-Dokuments beziehungsweise auf jedes Ergebnis des SPARQL Result Sets angewandt. Die Eigenschaften, die mit dieser Resource verknüpft sind, oder die Rückgabevariablen der SPARQL-Abfrage werden dem Template-Autor aufgelistet und können in den eigentlichen Template-Code übernommen werden. Zur Erstellung der Templates wird die deklarative Sprache *LeTL* (LESS Template Language) verwendet, welche auf der Smarty Template Language[8] basiert. Die Sprache ermöglicht es Template-Autoren, HTML-Quelltext (andere Formate wie z. B. XML sind ebenfalls möglich) und die abgefragten Daten miteinander zu kombinieren. Die Daten können nach Sprache oder Typ gefiltert und einfache Pfadabfragen können definiert werden. Ebenso können bedingte Abfragen und Schleifen genutzt werden.[9] Das Template wird anschließend

[6] `https://www.tiny.cloud`
[7] `https://json.org`
[8] `https://www.smarty.net`
[9] Schleifen funktionieren innerhalb des LESS-Systems allerdings nur bei Daten aus RDF-Dokumenten wie erwartet, jedoch nicht bei Daten aus SPARQL-Abfragen. Ein Implementierungsfehler sorgt dafür, dass bei der Iteration über Daten aus SPARQL-Abfragen, diese mehrfach angezeigt werden.

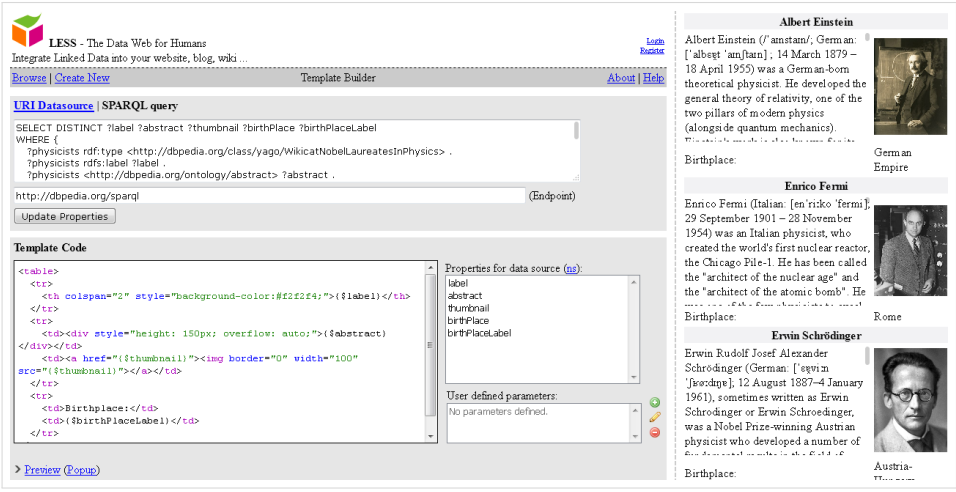


Abbildung 4.2: Ansicht des LESS-Template-Editors. Auf der linken Seite wird der Editor angezeigt, auf der rechten Seite die Vorschauansicht (diese befindet sich ursprünglich unterhalb des Editors, wurde zur besseren Lesbarkeit in dieser Abbildung jedoch verschoben). Im oberen Bereich des Editors werden die anzuzeigenden Daten entweder direkt über die Angabe einer URL zu einem RDF-Dokument oder über eine SPARQL-Abfrage bestimmt. In der Mitte wird das Template mittels der Template-Sprache *LeTL* definiert. Die verfügbaren Eigenschaften können aus der danebenstehenden Liste ausgewählt werden.

auf dem LESS-Server gespeichert und kann von dort abgerufen und in beliebige Webseiten eingebunden werden.

Eine großer Vorteil des LESS-Editor ist es, dass in den Fällen, in denen Template-Autoren nur Daten eines einzelnen RDF-Dokuments darstellen möchten, sie keine eigene Abfrage stellen, sondern nur die entsprechende URL des Dokuments angeben müssen. Nachteilig ist allerdings, dass die Autoren in der Lage sein müssen, HTML-Quelltext zu formulieren und, bei komplexeren Templates, diesen auch mit LeTL zu kombinieren. Der Editor unterstützt die Autoren hier nur auf einer sehr rudimentären Basis. Auch liefert der Editor keine Informationen über die Eigenschaften von verlinkten Resources. Bei der Formulierung von Pfadabfragen müssen Template-Autoren also über das Wissen verfügen, welche Daten in dem abgerufenen RDF-Dokument enthalten sind. Das System richtet sich also an die Programmierer, welche jedoch schnell an ihre Grenzen stoßen können, wenn sie nicht nur die Daten eines einzelnen RDF-Dokuments darstellen möchten. In diesen Fällen sind die Fähigkeiten des Allrounders von Nöten, welcher sowohl über die benötigten Programmierkenntnisse verfügt, die zum Erstellen der Templates erforderlich sind, wie auch über die Fachkenntnisse im Linked-Data-Bereich, über welche es zur Abfrage der Daten bedarf.

X3S-Editor

X3S (XSL-Transformed SPARQL-Results and Semantic Stylesheets) beschreibt einen Prozess, der auf Basis von CSS-Regeln, XSL-Transformationsregeln, Datenstrukturinformationen und einer SPARQL-Abfrage Instanzdaten einer bestimmten RDF-Klasse visuell in einem »Semantic Stylesheet« aufbereitet (vgl. Stegemann et al., 2012).

Nutzer des X3S-Editors können eine RDF-Klasse vorgeben, deren Instanzen sie darstellen möchten. Stylesheets werden in einer Baumhierarchie aufgebaut, sodass verlinkte Resources

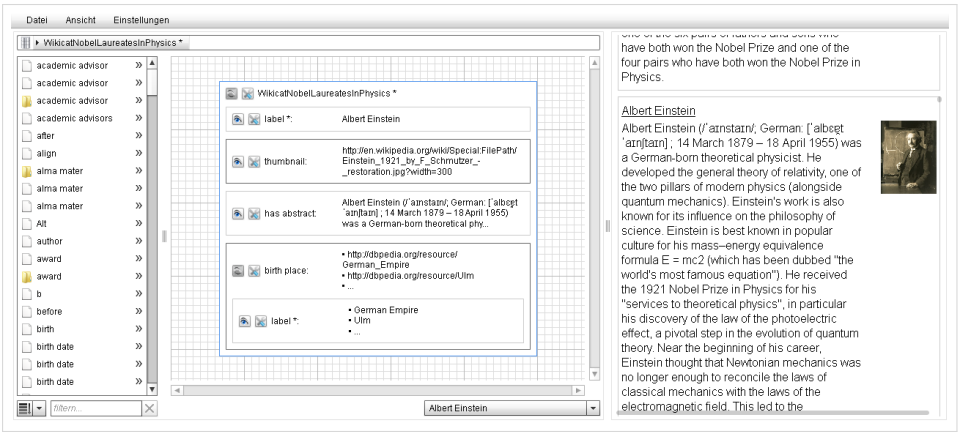


Abbildung 4.3: Editor zur Erstellung von »Semantic Stylesheets« nach dem X3S-Verfahren. Das Stylesheet wird hier für Daten der RDF-Klasse *Physik-Nobelpreisträger* aus der DBpedia-Datenbank erstellt. Im linken Bereich lassen sich die anzuzeigenden Eigenschaften auswählen, in der Mitte wird das Stylesheet konfiguriert und im rechten Bereich wird eine Live-Vorschauansicht des Stylesheets dargestellt.

verschachtelt hinzugefügt werden können (vgl. Abbildung 4.3). Alle verfügbaren Eigenschaften der Instanzen beziehungsweise der verlinkten Resources werden dem Nutzer kontextabhängig in einer Liste dargestellt und können mittels Drag & Drop in den Arbeitsbereich übernommen werden. Nachdem eine Eigenschaft hinzugefügt wurde, kann der Nutzer deren Darstellung konfigurieren, wobei die Konfigurationsmöglichkeiten durch den Editor stark eingeschränkt werden. Dem Nutzer wird immer eine Live-Vorschauansicht des Stylesheets angezeigt. Das Endergebnis kann als statisches HTML-Dokument oder als X3S-Datei gespeichert werden.

Ein großer Vorteil des X3S-Editors ist es, dass Stylesheet-Autoren weder Datenbankabfragen, noch HTML-Quelltext oder CSS-Regeln formulieren müssen. Bis auf die initiale Angabe der RDF-Klasse und gegebenenfalls selbstdefinierte Filterwerte, müssen die Nutzer keine Eingaben vornehmen, sondern können die Stylesheets »zusammenklicken«. Allerdings wird dadurch die Möglichkeit zur individuellen Stylesheet-Gestaltung stark eingeschränkt. Es können lediglich die CSS-Regeln angewandt werden, die der Editor vorgibt und komplexere SPARQL-Abfragen werden nicht unterstützt. Bereits Anfänger können Webseiten mit komplexe Datendarstellungen mittels X3S erzielen. Große Datenbanken wie DBpedia, die Daten vieler verschiedener Klassen mit heterogenen Eigenschaften aufweisen, können für diese Nutzergruppe jedoch eine große Herausforderung darstellen. In diesen Fällen bedarf es der Fachkenntnisse der Linked-Data-Experten.

Uduvudu

Uduvudu (vgl. Luggen et al., 2015) ist ein Framework zur Erstellung von Linked-Data-UIs auf Basis von JavaScript. Es erlaubt das Abrufen von Daten von SPARQL-Endpoints sowie deren Weiterverarbeitung und Darstellung in Webseiten. Zur Darstellung lassen sich *Templates* definieren, die automatisch auf die entsprechenden Eigenschaften angewandt werden können. Templates bestehen aus HTML-Quelltext mit eingebetteten Abfragen in einer einfachen Pfadabfragesprache. Zusätzlich lässt sich zu jedem Template eine JavaScript-Funktion definieren, die nach der Anwendung des Templates ausgeführt wird.

Tabelle 4.1: Vergleich der ausgewertete Eigenschaften der Editoren für Linked-Data-Präsentationen.

	ohne Programmierkenntnisse	Freiheiten bei Gestaltung	Unterstützung bei Erstellung
TopBraid Composer	–	✓	–
Dido	✓	–	✓
LESS Template Editor	–	✓	–
X3S-Editor	✓	–	✓
Uduvudu	–	✓	–

Das Framework verfügt über einen rudimentären Editor in einem frühen Entwicklungsstadium, welcher sich bei Bedarf innerhalb der Webseite zuschalten lassen kann. Dieser ermöglicht es dem Seitenersteller, den Quelltext der genutzten Templates anzupassen oder neue Templates hinzuzufügen. Der Editor ist an die Daten gebunden, die bereits aus der Datenbank abgerufen wurden. Das Hinzufügen von Daten oder die Änderung der SPARQL-Abfrage sind nicht vorgesehen.

Die Nutzer des Editors werden bei der Definition der Templates durch diesen nicht unterstützt. Ihnen wird lediglich ein einfaches Textfeld angezeigt, in welchem sie den Template-Quelltext formulieren oder bearbeiten können. Ebenso ist es innerhalb des Editors nicht möglich, die mit dem Template verbundene JavaScript-Funktion zu erstellen oder zu bearbeiten. Allerdings erlaubt die freie Definition des Templates dem Nutzer einen großen Handlungsspielraum, sodass auch komplexe Darstellungen möglich sind. Da der Editor über keine Speicherfunktion verfügt, gehen jedoch sämtliche Änderungen nach Verlassen der Webseite verloren. Die benötigten Java-Script-Kenntnisse schränken die möglichen Nutzergruppen stark ein. Programmierer können bereits gute Ergebnisse mit dem integrierten Editor erzielen, um jedoch auch komplexe Seiten zu Erstellen, werden zusätzlich auch fortgeschrittene Kenntnisse im Linked-Data-Bereich benötigt.

Zusammenfassung und Fazit

Bei der Betrachtung der Editoren sowie deren jeweiligen Stärken und Schwächen ließen sich drei wichtige Eigenschaften für einen guten Editor ableiten. Der Editor sollte sich ohne fortgeschrittene Programmierkenntnisse nutzen lassen, dem Nutzer viele Freiheiten bei der Gestaltung geben und ihn gleichzeitig bei der Erstellung der Webseite unterstützen.

Insbesondere die freie Gestaltungsmöglichkeit der Darstellung der Daten durch den Nutzer und die Nutzbarkeit des Editors ohne Programmierkenntnisse stehen sich bei den hier untersuchten Editoren diametral entgegen (vgl. Tabelle 4.1). Je mehr Freiraum dem Nutzer bei der Gestaltung der Webseite gegeben wird, umso mehr Wissen muss er im Bereich der Webprogrammierung vorweisen können. Die Editoren, die den Nutzer stark unterstützen, sind auch die, die nur wenige Gestaltungsmöglichkeiten bieten. Im Umkehrschluss unterstützen Editoren mit einem großen Gestaltungsspielraum den Nutzer kaum, sondern erwarten vorrangig, dass der Nutzer Quelltext eingibt. Somit haben diese Eigenschaften stets einen großen Einfluss auf die potenziellen Nutzergruppen der verschiedenen Editoren (vgl. Tabelle 4.2). Leider bietet keines der untersuchten Systeme ein ausgewogenes Verhältnis zwischen Unterstützung und freier Gestaltung.

Tabelle 4.2: Vergleich der Zielgruppen der Editoren für Linked-Data-Präsentationen im Hinblick auf die zuvor definierten Nutzergruppen (vgl. Abschnitt 1.2.3).

	Anfänger	Programmierer	Linked-Data-Experten	Allrounder
TopBraid Composer	–	–	–	✓
Dido	✓	–	–	–
LESS Template Editor	–	✓	–	✓
X3S-Editor	✓	–	✓	–
Uduvudu	–	✓	–	✓

4.1.2 Editoren für die Erstellung von SPARQL-Abfragen

Nachfolgend sollen verschiedene Editoren beschrieben werden, die die Nutzer bei der Erstellung von Abfragen im Linked-Data-Bereich unterstützen. Eine Vielzahl von Systemen erlaubt es Experten und auch Anfängern hinsichtlich Linked Data, teilweise sehr komplexe Abfragen zu erzeugen, mit denen sich Daten von SPARQL-Endpoints abrufen lassen (z. B. Relfinder, vgl. Heim et al., 2009; gFacet, vgl. Heim et al., 2010; SAMPO-UI, vgl. Ikkala et al., 2021). Die Abfragen lassen sich jedoch nicht immer extrahieren und wiederverwenden, sondern werden ausschließlich intern von diesen Systemen genutzt und ausgewertet. Daher sollen hier nur Editoren betrachtet werden, die für den Nutzer eine SPARQL-Abfrage erzeugen, die er im weiteren Verlauf für seine Zwecke weiterverwenden kann. Auf Basis einer systematischen Analyse einer Vielzahl von Editoren wurde eine Kategorisierung erarbeitet, in die sich diese Editoren einordnen lassen:

Textbasierte Eingabe: Die Nutzer formulieren wie gewohnt ihre Abfrage in textueller Form. Dabei unterstützt der Editor sie beispielsweise mittels Autovervollständigung und Syntaxhervorhebung. Beispiele für Editoren in dieser Kategorie sind der Flint SPARQL Editor[10] und YASGUI (vgl. Rietveld & Hoekstra, 2013).

Formularbasierte Eingabe: Die Nutzer stellen die Abfrage mithilfe von Formularen zusammen, welche die Struktur der Abfrage vorgeben. Die einzelnen Elemente der Abfrage wie Variablen, Resources, Eigenschaften und Filterwerte werden mittels Textfelder oder Dropdown-Listen eingegeben. Beispiele für Editoren in dieser Kategorie sind der DBpedia Graph Pattern Builder (vgl. Auer et al., 2007) und Konduit VQB (vgl. Ambrus et al., 2010).

Graphbasierte Eingabe: Die Nutzer erzeugen in einem grafischen Editor einen Graphen, der anschließend als Abfrage interpretiert wird. Sie folgen dabei in der Regel dem Prinzip von *Query by Example* (vgl. Zloof, 1975). Die Knoten und Kanten, die stellvertretend für die Werte stehen, die der Nutzer ausgegeben haben möchte, werden als variabel gekennzeichnet. Die anderen werden mit festen Werten versehen und schränken die Abfrage entsprechend ein. Einige der Editoren können diese Werte mittels Autovervollständigung

[10] https://github.com/TSO-Openup/FlintSparqlEditor

vorschlagen. Beispiele für Editoren in dieser Kategorie sind Semantic Crystal (vgl. Kaufmann & Bernstein, 2007), NIGHTLIGHT (vgl. Smart et al., 2008), SPARQLinG (vgl. Hogenboom et al., 2010), Smeagol (vgl. Clemmer & Davies, 2011), QueryVOWL (vgl. Haag et al., 2015), VESPa (vgl. Haag et al., 2016), iSPARQL[11], Visual SPARQL Builder[12] und Gruff[13].

Im Folgenden wird aus jeder der zuvor vorgestellten drei Kategorien ein Stellvertreter exemplarisch näher beschrieben. Anhand dieser spezifischen Beschreibungen lassen sich einige allgemeine Probleme und Nachteile der Editoren der jeweiligen Kategorien aufgezeigen.

YASGUI

YASGUI (vgl. Rietveld & Hoekstra, 2013) ist ein Online-Text-Editor für SPARQL-Abfragen mit Syntaxhervorhebung und Autovervollständigung für einige Elemente der Abfrage. So werden Präfixe von Graphtermen, falls bekannt, automatisch der Präfixdeklaration der Abfrage hinzugefügt. Ebenso können bereits verwandte Variablenbezeichnungen und innerhalb des SPARQL-Endpoints enthaltene Prädikate vervollständigt werden. Die Ergebnisse der Abfrage können in der Form ausgegeben werden, in der sie vom SPARQL-Endpoint zurückgeliefert werden oder als Tabelle, Diagramm oder in Kombination mit einer Karte dargestellt werden. Logische Voraussetzung für die beiden letztgenannten Darstellungsformen ist natürlich die inhaltliche Eignung für die entsprechende Darstellung.

Bei der Einhaltung der korrekten Struktur der Abfrage wird der Nutzer jedoch nicht unterstützt. Auch die Autovervollständigung kann zu Problemen führen, wenn beispielsweise ein falsches Präfix aufgelöst wird oder Prädikate vorgeschlagen werden, die Abfragen ohne Ergebnisse erzeugen, da der Kontext des Prädikats nicht beachtet wird.

Konduit VQB

Konduit VQB (vgl. Ambrus et al., 2010) ist ein formularbasiertes System zur Erstellung von SPARQL-Abfragen. Es ermöglicht die Generierung von Schema-basierten und Instanz-basierten SELECT- und CONSTRUCT-Abfragen. Bei den Schema-basierten Varianten werden die Abfragen durch Dropdown-Listen zusammengestellt. Dabei wird eine hierarchisch gegliederte Liste erzeugt, bei der die Elternknoten das Subjekt eines Triple-Patterns repräsentieren und Prädikat und Objekt eingerückt darunter dargestellt werden. Die Elemente der Triple-Pattern-Relation können als Variable markiert oder durch eine Klasse oder Eigenschaft festgelegt werden. Die Objekte können zusätzlich durch Filterwerte eingeschränkt werden. Gültige Klassen und Eigenschaften werden vom Editor vorgeschlagen.

Dadurch, dass die Struktur vom Editor vorgegeben wird, können nur syntaktisch korrekte Abfragen erstellt werden. Allerdings wird die Mächtigkeit von SPARQL entsprechend eingeschränkt, da nur noch Abfragen erstellt werden können, die eine hierarchische Struktur aufweisen. Abfragen mit einer Graphstruktur werden nicht unterstützt, ebenso wenig wie Gruppierungen und Vereinigungen.

QueryVOWL

QueryVOWL (vgl. Haag et al., 2015) ist eine grafische Abfragesprache, welche in SPARQL übersetzt wird. Sie orientiert sich an der VOWL-Notation (vgl. Lohmann et al., 2016) zur grafischen

[11] https://www.openlinksw.com/isparql/
[12] https://leipert.github.io/vsb/
[13] https://allegrograph.com/gruff2/

Darstellung von Ontologien. Die Abfrage wird in Form eines Graphen erstellt. Nutzer können Knoten erzeugen, welche entweder eine bestimmte Resource oder Klasse repräsentieren oder variabel sind. Diese Knoten können über Kanten verknüpft werden, welche wieder entweder mit einer festen Eigenschaft versehen werden oder variabel bleiben. Die möglichen festen Werte werden vom Editor derart eingeschränkt, dass nur Werte gewählt werden können, mit denen ein nicht-leeres Ergebnis zurückgeliefert wird. Bei der Auswahl eines variablen Knotens werden die entsprechenden Ergebnisse für diesen angezeigt. Die generierte SPARQL-Abfrage kann exportiert werden.

Bedingt durch seine graphbasierte Notation schränkt QueryVOWL die Mächtigkeit von SPARQL stark ein. Filter lassen sich nur für Literale angeben und auch nur mittels Konjunktion verknüpfen. Gruppierungen, Vereinigungen und Aggregatfunktionen werden nicht unterstützt. Das Erzeugen von komplexen Abfragen ist damit nicht möglich.

Zusammenfassung und Fazit

Die Analyse des aktuellen Forschungsstands der Editoren zur Erstellung von SPARQL-Abfragen hat gezeigt, dass die verschiedenen Kategorien unterschiedliche Stärken und Schwächen aufweisen. Während bei der textbasierten Eingabe das volle Potenzial von SPARQL ausgeschöpft werden kann, können die entsprechenden Editoren nur eine sehr begrenze Hilfestellung leisten. Da die Triple-Patterns der SPARQL-Abfrage an beliebiger Stelle des WHERE-Teils (wenn Vereinigungen und Unterabfragen außer Betracht gelassen werden) und in beliebiger Reihenfolge auftreten können, kann der Editor nur schwer ermitteln, was der Nutzer abfragen möchte oder kann. Die möglichen unterstützenden Funktionen dieser Editoren beschränken sich daher auf Syntaxhervorhebung, Autovervollständigung und die Vorformulierung von bestimmten Abfragestrukturen, die dem Nutzer ein Grundgerüst bieten, das dieser vervollständigen oder anpassen kann.

Die formularbasierte Eingabe gibt dem Nutzer eine Struktur vor, anhand derer er seine Abfrage formulieren muss. Durch diese Einschränkung kann der Editor bessere Vorschläge bei der Autovervollständigung machen, da ermittelt werden kann, welche Werte eine valide Abfrage erzeugen. Allerdings unterstützt keiner der untersuchten Editoren den vollständigen Funktionsumfang von SPARQL, sodass sich nur ein Bruchteil der möglichen SPARQL-Abfragen formulieren lässt.

Ein allgemeines Problem bei der graphbasierten Eingabe ist es, dass sich Filter nur unzureichend in Graphform darstellen lassen, da sie keinen originäreren Teil des Subgraphen darstellen, welcher mit der vollständigen Datenbasis des SPARQL-Endpoints abgeglichen werden soll. Sie wirken daher immer als Fremdkörper in der graphbasierten Notation und werden von einigen Editoren daher nur unzureichend oder auch gar nicht unterstützt. Das gleiche gilt auch für Gruppierungen, Vereinigungen, Unterabfragen und Aggregatfunktionen. Die graphbasierte Eingabe wirkt zwar auf den ersten Blick als logischster Kandidat für die Formulierung von SPARQL-Abfragen, da sie die gleiche Struktur wie die Datenbasis aufweist. Aufgrund der zuvor aufgezählten Probleme, ist sie tatsächlich aber die ungeeignetste Variante.

Resümierend lässt sich festhalten, dass sich die Freiheit bei der Formulierung von Abfragen und die Möglichkeiten zur Unterstützung durch den Editor entgegen stehen. Je stärker der Nutzer von einem Editor bei der Formulierung seiner SPARQL-Abfragen unterstützt werden soll, umso stärker muss der Funktionsumfang von SPARQL eingeschränkt werden.

4.2 Konzeptionelle Überlegungen

In diesem Abschnitt werden die Anforderungen hinsichtlich der Erstellung des Semwidg-Editors – SemwidgED – definiert. Zu Beginn wird kurz auf die potenziellen Nutzer des Editors eingegangen und anschließend ein Anwendungsszenario beschrieben, das aufzeigt, welche Funktionen von SemwidgED unterstützt werden sollten. Auf Grundlage dieser Erkenntnisse, der Untersuchung des Stands der Forschung (vgl. Abschnitt 4.1) und den allgemeinen Anforderungen an SemwidgJS (vgl. Abschnitt 3.2.2) werden abschließend die Anforderungen an SemwidgED erarbeitet.

4.2.1 Potenzielle Nutzer des Editors

SemwidgED soll ein Editor sein, der von möglichst vielen Personen genutzt werden können soll. Hauptzielgruppe des Editors ist, wie für das gesamte Semwidg-Projekt, die Gruppe der Programmierer (vgl. Abschnitt 1.2.3). Allerdings rücken hinsichtlich SemwidgED nun auch die Anfänger und Linked-Data-Experten etwas stärker in den Fokus. Die vereinfachte Nutzbarmachung von SemwidgJS und SemwidgQL durch den Editor soll es diesen beiden Gruppen zudem ermöglichen, die grundlegenden Funktionen des Semwidg-Projekts zu erproben, um sich anschließend gegebenenfalls in die weiterführenden Funktionen einzuarbeiten.

4.2.2 Anwendungsszenario

Nachfolgend wird ein fiktives Anwendungsszenario beschrieben, anhand dessen im nachfolgenden Abschnitt die Anforderungen an SemwidgED abgeleitet werden. Es wurde versucht ein möglichst realitätsnahes Szenario zu finden, welches möglichst viele Funktionen des Semwidg-Projekts abdeckt. Gleichzeitig wurde dieses Szenario auf die nötigsten Arbeitsschritte reduziert um Doppelungen bei der Beschreibung zu vermeiden. Diese werden bei der realen Nutzung des Editors sicher häufig auftreten.

Schulprojekt zum Thema »Nobelpreis für Physik«

Für ein Schulprojekt sollen einige Schüler eine Website über den Nobelpreis für Physik erstellen. Auf einer Unterseite sollen Detailinformationen zu den einzelnen Preisträgern angezeigt werden. Da ihnen für das Projekt kein Budget zur Verfügung gestellt wird, entscheiden sie sich dazu, einen kostenfreien Dienst zu nutzen, der ihnen etwas Speicherplatz und eine Datenbank zur Verfügung stellt. Der Server kann zwar PHP-Programme ausführen, doch es besteht keine Möglichkeit weitere ausführbare Software zu installieren. Sie entschließen sich dazu, eines der zahlreichen quelloffenen Content-Management-Systeme (CMS) wie WordPress[14], Drupal[15] oder Joomla![16] zu nutzen. Da sie den Aufwand scheuen, die Daten einiger Hundert Nobelpreisträger in die Datenbank einzutragen und sie diese nach Ablauf des Projekts auch nicht auf dem aktuellen Stand halten möchten, entscheiden sie sich dazu, SemwidgJS zu nutzen und die Daten von einem öffentlichen SPARQL-Endpoint abzurufen, welcher die gewünschten Daten bereitstellt. Dazu ersetzen sie den standardmäßig installierten WYSIWYG-Editor des CMS durch SemwidgED. Auf der Website des Semwidg-Projekts haben sie sich ein Tutorial-Video angeschaut, in dem ihnen der Umgang mit SemwidgED erläutert wurde.

[14] https://wordpress.org
[15] https://www.drupal.org
[16] https://www.joomla.org

Den Großteil des Inhalts der Website geben sie auf die selbe Weise ein, wie sie es auch mit dem vorinstallieren Editor getan hätten. Für die Liste der Nobelpreisträger nutzen sie jedoch die erweiterten Funktionen des Editors:

- Zuerst konfigurieren sie die Seite so, dass SemwidgJS DBpedia als Datenquelle nutzt.
- Da sie noch nicht wissen, welche Informationen genau sie anzeigen wollen, entscheiden sie sich dazu, stellvertretend einen Preisträger auszuwählen und zu explorieren, welche Eigenschaften über ihn in der Datenbank enthalten sind. Sie entscheiden sich für Albert Einstein als Stellvertreter, da sie davon ausgehen, dass über ihn besonders viele Informationen gespeichert sind. Der Editor unterstützt sie bei der Suche nach dem entsprechenden Resource-URI.
- Als erste Übung möchten sie den Namen Albert Einsteins ausgeben. Aus einer Liste, die ihnen alle verfügbaren Widget-Klassen anzeigt, wählen sie das Text-Widget aus. In einem Konfigurationsdialog für das Widget besteht die Möglichkeit, unterstützt durch den Editor, eine SemwidgQL-Abfrage zu formulieren. Der Editor zeigt ihnen alle vorhandenen Eigenschaften an, die mit Albert Einstein verknüpft sind, sodass sie den gewünschten Wert schnell finden. Dabei bekommen sie auch einen Überblick die anderen Werte, die zu ihm gespeichert sind.
- Da sie gesehen haben, dass auch ein kurzer Beschreibungstext zu Albert Einstein gespeichert ist, möchten sie diesen als nächstes anzeigen. Sie wählen wieder das Text-Widget aus und formulieren anschließend die entsprechende SemwidgQL-Abfrage. Nach dem Speichern fällt ihnen auf, dass der Text in der falschen Sprache angezeigt wird. Sie öffnen den Konfigurationsdialog für das Widget erneut und fügen einen Filterausdruck für die Sprache hinzu, sodass nur noch der deutsche Beschreibungstext angezeigt wird.
- Als nächstes möchten sie den Geburtsort von Albert Einstein auf einer Karte markieren und diese anzeigen. Sie wählen dazu das Karten-Widget aus. Anschließend müssen sie eine Pfadabfrage mit einer größeren Pfadlänge als bei den bisher genutzten Widgets erstellen. Während die bisherigen Werte direkt mit der ursprünglichen Resource verlinkt waren, lassen sich die benötigten Koordinaten nur abfragen, wenn zunächst die Resource des Geburtsorts aufgerufen wird. Da ihnen die Standardgröße der Karte nicht gefällt, passen sie diese mittels der entsprechenden CSS-Regeln an.
- Die bisherige Widget-Komposition soll nun in ein Template umgewandelt werden, das für die Anzeige der Informationen zu beliebigen Nobelpreisträgern genutzt werden kann. Sie markieren die Widgets, die Teil des Templates werden sollen, und ersetzen die spezifische Resource-URI von Albert Einstein durch einen generische Parameter.
- Sie erstellen eine Instanz des zuvor erzeugten Templates. Die verfügbaren Templates werden ihnen, wie schon die verfügbaren Widgets, in einer Liste angezeigt. In einem Konfigurationsdialog können sie die gewünschten Werte für die Template-Parameter eingeben.
- Abschließend fügen sie ein Drop-Down-Widget in die Seite ein, das alle Resources der RDF-Klasse *Physik-Nobelpreisträger* enthält. Diese Widget-Klasse kann mit einem anderen SemwidgJS-Element verknüpft werden und übergibt den ausgewählten Wert an das verknüpfte Widget. Der Editor schlägt ihnen das Template und den dafür erzeugten Parameter als Ziel vor.

Die Speicherung der Seite geschieht regulär über das genutzte CMS. Dieses erhält vom Editor den HTML-Quelltext, welcher in der Datenbank des CMS gespeichert wird.

4.2.3 Anforderungen

In diesem Abschnitt werden die wichtigsten Anforderungen an SemwidgED beschrieben. Diese ergeben sich zum Teil aus den Anforderungen an SemwidgJS (vgl. Abschnitt 3.2.2) und dessen Funktionen, dem aktuellen Stand der Forschung hinsichtlich der Editoren für Linked-Data-Präsentationen (vgl. Abschnitt 4.1.1) und der Editoren zur Abfrageformulierung (vgl. Abschnitt 4.1.2) sowie den zuvor erfolgten Definition der potenziellen Nutzer und des Anwendungsszenarios.

Grundlegende Anforderungen: Hauptaufgabe von SemwidgED soll es sein, SemwidgJS-Widgets in HTML-Dokumente einbetten zu können. Nutzer müssen also in der Lage sein, die vier Elementtypen von SemwidgJS *Konfigurationselemente* (vgl. Abschnitt 3.3.2), *Widgets* (vgl. Abschnitt 3.3.3) sowie *Templates* und *Template-Instanzen* (vgl. Abschnitt 3.3.4) erzeugen, bearbeiten und entfernen zu können. Bereits erstellte Webseiten sollen erneut eingelesen werden können, um sie bei Bedarf anpassen zu können. Im Gegenzug muss es möglich sein, den Quelltext der erstellten Webseite abspeichern zu können. Regulärer HTML-Text muss sich mit Linked Data mischen lassen.

Unterstützung bei der Einbettung von Widgets: Die Untersuchung der Editoren für Link-Data-Präsentationen (vgl. Abschnitt 4.1.1) hat gezeigt, dass sich ein guter Editor durch drei Eigenschaften besonders auszeichnet. Er sollte sich ohne fortgeschrittene Programmierkenntnisse nutzen lassen, dem Nutzer viele Freiheiten bei der Gestaltung geben und ihn gleichzeitig bei der Erstellung der Webseite unterstützen. Da sich die Freiheiten bei der Gestaltung und die Unterstützung bei der Erstellung der Webseite in allen ausführlich untersuchten Editoren gegenseitig ausschlossen, sollte der Editor die Nutzer zwar möglichst stark unterstützen können, vornehmlich in einem WYSIWYG-Modus, ihnen diese Unterstützung jedoch nicht aufdrängen. Experten-Nutzer sollten immer die Möglichkeit haben, den Quelltext direkt bearbeiten, danach aber auch wieder in den unterstützenden Modus zurückwechseln zu können.

Unterstützung bei der Formulierung von Abfragen: Die Untersuchung der Editoren zur Abfrageformulierung (vgl. Abschnitt 4.1.2) hat ähnliche Ergebnisse geliefert, wie die Untersuchung der Editoren für Link-Data-Präsentationen. Je stärker die Systeme den Nutzer unterstützt haben, umso stärker haben sie ihn in seiner Freiheit bei der Formulierung der Abfragen eingeschränkt. Da SemwidgJS zwei verschiedene Abfragesprachen unterstützt, nämlich SPARQL und SemwidgQL, sollten sich auch beide Abfragesprachen nutzen lassen. Das Hauptaugenmerk liegt in dieser Arbeit jedoch auf der konzeptionellen Gestaltung der unterstützenden Eingabe von SemwidgQL-Abfragen. Zur Eingabe von SPARQL-Abfragen kann auf bereits bestehende Lösung zurückgegriffen werden. Von den drei zuvor untersuchten Editor-Kategorien bietet sich für die Experten die textbasierte Eingabe an, da sie die vollständige Kontrolle über die Abfrageformulierung bietet. Für alle anderen Nutzer bietet sich eine formularbasierte Eingabe an, welche eine stärkere Unterstützung ermöglicht, als die textbasierte Eingabe. Die graphbasierte Eingabe, welche in Rahmen der Untersuchung als die ungeeignetste Eingabevariante erwies, lässt sich für SemwidgQL ohnehin nicht umsetzen, da SemwidgQL kein graphbasierter Aufbau zugrunde liegt, sondern ein hierarchischer.

Zusätzliche Funktionen: Da es eines der Ziele war, dass SemwidgJS clientseitig nutzbar ist und bis auf den SPARQL-Endpoint keine weitere Serversoftware benötigt, sollte dies auch für SemwidgED gelten. Sollte der Nutzer bereits weitere Serversoftware wie ein Content-Management-System einsetzen, sollte sich SemwidgED in dieses integrieren lassen.

Die hier beschrieben Anforderungen stellen die Mindestanforderungen dar, die an SemwidgED gestellt werden. Letztendlich wurden noch weitere Funktionen in den Editor eingefügt. Diese werden in Abschnitt 4.3 genauer beschrieben werden.

4.3 Implementierung

Auf Grundlage der zuvor gemachten konzeptionellen Überlegungen (vgl. Abschnitt 4.2), wird in diesem Abschnitt die Implementierung von SemwidgED beschrieben. Zu Beginn werden einige allgemeine Informationen bezüglich der Implementierung vorgestellt. Diese beziehen sich unter anderem auf die Software-seitige Basis des Editors und dessen Hauptsteuerungskomponente. Anschließend werden die implementierten Funktionen von SemwidgED beschrieben. Die Reihenfolge der Beschreibungen richtet sich dabei nach der Reihenfolge, in der sie ein Nutzer, wie zum Beispiel im Rahmen des Anwendungsszenarios (vgl. Abschnitt 4.2.2) skizziert, vermutlich nutzen wird.

4.3.1 Allgemeine Informationen

Einleitend werden einige Informationen bezüglich der Basis, auf welcher die Implementierung von SemwidgED aufbaut, vorgestellt und die Hauptsteuerungskomponente des Editors, die *SemwidgED Toolbox*, betrachtet. Anschließend werden die Funktionen zur Vorschlagsgenerierung und Autovervollständigung, die den Nutzer an verschiedenen Stellen des Editors bei der Erstellung der Webseite unterstützen, herausgearbeitet.

Basis des Editors

SemwidgED wurde als Plug-In für den freien WYSIWYG-HTML-Editor CKEditor[17] entwickelt. CKEditor und das SemwidgED-Plug-In wurden in JavaScript programmiert und sind daher in der Lage mit SemwidgJS zu interagieren. Im weiteren Verlauf wird der CKEditor in Verbindung mit dem SemwigdED-Plugin vereinfacht als SemwidgED bezeichnet. Viele Content-Management-Systeme nutzen den CKEditor standardmäßig (z. B. Drupal[18], Bolt[19]) oder können so konfiguriert werden, dass sie ihn nutzen (z. B. Joomla![20]). Neben dem verbreiteten Einsatz des Editors und der Möglichkeit Plug-Ins einzubinden, zeichnet ihn auch die Unterstützung von Widgets aus. Im CKEditor-Kontext sind Widgets komplexe Elemente, die mehrere HTML-Elemente zusammenfassen können und vom Editor als zusammengehörig betrachtet werden. Diese Elemente können nicht einfach wie normale HTML-Elemente editiert werden, sondern werden über ein Dialogfenster konfiguriert. SemwidgED betrachtet alle SemwidgJS-Elemente als CKEditor-Widgets. Da SemwidgJS-Widgets, Templates und Template-Instanzen selbst verschiedene HTML-Elemente enthalten und gruppieren können, wird auf diese Weise verhindert, dass sie von einem Nutzer versehentlich auf eine Weise bearbeitet werden, welche ungültige SemwidgJS-Elemente erzeugt. Zudem können auf diese Weise auch eigentlich nicht sichtbare SemwidgJS-Elemente wie Templates und Konfigurationselemente innerhalb der WYSIWYG-Ansicht sichtbar und somit für den Nutzer leicht konfigurierbar gemacht werden. Alle anderen Elemente werden, genau wie innerhalb einer regulären Webseite, verarbeitet und dargestellt.

SemwidgED-Toolbar

Die SemwidgED-Toolbar ist eines der Hauptelement von SemwidgED (vgl. Abbildung 4.4, links). Sie ist unterteilt in die drei Abschnitte *SemwidgJS Toolbox*, *Widget Management* und

[17] https://ckeditor.com/ckeditor-4/
[18] https://www.drupal.org
[19] https://boltcms.io
[20] https://www.joomla.org

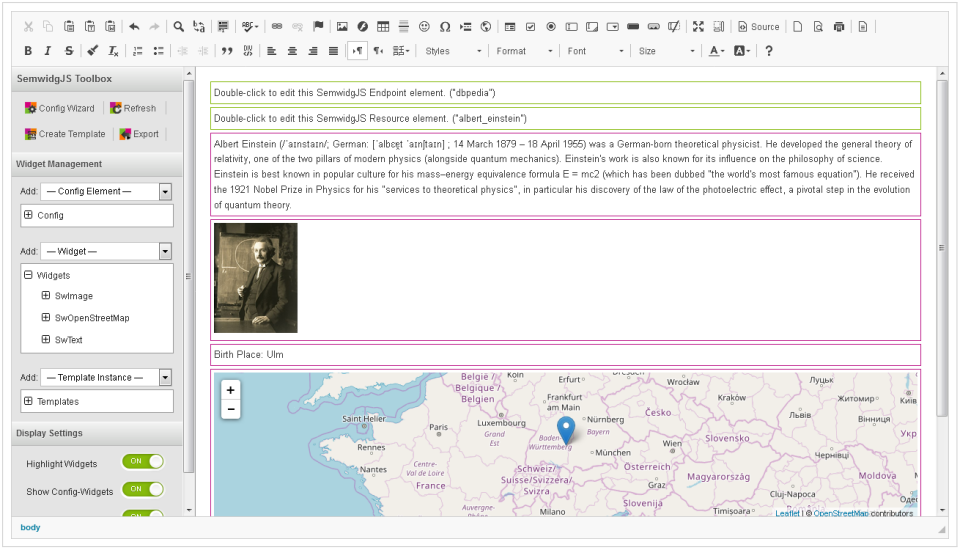


Abbildung 4.4: SemwidgED ist in drei Bereiche aufgeteilt. Am oberen Rand befindet sich die CKEditor-Toolbar, die die regulären HTML-Editor-Funktionen zur Verfügung stellt. Am linken Rand befindet sich die SemwidgED-Toolbar. Über sie lassen sich SemwidgJS-Elemente in die Webseite einbinden. Die restliche Fläche zeigt den Inhalt der Webseite an, welche mit SemwidgED erstellt wurde.

Display Settings. Über die Toolbox lassen sich der Konfigurationsassistent starten (vgl. Abschnitt 4.3.2), Templates aus bereits erstellten Inhalten generieren (vgl. Abschnitt 4.3.5) und die erstellte Webseite exportieren (vgl. Abschnitt 4.3.6).

Mit Hilfe des Widget-Management-Bereichs lassen sich SemwidgJS-Widgets (vgl. Abschnitt 4.3.3), Konfigurationselemente und Template-Instanzen (vgl. Abschnitt 4.3.5) einbinden und verwalten. Alle verfügbaren SemwidgJS-Elemente lassen sich über ein entsprechendes Drop-Down-Menü auswählen. Die verfügbaren Elemente werden zur Laufzeit aus dem internen SemwidgJS-Modell und dem erstellten HTML-Dokument ausgelesen, sodass auch fremde Widgets-Klassen und selbst generierte Templates angezeigt werden können. Bereits instanziierte Elemente lassen sich über drei entsprechende Listen suchen, konfigurieren und löschen. Diese Funktionen lassen sich auch direkt über die Live-Ansicht aufrufen, jedoch muss dazu erst das entsprechende Element innerhalb dieser Ansicht lokalisiert werden, was bei größeren HTML-Dokumenten mühsam sein kann.

Der Display-Settings-Bereich ermöglicht die Anpassung der Darstellung der SemwidgJS-Elemente innerhalb der Live-Ansicht. SemwidgED zeigt standardmäßig auch Konfigurationselemente und Template-Definitionen an, die in einer Webseite eigentlich nicht sichtbar sind. Zudem werden alle Elemente mit einem farbigen Rahmen markiert, um sie im Vergleich zu regulären HTML-Elementen hervorzuheben. Dieses Verhalten lässt sich über entsprechende Konfigurationselemente anpassen beziehungsweise deaktivieren.

Vorschlagsgenerierung und Autovervollständigung für Eingabefelder

An vielen Stellen des Editors wird von einem Nutzer gefordert, dass er eine textuelle Eingabe vornimmt. Dabei unterscheidet sich der Freiheitsgrad der erforderlichen Eingabewerte und damit die Möglichkeit zur Generierung von Vorschlägen teilweise erheblich. Für SemwidgED

wurden für vier verschiedene Freiheitsgrade jeweilige Funktionen zur Vorschlagsgenerierung implementiert.

Vordefinierte Werte: SemwidgJS gibt für einige Elemente, beziehungsweise deren Eigenschaften eine eingeschränkte Menge von Konfigurationswerten vor. Andere Werte als diese sind nicht erlaubt. Dies ist die einfachste Variante bezüglich der Anzeige von Vorschlägen, da diese Werte bekannt sind und daher fest einprogrammiert werden konnten. Unter anderem wird diese Variante bei der Konfiguration von Endpoint-Elementen zum Einsatz. SemwidgJS kann die Daten entweder über das GET- oder das POST-Protokoll vom SPARQL-Endpoint abrufen. Entsprechend können diese beiden Werte vorgeschlagen werden. Theoretisch hätte in diesem Fall auch ein Drop-Down-Menü als Eingabeelement genutzt werden können. Sollte SemwidgJS jedoch einmal erweitert werden, sodass noch weitere Konfigurationswerte möglich werden, hätte auch SemwidgED zwingend angepasst werden müssen. Ein Textfeld, welches eine freie Eingabe ermöglicht, bleibt auch in diesem Fall uneingeschränkt nutzbar.

Zur Laufzeit definierte Werte: Viele SemwidgJS-Elemente werden untereinander anhand ihrer IDs verknüpft. Diese IDs werden vom Nutzer definiert und während der Laufzeit von SemwidgED ausgelesen. Bei Bedarf können sie als Eingabewerte vorgeschlagen werden. Unter anderem wird diese Variante der Vorschlagsgenerierung bei der Verknüpfung von Resource-Elementen und dem Endpoint-Element, welches den SPARQL-Endpoint repräsentiert genutzt, in dem die Daten zu dieser Resource enthalten sind. Einige Widgets besitzen zudem Eigenschaften vom Typ *Selector* (vgl. Abschnitt 3.3.3: *Eigenschaftentypen*). Selector-Eigenschaften definieren andere SemwidgJS-Elemente oder deren Eigenschaften als Ziel für Widget-abhängige Funktionen. In der Regel werden sie genutzt, um einen ausgewählten Werte in ein anderes Element zu schreiben (z. B. kann der URI eines Resource-Elements adressiert und überschrieben werden: `someResource.value`). Da alle SemwidgJS-Elemente dem Editor bekannt sind und sich alle verfügbaren Eigenschaften auslesen lassen, können auch hier entsprechende Vorschläge angezeigt werden.

Dynamisch erweiterbare Werte: Bei der Formulierung von Abfragen, schränken bisher gemachte Eingaben die Möglichkeiten für darauffolgende Werte ein. Die Vorschlagsgenerierung muss sich daher dynamisch an die bisherigen Eingaben anpassen.[21] Für die Unterstützung bei der Formulierung von SemwidgQL-Abfragen wurden zwei verschiedene Eingabemodi implementiert. Für Experten-Nutzer wurde ein textbasierter Eingabemodus und für Anfänger ein formularbasierter Eingabemodus entwickelt. Die Vorschlagsgenerierung ist in der Lage, für jeden Abschnitt der SemwidgQL-Abfrage, entsprechende Vorschläge zu erzeugen. Für das Auffinden verlinkter Eigenschaften nutzen beide Ansätze das selbe Vorgehen. Hierbei wird der Teil der Abfrage, der momentan bearbeitet wird durch eine Variable ersetzt. Der Editor fragt daraufhin die URIs der möglichen Variablenbelegungen beziehungsweise Eigenschaften zuzüglich deren menschenlesbaren Bezeichnungen (`rdfs:label`) ab. Weiterhin wird für jede Eigenschaft ein Beispielwert abgerufen, der dann zusammen mit dieser, der Bezeichnung und der Richtung der Verlinkung (eingehen oder ausgehen) in der Vorschlagsliste angezeigt wird. Zu Beginn kann das System immer

[21] SemwidgJS unterstützt sowohl SPARQL- wie auch SemwidgQL-Abfragen. Da bereits eine Vielzahl von SPARQL-Editoren existiert (vgl. Abschnitt 4.1.2) und der Fokus dieser Arbeit auf SemwidgQL liegt, wurde darauf verzichtet eine fortgeschrittene Unterstützung für die Formulierung von SPARQL-Abfragen in SemwidgED zu implementieren. Der Nutzer kann bei Bedarf eine SPARQL-Abfrage innerhalb eines regulären Textfeldes formulieren.

den Wildcard-Operator und, falls vorhanden, die bereits konfigurierten Resources vorschlagen. Die beiden Modi werden in Abschnitt 4.3.4 detailliert beschrieben.

Freie Werte: Für das Auffinden von Resource-URIs wurde eine Freitextsuche implementiert, die auf den Bezeichnungen (RDFS:LABEL) der Resources arbeitet. Im Gegensatz zu den bisher beschriebenen Varianten, können dem Nutzer hier nicht von Beginn an Vorschläge unterbreitet werden. Die möglichen Werte sind zwar theoretisch bekannt, da eine Liste aller im SPARQL-Endpoint vorhandenen Resources abgerufen werden könnte. Jedoch ist dies in vielen Fällen aufgrund der großen Menge an Daten nicht praktikabel. Nutzer müssen daher einen Suchbegriff definieren und erhalten anschließend, nach Relevanz geordnet, eine Liste der in Betracht kommenden Resources. Die Abfrage von Resource-URIs wird in Abschnitt 4.3.2 detailliert beschrieben.

In allen Fällen hat der Nutzer stets die Möglichkeit, auch Werte einzugeben, die nicht vorgeschlagen werden. Dies kann zum Beispiel dann notwendig sein, wenn SemwidgJS neue Funktionen hinzugefügt wurden, die noch nicht von SemwidgED unterstützt werden. Oder weil die Verbindung zum SPARQL-Endpoint momentan gestört ist und so keine Vorschläge generiert werden können. Oder weil Nutzer bereits Eingaben vorwegnehmen möchten, die erst im weiteren Verlauf der Arbeit mit dem Editor zu validen Ergebnissen führen. Eine zu rigide Steuerung der Eingabe würde die Nutzer in diesen Fällen stärker behindern als ihnen nutzen.

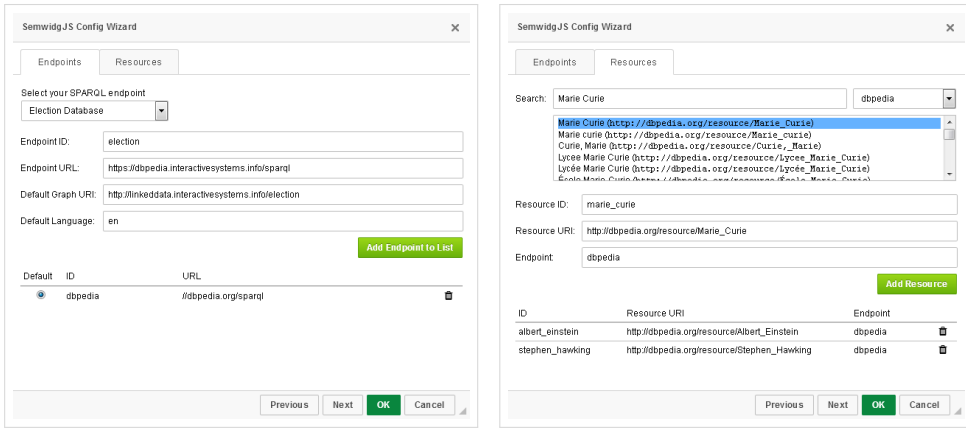
4.3.2 Konfigurationsassistenz für SPARQL-Endpoints und Resources

Eine der ersten Tätigkeiten, die ein Nutzer durchführen muss, um Daten mittels SemwidgJS in seine Webseite einzubinden, ist es, eine Verbindung zu einem SPARQL-Endpoint herzustellen und gegebenenfalls ein oder mehrere Resource-Elemente zu konfigurieren, die die Entitäten repräsentieren, zu denen Informationen abgerufen werden sollen. Zur Unterstützung der Nutzer bei der Konfiguration von Resource-Elementen und des Zugriffs auf einen oder mehrere SPARQL-Endpoints wurde ein Assistent implementiert. Die Nutzung des Assistenten ist optional. SPARQL-Endpoint-Konfigurationen und Resource-Elemente können, wie alle anderen Semwidg-Elemente auch, direkt über die SemwidgED-Toolbar hinzugefügt werden.

Der SPARQL-Endpoint-Konfigurator (vgl. Abbildung 4.5a) enthält eine vordefinierte Liste mit SPARQL-Endpoint-Konfigurationen von relevanten öffentlich zugänglichen SPARQL-Endpoints, die der Webseite hinzugefügt werden können. Ebenso können eigene Konfigurationen vom Nutzer definiert werden. Der Konfigurator zeigt eine Liste aller hinzugefügten SPARQL-Endpoint-Konfigurationen an und ermöglicht das nachträgliche Editieren und Löschen dieser und das Festlegen eines Standard-SPARQL-Endpoints. Abfragen die nicht spezifisch einem anderen Endpoint zugewiesen sind, werden an diesen Standard-Endpoint gestellt.[22]

Der Resource-Konfigurator (vgl. Abbildung 4.5b) enthält eine integrierte Suchfunktion, die es dem Nutzer erlaubt, die passende Resource-URI anhand des verknüpften Labels zu finden. Die Suche wird automatisch gestartet, wenn nach der letzten Eingabe des Nutzers die Zeit von einer Sekunde verstrichen ist. Diese Zeitspanne hat sich bereits in anderen Projekten (z. B. Relfinder, vgl. Heim et al., 2009) als sinnvoll erwiesen. Die Verzögerung ist so gering, dass sie dem Nutzer nicht störend auffällt. Außerdem werden keine oder nur vereinzelte unnötige Abfragen an den SPARQL-Endpoint gestellt, die die relevanten Abfragebearbeitungen gegebenenfalls

[22] Werden zum Beispiel Wildcard-Abfragen in SemwidgQL formuliert, werden in diesen Fällen auch keine Resource-Elemente genutzt, welche mit einem SPARQL-Endpoint verknüpft sein könnten. Entweder muss diese Verknüpfung daher zusätzlich für die Abfrage beziehungsweise das entsprechende Widget hinzugefügt werden oder SemwidgJS greift auf den festgelegten Standard-SPARQL-Endpoint zurück.



(a) SPARQL-Endpoint-Konfigurator **(b)** Resource-Konfigurator

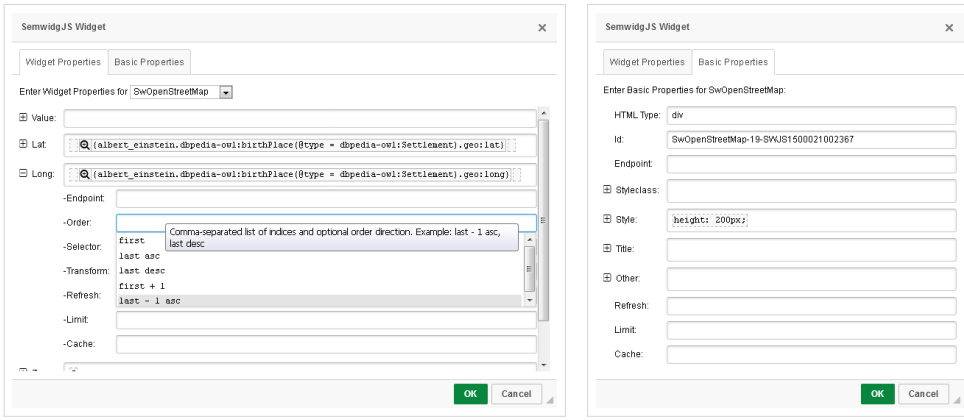
Abbildung 4.5: Der Konfigurationsassistent besitzt zwei Bereiche. Im ersten können Nutzer SPARQL-Endpoint-Konfigurationen zur Webseite hinzufügen. Im zweiten können sie Resource-Elemente hinzufügen. Dabei können sie eine Suchfunktion nutzen.

noch weiter hinauszögern. Je nach Auslastung des SPARQL-Endpoints und der Anzahl der unnötigerweise gestellten Abfragen, zögert die sie Abfragebearbeitung sogar erheblich über die Zeitspanne von einer Sekunde hinaus. Bezüglich der Suchfunktion und der Sortierung der angezeigten Ergebnisse wurden verschiedene Abfragen und Relevanzmetriken evaluiert:

- Die trivialste der untersuchten Varianten ist es, alle Resources abzufragen, die den Suchtext in ihrem verlinkten Label enthalten und die Ergebnisliste alphabetisch sortiert anzuzeigen. Da die Ergebnismengen allerdings sehr groß sein können, müsste der Nutzer hier in vielen Fällen erst eine lange Liste nach der gewünschten Resource durchsuchen. Eine Sortierung der Ergebnisse anhand der Relevanz ist daher erstrebenswert.

- Eine Möglichkeit zur Bestimmung der Relevanz eines Ergebnisses kann über die Bestimmung der Relevanz der entsprechenden Resource innerhalb der gesamten Datenbasis erfolgen. Diese Relevanz lässt sich beispielsweise über die Valenz der Resource bestimmen. Die Anzahl der eingehenden (*Eingangsgrad*) oder ausgehenden (*Ausgangsgrad*) Verlinkungen mit anderen Resources kann ein Indiz bezüglich der Wichtigkeit dieser Resource sein. Eine Resource, die mit vielen anderen Resources verlinkt ist, ist vermutlich relevanter als eine Resource mit nur wenigen Verlinkungen. Allerdings kann die Sortierung anhand dieser Metrik dazu führen, dass vergleichsweise weniger relevante Resources nur noch schwer gefunden werden können. Dies betrifft vor allem Resources mit kurzen Labeln, die Teil des allgemeinen – gegebenenfalls fremdsprachigen – Wortschatzes sind.[23] Da die Sortierung keinen Hinweis darauf gibt, an welcher Stelle sich die gesuchte Resource befindet, müsste im schlimmsten Fall die gesamte Liste durchsucht werden.

[23] Der Film *Heat* mit Al Pacino und Robert De Niro in den Hauptrollen, wird beispielsweise bei einer Abfrage des DBpedia-SPARQL-Endpoints und der Sortierung nach eingehenden Kanten erst an 26. Stelle gelistet. Vergleichbares gilt für den Song *Jump* der Band Van Halen, der nach dieser Sortierung erst an 20. Stelle gelistet wird.



(a) Spezifische Eigenschaften **(b)** Allgemeine Eigenschaften

Abbildung 4.6: Dialog zur Konfiguration von SemwidgJS-Widgets. Innerhalb des ersten Tabs können für eine Widget-Klasse spezifische Eigenschaften definiert werden. Im zweiten Tab lassen sich die allgemeinen Eigenschaften definieren, die jedes Widget bereit stellt.

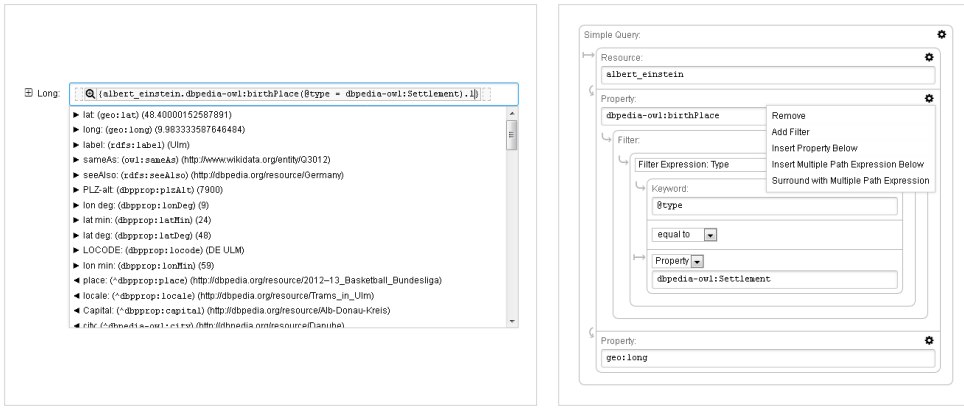
- Alternativ kann die Relevanz eines Ergebnisses für einen Nutzer anhand der Ähnlichkeit des Labels zu dem eingegebenen Suchtext bestimmt werden. Je ähnlicher das gefundene Label dem eingegebenen Suchtext entspricht, umso relevanter dürfte es für den Nutzer sein. Hier bieten sich verschiedene Ähnlichkeitsmaße an. Für SemwidgED wurde die Levenshtein-Distanz (vgl. Levenshtein, 1966) gewählt. Diese beschreibt die minimale Anzahl von Einfüge-, Lösch- und Ersetzungsoperationen, die benötigt werden um eine Zeichenkette in eine andere umzuwandeln. Wahlweise hätte auch die Jaro-Winkler-Distanz (vgl. Winkler, 1990) oder der Wagner-Fischer-Algorithmus gewählt werden können, welche vergleichbare Werte berechnen.

Letztendlich wurde die Sortierung der Ergebnisse anhand der Levenshtein-Distanz gewählt. Je ähnlicher ein Ergebnis dem Suchtext ist, umso weiter oben erscheint es in der Ergebnisliste. Unabhängig davon, ob es stark mit anderen Resources verknüpft ist oder nicht. Im Gegensatz zur Valenz-basierten Sortierung, welche anhand einer für den Nutzer nicht sichtbaren Meta-Eigenschaft erfolgt, bietet die ähnlichkeitsbasierte Sortierung dem Nutzer einen Hinweis darauf, warum ein Ergebnis an einer bestimmten Stelle steht. Dies erleichtert das Explorieren der Ergebnisliste.

Bereits hinzugefügte Resource-Elemente werden in einer Liste angezeigt. Diese können, wie schon die SPARQL-Endpoint-Konfigurationen, gelöscht und editiert werden.

4.3.3 Widget-Einbindung

Über die SemwidgED-Toolbar können SemwidgJS-Widgets in die Webseite eingebunden werden. Diese werden an der Stelle der aktuellen Cursor-Position eingefügt. Die Konfiguration der Widgets erfolgt über ein Dialogfenster mit zwei Tabs (vgl. Abbildung 4.6). Innerhalb des ersten Tabs können die für eine bestimmten Widget-Klasse spezifischen Eigenschaften definiert werden. Gleiches gilt für die »Value«-Eigenschaft. Die Klasse des Widgets kann in diesem Tab zu jeder Zeit gewechselt werden. So können beispielsweise Text-Widgets, die nur ein einzelnes Ergebnis ausgeben, in List-Widgets umgewandelt werden, die mehrere Ergebnisse darstellen



(a) Textbasierte Eingabe **(b)** Formularbasierte Eingabe

Abbildung 4.7: Unterstützung bei der Formulierung von SemwidgQL-Abfragen. Die textBasierte Eingabe richtet sich an fortgeschrittene Nutzer, während die formularbasierte Eingabe auch Anfängern die Formulierung von komplexeren SemwidgQL-Abfragen ermöglicht.

können. Innerhalb des zweiten Tabs können die allgemeinen Eigenschaften definiert werden, die von allen Widgets unterstützt werden. In den meisten Fällen wird der Nutzer nur Eigenschaften des ersten Tabs definieren wollen.

SemwidgED liest die Typen aller Eigenschaften eines Widgets (vgl. Abschnitt 3.3.3: *Eigenschaftentypen*) zur Laufzeit aus und erzeugt basierend auf diesen Informationen die passenden Eingabefelder mit Vorschlagsgenerierung. Für Eigenschaften, für die eine Menge mit vordefinierten möglichen Werten existiert, wird eine Liste mit diesen Werten erzeugt, die dem Nutzer bei der Bearbeitung dieser Eigenschaft angezeigt wird. Eingabefelder für Eigenschaften, die eine SemwidgQL- oder SPARQL-Abfrage als Wert erwarten, verfügen über eine dynamische Vorschlagsgenerierung, die den Nutzer bei der Erstellung von SemwidgQL-Abfragen unterstützt. Diese Funktion wird in Abschnitt 4.3.4 detailliert beschrieben.

Zudem werden für alle Eigenschaften vom Typ *Query* die Eingabefelder der entsprechenden Untereigenschaften erzeugt. Über diese können beispielsweise ein bestimmter SPARQL-Endpoint angegeben werden, der für diese spezielle Eigenschaft genutzt werden soll, oder die Sortierung der Ergebnisse festgelegt werden.

4.3.4 Unterstützung bei der Formulierung von SemwidgQL-Abfragen

Für SemwidgED wurden zwei Varianten für die Unterstützung bei der Formulierung von SemwidgQL-Abfragen implementiert. Zum einen eine textbasierte Eingabe für fortgeschrittene Anwender, zum anderen eine formularbasierte Eingabe für Anfänger. Beide Varianten werden nachfolgend beschrieben.

Textbasierte Eingabe

Die textbasierte Eingabe (vgl. Abbildung 4.7a) ermöglicht die Eingabe von statischem Text sowie von SemwidgQL- und SPARQL-Abfragen. Das Eingabefeld erkennt SemwidgQL- und SPARQL-Abfragen automatisch anhand der einfachen beziehungsweise doppelten geschweiften Klammern, von denen sie umschlossen werden und hebt sie visuell hervor. Sobald eine

SemwidgQL-Abfrage erkannt wurde, wird vor dieser zusätzlich ein Button angezeigt, der ein zusätzliches Dialogfenster zur formularbasierten Eingabe öffnet.

Das Eingabefeld unterstützt fortgeschrittene Nutzer bei der Formulierung von SemwidgQL-Abfragen auf eine Weise, die den meisten dieser Nutzer aus der Anwendung von integrierten Entwicklungsumgebungen (IDE) bekannt sein dürften. Während des Tippens erscheint unterhalb der Eingabe eine Liste mit Vorschlägen, wie die momentane Abfrage vervollständigt werden könnte. Über die Pfeiltasten kann ein Vorschlag selektiert und mittels der Eingabetaste dem Abfragetext hinzugefügt werden.

Hierbei werden alle Eigenschaften abgefragt, die beim Einsetzen in die aktuelle Bearbeitungsposition eine Abfrage mit einem nicht-leeren Ergebnis erzeugen. Dabei werden alle vollständigen Teile der Abfrage vor und gegebenenfalls hinter der Bearbeitungsposition (begrenzt durch Punkte, Klammern etc.) miteinbezogen. Der aktuell bearbeitete Teil wird durch eine Variable ersetzt. Dem Nutzer wird anschließend die Bezeichnung der Eigenschaft und ein Beispielwert angezeigt. Zudem wird dargestellt, ob es sich um eine reguläre oder inverse Eigenschaft handelt. Wird an der entsprechenden Stelle der Abfrage statt einer Eigenschafte eine Resource erwartet, werden die IDs der der Webseite bereits zugefügten Resource-Elemente sowie die Wildcard-Resource angezeigt.

Formularbasierte Eingabe

Die formularbasierte Eingabe (vgl. Abbildung 4.7b) ermöglicht auch weniger fortgeschrittenen Nutzern dir Formulierung von komplexeren SemwidgQL-Abfragen. Dadurch dass das Formular die Struktur der Abfrage vorgibt, kann der Nutzer bei der formularbasierte Eingabe der Abfrage deutlich stärker als bei textbasierten Eingabe geführt werden, wodurch Fehler vermieden werden und ein exploratives Vorgehen unterstützt wird. Bereits formulierte Abfragen, die über die textbasierte Eingabe erzeugt wurden, werden in das Formular übernommen, sofern sie keine syntaktischen Fehler aufweisen. Die formularbasierte Eingabe unterstützt sämtliche SemwidgQL-Elemente, sodass jede SemwidgQL-Abfrage darüber darstellbar ist.

Die Abfrageelemente mit den dazugehörigen Formularfeldern werden in einer hierarchischen Struktur geschachtelt dargestellt. Nutzer können zulässige neue Felder über ein Drop-Down-Menü auswählen und hinzufügen. Die Vorschlagsgenerierung funktioniert auf die gleiche Weise wie bei der textbasierten Eingabe. Das momentan bearbeitete Element der Abfrage wird durch eine Variable ersetzt und es werden alle Eigenschaften abgefragt, die beim Einsetzen in die aktuelle Bearbeitungsposition eine Abfrage mit einem nicht-leeren Ergebnis erzeugen. Wird an der entsprechenden Stelle der Abfrage statt einer Eigenschafte eine Resource erwartet, werden die IDs der bereits zugefügten Resource-Elemente der Webseite und die Wildcard-Resource angezeigt.

Für hinzugefügte Filter kann der Nutzer die Art des Filters angeben, wodurch der Editor ihm zusätzliche Unterstützung anbieten kann. Für Filter auf Basis von *Filter-Schlüsselwörtern* (vgl. Abschnitt 2.3.3) kann der erwartete Datentyp (Zeichenkette, Zahl, boolescher Ausdruck, Zeiträume etc.) überprüft werden und es werden für einige Datentypen vordefinierte Beispielwerte angezeigt, die den Nutzer die Formulierung erleichtern soll (z. B. für Zeiträume: »60 seconds«).

Das Dialogfeld, welches die formularbasierte Eingabe enthält, enthält zusätzlich zwei Tabs, die die aus dem Formular generierte SemwidgQL-Abfrage und die übersetzte SPARQL-Abfrage darstellen sowie ein Vorschaufenster, welches die Ergebnisse dieser Abfrage auflistet.

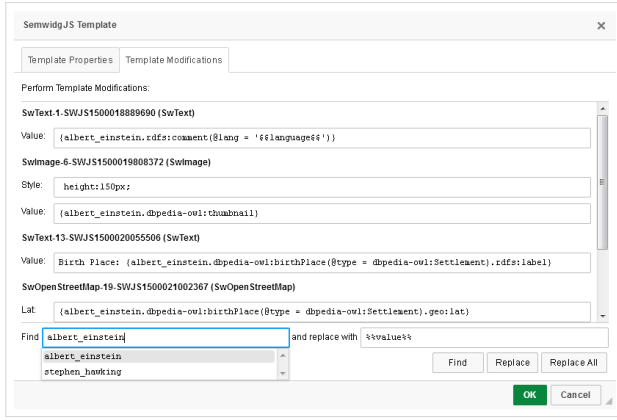


Abbildung 4.8: Dialog zur Template-Generierung. Der Dialog verfügt über eine Suchen-und-Ersetzen-Funktion, die den Austausch beliebiger Zeichenketten innerhalb der Eigenschaften der SemwidgJS-Elemente ermöglicht.

4.3.5 Template-Generierung und -Instanziierung

Bereits erstellte Inhalte – statische Inhalte, SemwidgJS-Widgets oder eine Kombination aus beidem – kann vom Nutzer in Templates umgewandelt werden. Dabei können Teile der enthaltenen Inhalte durch Platzhalter ersetzt werden. Diese Templates können anschließend mit verschiedenen Parameterwerten für diese Platzhalter instanziiert werden.

Template-Generierung

Um ein Template zu generieren, muss der umzuwandelnde Inhalt in der WYSIWYG-Ansicht selektiert und die *Create-Template*-Funktion aus der *SemwidgJS Toolbox* ausgewählt werden. Der Dialog zur Template-Generierung wird daraufhin gestartet. Das generierte Template ersetzt den zuvor selektierten Teil der Webseite.

Der Dialog besteht aus zwei Tabs. Im ersten Tab können die Parameter verwaltet werden, die das zu erstellende Template nutzen kann. Hier wird standardmäßig ein Parameter mit der Bezeichnung *value* angelegt. Der zweite Tab (vgl. Abbildung 4.8) ermöglicht die Modifikation der Widget-Parameter aller für die Template-Generierung selektierter SemwidgJS-Widgets. Der Nutzer kann diese Werte händisch ändern oder die Suchen-und-Ersetzen-Funktion nutzen. Für die Suche schlägt das System alle in der Seite definierten Resource-Element-IDs vor. Es wurde davon ausgegangen, das diese in den meisten Fällen die passenden Kandidaten für eine Ersetzung durch Parameter sind. Es können allerdings auch alle anderen Zeichenketten als Sucheingabe genutzt werden. Für das Ersetzen schlägt das System die zuvor im ersten Tab definierten Template-Parameter vor.

Template-Instanziierung

Über die SemwidgED-Toolbar können Template-Instanzen, genau wie Widgets, in die Webseite eingebunden werden. Diese werden an der Stelle der aktuellen Cursor-Position eingefügt. Der

Dialog zur Konfiguration der Template-Instanz listet alle für das Template verfügbaren Parameter auf. Die Autovervollständigung schlägt die für die Website definierten Resource-Element-IDs vor.

4.3.6 Export

Mittels der Export-Funktion aus der *SemwidgED Toolbox* lässt sich die erstellte Webseite speichern. Dabei stehen drei Speicher-Modi zur Auswahl:

Plain data: Dieser Modus speichert den inneren Quelltext der Webseite, also den Inhalt des *Body*-Tags. Dieser Modus kann genutzt werden, wenn der erstellte Inhalt in eine andere Webseite eingebettet werden soll. Für sich allein stellt der exportierte Quelltext kein vollwertiges HTML-Dokument dar. Der Nutzer kann zusätzlich wählen, ob die beiden JavaScript-Bibliotheken jQuery und SemwdigJS mit eingebunden werden sollen.

HTML document: Dieser Modus erzeugt ein vollwertiges HTML-Dokument, welches in diesem Zustand auf einen Webserver geladen werden könnte. Die beiden JavaScript-Bibliotheken jQuery und SemwdigJS werden automatisch eingebunden und einige vordefinierte CSS-Regeln sorgen dafür, dass die Webseite ein ansprechendes Aussehen erhält.

HTML document + Editor: Dieser Modus verhält sich wie der vorherige, bettet aber zusätzlich noch SemwidgED selbst mit in das Dokument ein. Damit wird die exportierte Webseite zu einem aktiven HTML-Dokument, vergleichbar mit Dido (vgl. Abschnitt 4.1.1 bzw. Karger et al., 2009), welches es ermöglicht, die enthaltenen Daten ohne zusätzliche Werkzeuge weiter zu bearbeiten.

Bei allen drei Modi wird dem Nutzer stets der Quelltext des exportierten Inhalts angezeigt, sodass immer ersichtlich ist, welche Daten welcher Modus exportiert.

4.3.7 Fazit

Bei der Implementierung von SemwidgED konnten sämtliche zuvor gestellten Anforderungen (vgl. Abschnitt 4.2.3) umgesetzt werden. Der Editor unterstützt Nutzer bei der Erstellung und Konfiguration von Widgets, Templates und Konfigurationselementen. Ebenso unterstützt er Nutzer bei der Formulierung von SemwidgQL-Abfragen.

Da SemwidgED auf einem erprobten und weit verbreitetem HTML-Editor (CKEditor) basiert, lässt sich neben allen vorhandenen SemwidgJS-Elementen auch regulärer HTML-Quelltext erzeugen, bearbeiten und mit SemwidgJS-Elementen zusammenführen. Dies wäre ohne den CKEditor als Basis, oder auch einen vergleichbaren Editor, in angemessener Zeit nicht umsetzbar gewesen. Allerdings wurden durch den Basis-Editor bei der Implementierung auch mehrere Designentscheidungen aufgezwungen, wodurch einige Operationen, welcher von Nutzern bei der Erstellung und Bearbeitung von SemwidgJS-Elementen durchgeführt werden müssen, umständlicher sind, als sie in einer vollständigen Eigenimplementierung vermutlich sein würden. Die nachfolgend beschriebene Nutzerstudie wird jedoch aufzeigen, dass mit SemwidgED dennoch effektiv, effizient und zufriedenstellen gearbeitet werden kann.

Das SemwidgED-Plug-In für den CKEditor und dessen Quelltext wurden auf der Semwidg-Projektseite[24] unter der MIT-Open-Source-Lizenz veröffentlicht. Ebenso kann der Editor von dort aus direkt ausgeführt werden.

[24] `https://semwidg.org/page/semwidged`

4.4 Empirische Nutzerstudie zur Evaluation der Gebrauchstauglichkeit

In einer empirischen Nutzerstudie wurde die Gebrauchstauglichkeit von SemwidgED evaluiert. Ziel der Studie war es, zu untersuchen, ob die Probanden effektiv und effizient mit SemwidgED arbeiten konnten. Zudem wurde die Nutzerzufriedenheit anhand verschiedener Fragebögen ermittelt. Die Probanden entstammen aus zwei Gruppen. Zum einen aus der eigentlichen Zielgruppe des Semwidg-Projekts – nämlich Anwender mit Programmiererfahrung, die jedoch keine Erfahrung mit dem Semantic Web und Linked-Data-Techniken aufweisen können – und zum anderen aus der Gruppe der Anwender ohne weitreichende Programmiererfahrung. Aufgabe der Probanden war es, Informationen über einen Schauspieler mittels verschiedener Widgets anzuzeigen. Diese Zusammenstellung sollte in ein Template überführt werden und anschließend auf mehrere Schauspieler angewandt werden.

4.4.1 Methodik

Im folgenden Unterabschnitt werden das Studiendesign, die eingesetzten Fragebögen, die Stichprobe, der Ablauf der Studie und die Erfassung der Versuchsdaten beschrieben.

Studiendesign

Die hier beschriebene Nutzerstudie folgt einem Mixed-Methods-Ansatz. Es wurden objektive Messungen der Performanz der Probanden und subjektive Antworten der Probanden aus mehreren Fragebögen betrachtet. Für die Messung der Performanz mussten die Probanden eine szenariobasierte Aufgabe mit SemwidgED bearbeiten, welche aus mehreren aufeinander aufbauende Teilaufgaben bestand.

Die Effektivität wurde anhand der Korrektheit der bearbeiteten Teilaufgaben gemessen. Die Effizient wurde anhand der benötigten Bearbeitungszeit gemessen. Zur Ermittlung der Nutzerzufriedenheit wurden mehrere Fragebögen ausgewertet, die die Probanden während und nach der Bearbeitung der Aufgabe ausfüllten.

Fragebögen

Nach jeder Teilaufgabe wurden die Probanden angewiesen, den *After-Scenario Questionnaire* (ASQ) nach Lewis (1991) zu beantworten. Der Fragebogen besteht aus drei Aussagen (vgl. Tabelle E.2), betreffend die Zufriedenheit der Probanden bei der Bearbeitung der Aufgabe hinsichtlich der Leichtigkeit, des Zeitaufwands und der unterstützenden Informationen. Diese Aussagen sollen anhand einer siebenstufigen Likert-Skala bewertet werden.[25] Aus den Bewertungen der einzelnen Aussagen lässt sich ein Durchschnittswert bilden.

Nach Beendigung aller Teilaufgaben bewerteten die Probanden die zehn Aussagen des *System Usability Scale* (SUS) nach Brooke (1996). Diese zehn Aussagen (vgl. Tabelle E.3) werden anhand einer fünfstufigen Likert-Skala eingeordnet, wobei die Aussagen abwechselnd positiv und negativ formuliert sind. Aus den Bewertungen lässt sich ein Gesamt-Score berechnen, der zwischen 0 und 100 liegt. Je höher der Wert ist, desto gebrauchstauglicher ist das bewertete System. Nach Tullis & Stetson (2004) ist bereits eine geringe Anzahl von Probanden ausreichend, um mittels des SUS-Fragebogens verlässliche Werte über die Gebrauchstauglichkeit eines Systems zu erlangen. Die Items 4 und 10 messen laut Lewis & Sauro (2009) die Erlernbarkeit. Alle anderen Items messen die Gebrauchstauglichkeit.

[25] Im Gegensatz zu den meisten anderen Fragebögen signalisieren niedrige Werte beim ASQ Zustimmung und hohe Werte Ablehnung.

Außerdem füllten die Probanden den *ErgoNorm*-Fragebogen nach Dzida et al. (2000) aus. Der Fragebogen enthält 28 Fragen aus den sieben Kategorien *Aufgabenangemessenheit* (AA), *Selbstbeschreibungsfähigkeit* (SF), *Steuerbarkeit* (ST), *Erwartungskonformität* (EK), *Fehlertoleranz* (FT), *Individualisierbarkeit* (IN) und *Lernförderlichkeit* (LF) (vgl. Tabelle E.4). Zu jeder Frage kann eine der Optionen »Ja«, »Nein« und »Frage trifft nicht zu« gewählt werden. Bei negativ bewerteten Item soll eine Begründung im Freitext angegeben werden. Zusätzlich kann dann die Option »Ich empfinde dies als sehr störend« markiert werden. Zwei der Fragen wurden im Rahmen der Studie nicht gestellt (AA9 und SF5), da sie nicht zum Szenario passten oder die abgefragte Funktion in SemwidgED nicht enthalten war. Im Gegensatz zu den beiden anderen Fragebögen liefert dieser keine quantitativen Werte, sondern qualitative Aussagen, die dabei helfen können, Usability-Probleme zu identifizieren.

Beschreibung der Stichprobe

An der Studie nahmen 23 Probanden (14 weiblich, 9 männlich) im Alter zwischen 18 und 34 Jahren ($M = 23{,}35$; $SD = 4{,}72$) aus dem universitären Umfeld teil. Als höchsten Bildungsabschluss gaben 17 Probanden die allgemeine Hochschulreife an. Über einen Hochschulabschluss verfügten 6 Probanden. Ein konzeptionelles Vorwissen bezüglich des Semantic Webs und Linked Data war bei lediglich 2 Probanden vorhanden. Jedoch gaben sie an, keine Experten in diesen Bereichen zu sein. Bezüglich allgemeiner Web-Techniken gaben 7 Probanden an, dass sie Erfahrungen in diesem Bereich vorweisen können. Die meisten davon durch den Betrieb einer eigenen Website. Fortgeschrittene Programmiererfahrung war bei insgesamt 7 Probanden vorhanden. Die übrigen 16 Probanden gaben an, dass sie zwar eine entsprechende Vorlesung besucht haben, darüber hinaus jedoch keine weitere Programmiererfahrungen gesammelt haben. Im Rahmen der Auswertung wurden die Probanden in zwei Gruppen unterteilt. Die erste Gruppe bilden die Probanden mit fortgeschrittene Programmiererfahrung, welche im weiteren Verlauf als *Programmierer* bezeichnet werden. Die zweite Gruppe bilden die Probanden mit nur rudimentärer Programmiererfahrung, welche im weiteren Verlauf als *Anfänger* bezeichnet werden. Während die Programmierer die eigentliche Zielgruppe des Semwidg-Projekts widerspiegeln, ist es kein primäres Ziel, auch die Anfänger anzusprechen. Dennoch wäre es wünschenswert, wenn auch diese SemwidgED in zufriedenstellender Weise nutzen können. Die Einordnung der Probanden in die beiden Gruppen entspricht der Klassifikation, welche in Abschnitt 1.2.3 vorgenommen wurde.

Ablauf

Zu Beginn wurden die Probanden über das Ziel der Studie und den Anwendungszweck von SemwidgED informiert. Damit alle Probanden die gleichen Informationen bekamen, wurden ihnen diese in schriftlicher Form vorgelegt (vgl. Abbildung E.1). Zusätzlich wurde den Probanden eine Übersicht über die Funktionen des Editors und die Syntax von SemwidgQL (vgl. Abbildung E.2) ausgehändigt sowie eine Liste mit bekannten Schauspielern für die sichergestellt war, dass der genutzte SPARQL-Endpoint alle geforderten Daten enthielt. Der ebenfalls ausgehändigt Aufgabenzettel enthielt neben den Aufgaben (vgl. Tabelle E.1) die entsprechenden Felder zum Beantworten des ASQ-Fragebogens (vlg. Tabelle E.2).

Nachdem die Probanden die ausgehändigten Materialien studiert hatten, schauten sie sich ein etwa 7 ½ Minuten langes Tutorial-Video an, in dem die Funktionen des Editors erläutert wurden. Dabei wurden alle Aktionen gezeigt, die später von den Probanden für die Bearbeitung der Aufgabe benötigt wurden. Die Probanden hatten die Möglichkeit das Video zu jeder Zeit zu

pausieren, an eine vorherige Stelle zurückzuspringen oder die Wiedergabegeschwindigkeit an ihre Bedürfnisse anzupassen.

Anschließend sollten die Probanden einen Bearbeitungsprozess (Aufgabe 2) eigenständig mit SemwidgED durchführen. Dabei mussten Informationen über einen Schauspieler mittels verschiedener Widgets angezeigt werden. Diese Zusammenstellung musste in ein Template überführt und anschließend auf mehrere Schauspieler angewandt werden. Dieser Prozess war in insgesamt sieben Teilaufgaben unterteilt. Nach jeder Teilaufgabe wurden die Probanden angewiesen, die Nutzung von SemwidgED anhandes des ASQ zu bewerten.

Abschließend bewerteten sie die Gebrauchstauglichkeit von SemwidgED anhand des SUS- und ErgoNorm-Fragebogens. Die Teilnahme wurde auf Wunsch mit einer Versuchspersonenstunde[26] vergütet. Eine finanzielle Vergütung wurde nicht gewährt.

Datenerfassung

Bei der Arbeit mit SemwidgED wurde bei allen Probanden der Bildschirminhalt in einem Screencast aufgezeichnet. Dadurch konnten nachträglich die genauen Bearbeitungszeiten der einzelnen Teilaufgaben ermittelt werden. Die Bearbeitungszeit begann mit der ersten Bewegung des Mauszeigers nach dem Lesen des Aufgabentextes und endete mit dem letzten Button-Klick, der für den Abschluss der jeweiligen Teilaufgabe benötigt wurde. Zusätzlich konnten so Probleme, die die Probanden bei der Bearbeitung der Teilaufgaben hatten, nachträglich ermittelt werden. Die Bewertungen der Aussagen des ASQ erfolgte direkt schriftlich auf dem Aufgabenzettel. Die Erfassung der Daten der beiden anderen Fragebögen (SUS und ErgoNorm) und der soziodemographischen Daten erfolgte über das Onlinebefragungsportal SoSci Survey[27].

4.4.2 Ergebnisse

Im folgenden Unterabschnitt sollen die Ergebnisse der Nutzerstudie vorgestellt werden. Zu Beginn wurde die Korrektheit der Antworten untersucht. Anschließend wurden die Ergebnisse hinsichtlich der Bearbeitungszeit analysiert. Insbesondere wurde der Unterschied der Leistung zwischen Programmieren und Anfängern betrachtet. Anschließend wurden die subjektiven Einschätzungen der Probanden untersucht, welche sich aus der Auswertung der Fragebögen ASQ, SUS und ErgoNorm ergab. Abschließend werden die Probleme der Probanden bei der Bearbeitung der Aufgaben analysiert, die anhand der Screencasts ermittelt werden konnten.

Die Auswertung der quantitativen Daten erfolgte deskriptiv und vergleichend. Ergebnisse, die als statistisch signifikant bezeichnet werden, genügen mindestens einem fünfprozentigen Signifikanzniveau. Die Effektstärken wurden anhand der Kategorisierung von Cohen (1992) beurteilt.

Korrektheit der Antworten

Sämtliche Probanden waren in der Lage, die ihnen gestellten Aufgaben zu lösen. Dazu benötigten sie die zusätzlichen Informationen, die ihnen in Form des Handouts (vgl. Abbildung E.2) zur Verfügung gestellt wurden. Kein Proband kam ohne diese Hilfestellung aus.

[26] Versuchspersonenstunden (VPN-Stunden) sind Bescheinigungen über die Teilnahme an wissenschaftlichen Studien und deren Dauer. Studierende des Master-Studiengangs Angewandte Kognitions- und Medienwissenschaft an der Universität Duisburg-Essen benötigen eine vorgegebene VPN-Stundenanzahl, um zu bestimmten Klausuren zugelassen zu werden.

[27] `https://www.soscisurvey.de`

Tabelle 4.3: Vergleich der Bearbeitungszeiten (s) der Teilaufgaben und der Gesamtzeit zwischen Programmierern und Anfängern.

	Programmierer[a]		Anfänger[b]		t-Test				
	M	*SD*	*M*	*SD*	*t*	*df*	*p*	*r*	
Aufgabe 2a	108,86	63,29	108,53	62,97	0,011	20	,991	,000	
Aufgabe 2b	163,71	45,80	250,40	95,72	-2,257	20	,035	,211	*
Aufgabe 2c	86,14	52,34	181,73	90,63	-2,576	20	,018	,259	*
Aufgabe 2d[c]	89,00	31,47	203,87	80,83	-4,782	19,718	<,001	,550	***
Aufgabe 2e[c]	106,86	28,77	203,40	64,86	-4,835	19,999	<,001	,552	***
Aufgabe 2f	69,71	31,14	84,20	45,52	-0,758	20	,457	,029	
Aufgabe 2g[c]	170,43	95,57	187,13	54,44	-0,431	7,877	,678	,026	
Aufgabe 2 (∑)	794,71	206,26	1219,27	248,17	-3,924	20	<,001	,448	***

[a] (n = 7) (*: p <,05; **: p <,01; ***: p <,001)

[b] (n = 16)

[c] (Da keine Varianzhomogenität vorlag, wurde ein t-Test mit Welch-Korrektur berechnet)

Tabelle 4.4: Vergleich der Bearbeitungszeiten (s) der Teilaufgaben 2b und 2d.

	Aufgabe 2b		Aufgabe 2d		t-Test				
	M	*SD*	*M*	*SD*	*t*	*df*	*p*	*r*	
Programmierer	163,71	45,80	89,00	31,47	7,320	6	<,001	,915	***
Anfänger	250,40	95,72	203,87	80,83	1,204	14	,249	,100	
Gesamt	222,82	91,74	167,32	87,39	2,103	21	,048	,181	*

(*: p <,05; **: p <,01; ***: p <,001)

Analyse der Bearbeitungsdauer

Im folgenden Abschnitt werden die Zeiten, die für die Bearbeitung der Aufgabe 2, sowohl im Gesamten wie auch bezüglich der einzelnen Teilaufgaben, im Hinblick auf Unterschiede zwischen den beiden Gruppen betrachtet. Eine besondere Aufmerksamkeit liegt dabei auf den beiden Teilaufgaben 2b und 2d. Diese sind fast identisch, wobei die zweite der beiden Aufgaben eine etwas komplexere Abfrage erfordert. Daher kann vermutet werden, dass sich an diesen beiden Aufgaben mögliche Lerneffekte zeigen. Während der Aufgabenbearbeitung durch Proband B02 kam es zu mehreren extern verursachten Verbindungsproblemen zum SPARQL-Endpoint, wodurch die Zeiten verfälscht wurden. Die gemessenen Daten des Probanden wurden daher von der folgenden Untersuchung ausgeschlossen.

Zwischen den beiden Gruppen der Anfänger und Programmierer, gab es erhebliche Unterschiede bezüglich der gesamten Bearbeitungsdauer. Zur Bestimmung der Unterschiede wurde ein t-Test für unabhängige Stichproben berechnet (vgl. Tabelle 4.3). Die Programmierer waren statistisch signifikant schneller als die Anfänger bei der Bearbeitung der gesamten Aufgaben. Die Effektstärke liegt bei r = ,45 und entspricht damit einem mittleren bis starken Effekt. Durchschnittlich benötigten die Anfänger etwa 50 % mehr Zeit als die Programmierer. Werden die Teilaufgaben getrennt betrachtet, zeigt sich, dass beide Gruppen für die erste Teilaufgabe im Durchschnitt die gleiche Zeit benötigten. Bei allen anderen Teilaufgaben waren die Programmierer schneller als die Anfänger. Dieser Unterschied war bezüglich vier der sechs Teilaufgaben statistisch signifikant.

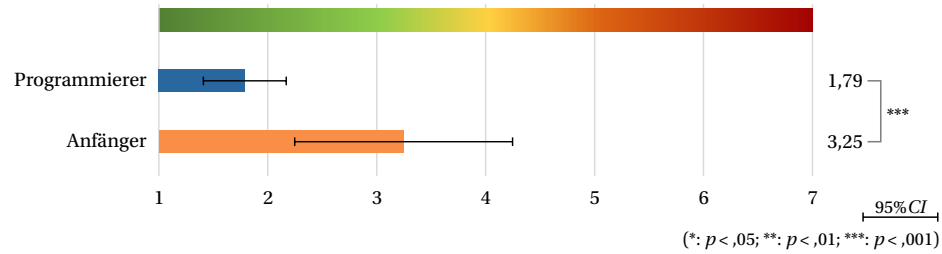


Abbildung 4.9: Vergleich der durchschnittlich vergebenen ASQ-Bewertungen von Programmierern und Anfängern. Je niedriger der Wert ist, desto besser ist die Bewertung.

Zur Bestimmung der Unterschiede zwischen den Teilaufgaben 2b und 2d wurden mehrere t-Tests für abhängige Stichproben berechnet (vgl. Tabelle 4.4). Insgesamt waren die Probanden bei der Bearbeitung der Teilaufgabe 2d leicht statistisch signifikant schneller, als bei der Bearbeitung der Teilaufgabe 2b, bei einem schwachen Effekt. Werden die beiden Gruppen der Anfänger und Programmierer jedoch getrennt betrachtet, zeigt sich, dass sich nur die Programmierer statistisch signifikant – dafür stark und mit einem ebenfalls starken Effekt – verbessert haben. Die Anfänger konnten die benötigte Zeit zwar auch um ungefähr 20 % reduzieren, jedoch ist dieser Unterschied nicht statistisch signifikant. Dennoch konnte ein schwacher Effekt ermittelt werden.

Analyse der Bewertungen des After-Scenario Questionnaires (ASQ)

Nachfolgend werden die Bewertungen des ASQ, welche die Probanden nach der Bearbeitung der einzelnen Teilaufgaben abgegeben haben, untersucht. Der Fokus liegt wieder auf der Analyse der Unterschiede, die sich zwischen Programmieren und Anfängern ergaben. Nach Lewis (1995) ist es zulässig, einen Durchschnittswert aus den drei Items des Fragebogens zu berechnen.

Zur Bestimmung der Unterschiede zwischen Programmieren und Anfängern wurden mehrere Mann-Whitney-U-Tests für unabhängige Stichproben berechnet. Auf der siebenfach abgestuften Likert-Skala des ASQ-Fragebogens haben die Programmierer SemwidgED durchschnittlich mit 1,79 bewertet. Die Anfänger haben den Editor mit 3,25 bewertet. Dieser Unterschied ist statistisch signifikant (vgl. Abbildung 4.9).

Werden die Bewertungen für die einzelnen Teilaufgaben getrennt betrachtet (vgl. Tabelle 4.5), zeigt sich, dass die Programmierer in allen Fällen zufriedener mit SemwidgED waren als die Anfänger. Für fünf der sieben Teilaufgaben war dieser Unterschied statistisch signifikant. Bei diesen Teilaufgaben entsprechen die berechneten Effektstärken jeweils einem starken Effekt. Bei den beiden Teilaufgaben, bei denen keine statistisch signifikanten Unterschiede festgestellt werden konnten, entsprechen die ermittelten Effektstärken immer noch mittleren Effekten.

Werden zusätzlich die einzelnen Items des Fragebogens getrennt betrachtet, zeigt sich, dass die Unterschiede hinsichtlich der Items 1 (Leichtigkeit) und 2 (Zeitaufwand) bei sechs der sieben Teilaufgaben statistisch signifikant war. Die Unterschiede hinsichtlich des dritten Items (unterstützende Informationen) war nur bei drei Teilaufgaben statistisch signifikant. Auch bei den nicht signifikanten Unterschieden konnte mindestens ein schwacher bis mittlerer Effekt festgestellt werden. Sowohl von den Programmierern wie auch von den Anfängern wurde Item

Tabelle 4.5: Vergleich der Ergebnisse der ASQ-Bewertung von Programmierern und Anfängern, aufgeschlüsselt nach den Teilaufgaben und einzelnen Items des Fragebogens.

		Programmierer		Anfänger		Mann-Whitney-U-Test				
	ASQ	*Mdn*	*n*	*Mdn*	*n*	*U*	*z*	*p*	*r*	
Aufgabe 2a	Item 1	2	7	2,5	16	38,5	−1,22	,224	,254	
	Item 2	1	7	2,5	16	21,5	−2,40	,016	,500	*
	Item 3	2	7	3	16	44,0	−0,83	,406	,173	
	Gesamt	1,667	7	2,667	16	33,5	−1,51	,131	,315	
Aufgabe 2b	Item 1	2	7	4,5	16	16,0	−2,73	,006	,568	**
	Item 2	2	7	4	16	26,0	−2,03	,042	,423	*
	Item 3	3	7	4	16	32,0	−1,64	,102	,341	
	Gesamt	2	7	4,5	16	20,0	−2,42	,015	,505	*
Aufgabe 2c	Item 1	1	7	2,5	16	15,5	−2,84	,004	,593	**
	Item 2	1	7	3	16	20,0	−2,52	,012	,524	*
	Item 3	2	7	3,5	16	26,0	−2,04	,041	,426	*
	Gesamt	1,333	7	2,833	16	14,5	−2,79	,005	,581	**
Aufgabe 2d	Item 1	1	7	3,5	16	11,0	−3,10	,002	,646	**
	Item 2	2	7	3	16	22,5	−2,30	,022	,479	*
	Item 3	2	7	4	16	22,0	−2,31	,021	,481	*
	Gesamt	1,333	7	3,5	16	13,5	−2,86	,004	,597	**
Aufgabe 2e	Item 1	2	7	3	16	22,5	−2,30	,021	,480	*
	Item 2	1	7	3	16	24,5	−2,19	,029	,457	*
	Item 3	3	7	3,5	16	34,0	−1,49	,136	,311	
	Gesamt	2	7	3,5	16	22,0	−2,28	,022	,476	*
Aufgabe 2f	Item 1	1	7	2	16	25,0	−2,21	,027	,460	*
	Item 2	1	7	2	16	28,5	−1,96	,050	,409	
	Item 3	2	7	3	16	40,5	−1,08	,282	,224	
	Gesamt	1,667	7	2,333	16	29,5	−1,79	,073	,374	
Aufgabe 2g	Item 1	1	7	2	16	19,5	−2,61	,009	,543	**
	Item 2	1	7	2	16	24,5	−2,40	,016	,500	*
	Item 3	1	7	2,5	16	22,0	−2,37	,018	,493	*
	Gesamt	1	7	2,167	16	18,0	−2,59	,010	,540	*
Aufgabe 2	Item 1	1	49	3	112	1152,0	−6,02	<,001	,474	***
(Gesamt)	Item 2	1	49	3	112	1226,0	−5,77	<,001	,454	***
	Item 3	2	49	3	112	1581,5	−4,35	<,001	,343	***
	Gesamt	1,667	49	3	112	1143,5	−5,91	<,001	,466	***

(*: $p <$,05; **: $p <$,01; ***: $p <$,001)

2 in den meisten Fällen am besten bewertet. Die kritischsten Bewertungen wurden bei beiden Gruppen bezüglich Item 3 gemacht.

Analyse der Bewertungen des System Usability Scales (SUS)

Im weiteren Verlauf werden die Bewertungen des SUS-Fragebogens untersucht, welcher von den Probanden nach der Bearbeitung der gesamten Aufgabe 2 aufgefüllt wurde. Das Hauptaugenmerk liegt wieder auf dem Vergleich der Unterschiede zwischen Programmieren und Anfängern. Zur Bewertung der Unterschiede zwischen den beiden Gruppen wurde für den SUS-Score ein Mann-Whitney-U-Test für unabhängige Stichproben berechnet.

Tabelle 4.6: Vergleich der berechneten SUS-Scores von Programmierern und Anfängern.

	Programmierer		Anfänger		Mann-Whitney-U-Test				
	Mdn	*n*	*Mdn*	*n*	*U*	*z*	*p*	*r*	
SUS-Score	72,5	7	56,25	16	10,0	−3,09	,002	,643	**

(*: $p <$,05; **: $p <$,01; ***: $p <$,001)

Insgesamt wurden deutliche Unterschiede zwischen den Bewertungen der Programmierer und Anfänger ermittelt. Der durchschnittlich ermittelte SUS-Score lag bei den Programmierern bei 75,71 und damit nach Bangor et al. (2009) im Bereich zwischen »Good« und »Excellent«. Bei den Anfängern lag der Wert bei 49,38 und kann damit noch als »OK« eingestuft werden. Der Unterschied zwischen den beiden Bewertungen ist statistisch signifikant (vgl. Abbildung 4.10 und Tabelle 4.6).

Die Resultate der deskriptiven Analyse der einzelnen Items des Fragebogens werden in Abbildung 4.11 in grafischer Form dargestellt. Es zeigt sich, dass die Programmierer SemwidgED hinsichtlich jedes einzelnen Fragebogen-Items besser bewertet haben als die Anfänger. Während unter den Anfängern bei fast allen Items das komplette Bewertungsspektrum genutzt wurde, wurden von den Programmierern nur bezüglich zwei Items negative Bewertungen vergeben.

Analyse der Bewertungen des ErgoNorm-Fragebogens

In diesem Unterabschnitt werden die Bewertungen des ErgoNorm-Fragebogens analysiert. Im Gegensatz zu den vorherigen Untersuchungen wird hier nicht zwischen den beiden Gruppen unterschieden. Zusammengenommen haben die Probanden 598 Fragen beantwortet. Dabei wurde SemwidgED in 413 der Antworten (69 %) positiv und 110 Fragen (18 %) negativ bewertet. Zusätzlich wurden 8 der negativ beantworteten Fragen (1 %) mit »Ich empfinde dies als sehr störend« bewertet. Die restlichen 75 Fragen (13 %) wurden von den Probanden mit »Frage trifft nicht zu« beantwortet.

SemwidgED wurde hinsichtlich der sieben Kategorien des Fragebogens sehr unterschiedlich bewertet. Von den Fragen der Kategorie *Selbstbeschreibungsfähigkeit* wurden 38 % negativ beantwortet (vgl. Abbildung 4.12). In dieser Kategorie finden sich auch 5 der 8 Antworten, die zusätzlich als besonders störend markiert wurden. Die Fragen der Kategorien *Aufgabenangemessenheit*, *Erwartungskonformität* und *Fehlertoleranz* wurden in 14 % bis 20 % der Fälle negativ bewertet. *Steuerbarkeit* und *Lernförderlichkeit* wurden kaum, *Individualisierbarkeit* wurde gar nicht negativ bewertet.

In fast allen Fällen wurde die Frage SF6 negativ bewertet, welche danach fragt, ob der Proband das Handout häufig konsultieren musste (vgl. Abbildung 4.13). Häufig wurde von den Probanden angemerkt, dass sie sich nicht alle Aktionen aus dem Tutorial-Video einprägen konnten und sich daher unsicher bei der Arbeit mit dem Editor fühlten. Entsprechend musste das Handout besonders zu Beginn der Studie häufig konsultiert werden. Alle weiteren Fragen wurden deutlich seltener negativ bewertet. Drei Fragen wurden ausschließlich positiv bewertet.

Zusätzliche Beobachtungen

Abschließend werden einige Beobachtungen aufgeführt, die während der Studie von den Versuchsleitern gemacht wurden. Auffällig war, dass ein Großteil der Programmierer versuchte, die

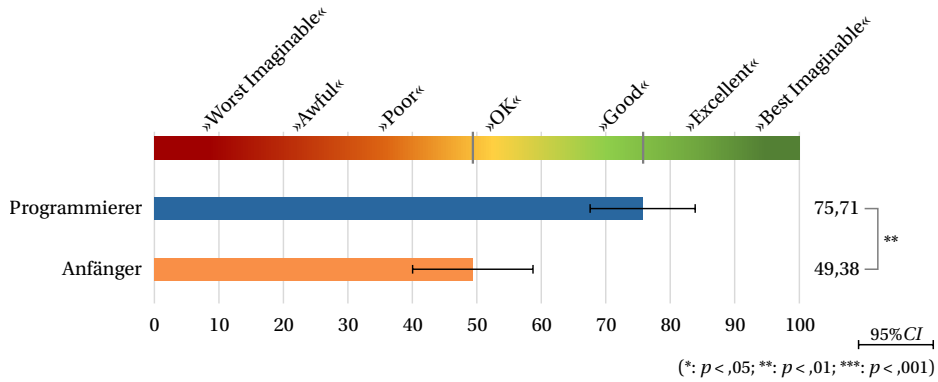


Abbildung 4.10: Vergleich der durchschnittlich vergebenen SUS-Scores von Programmierern und Anfängern. Je höher der Wert ist, desto besser ist die Bewertung. Die Einordnung der Bewertungen basiert auf den Studienergebnissen von Bangor et al. (2009).

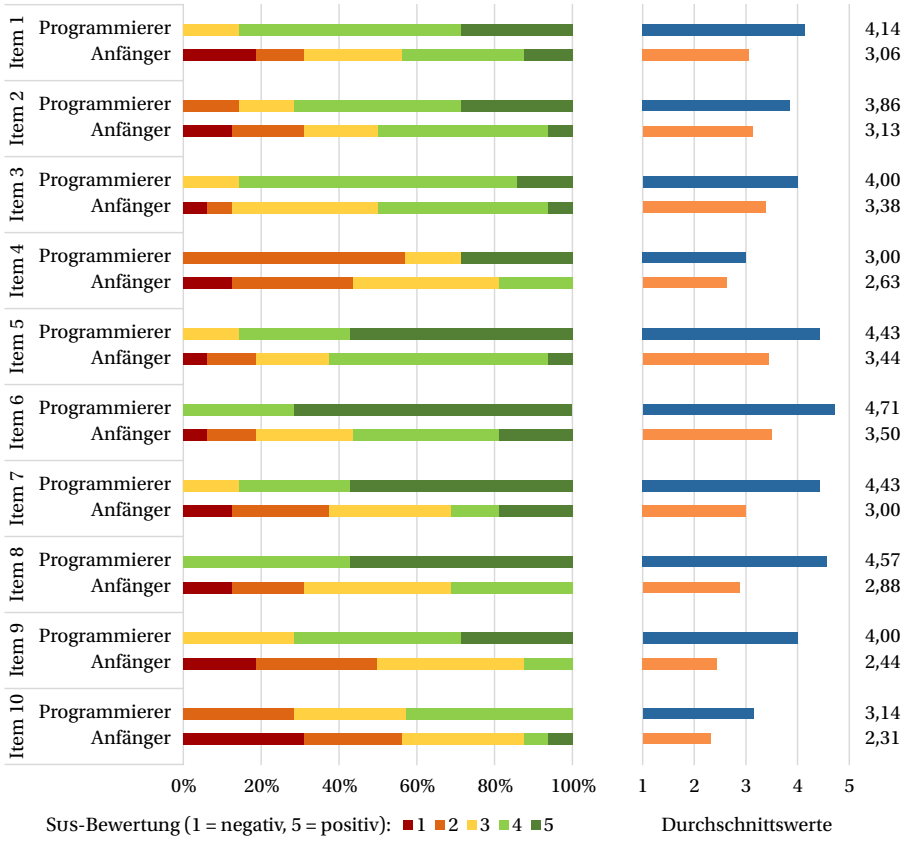


Abbildung 4.11: Vergleich der Ergebnisse der SUS-Bewertung von Programmierern und Anfängern, aufgeschlüsselt nach den einzelnen Items des Fragebogens (vgl. Tabelle E.3). Die Ergebnisse der negativ formulierten Items wurden zur besseren Vergleichbarkeit umgepolt.

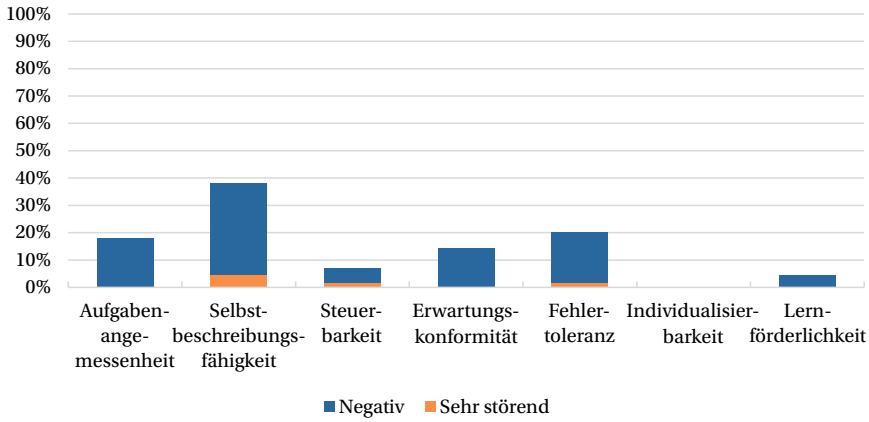


Abbildung 4.12: Relative Anzahl der negativen Bewertungen innerhalb der einzelnen Kategorien des ErgoNorm-Fragebogens.

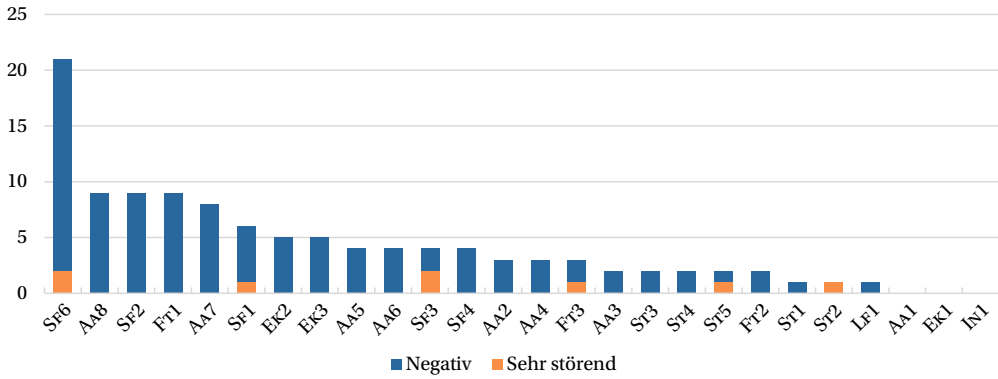


Abbildung 4.13: Absolute Anzahl der negativen Bewertungen der einzelnen Items des ErgoNorm-Fragebogens.

SemwidgQL-Abfragen direkt einzugeben, ohne den Query-Editor zu nutzen. Auch wenn sich Probleme bei der Abfrageformulierung ergaben, änderten sie ihr Vorgehen nicht. Die Anfänger versuchten hingegen nur vereinzelt, die Abfragen direkt einzugeben.

Bei der Bearbeitung von Teilaufgabe 2e, bei der der Geburtsort des gewählten Schauspieles auf einer Karte markiert werden sollte, versuchten fast alle Probanden zuerst, eine Abfrage in das Value-Eingabefeld einzugeben, wie sie es von den vorherigen Widgets gewohnt waren. Das Karten-Widget zeigt dieses Value-Eingabefeld zwar an, es wird jedoch nicht ausgewertet.[28] Dieses Verhalten verunsicherte einige der Probanden.

Ein weiteres Problem zeigte sich gelegentlich bei der Erstellung von Abfragen mit Hilfe des Query-Editors. Einige Probanden versuchten nachträglich den Mittelteil der Abfrage zu verändern. Da der Editor für die Autovervollständigung nicht nur den vorherigen Teil der Abfrage,

[28] Die Eigenschaft »value« ist eine Eigenschaft des Basis-Widgets und wird von allen abgeleiteten Widgets geerbt. Ob und wie diese Eigenschaft genutzt werden soll, entscheidet der Widget-Programmierer. Da SemwidgED nicht ermitteln kann, ob eine Eigenschaft auch benötigt wird, werden entsprechende Eingabefelder für alle verfügbaren Eigenschaften angezeigt.

sondern auch den nachfolgenden Teil miteinbezieht, wurde den Probanden nur noch eine Teilmenge der vorhandenen Eigenschaften angezeigt, in denen die von ihnen gesuchte Eigenschaft nicht enthalten war.

4.4.3 Diskussion der Ergebnisse

Anhand der Ergebnisse der Nutzerstudie konnte gezeigt werden, dass die Probanden aus der Zielgruppe des Semwidg-Projekts effektiv und effizient mit SemwidgED arbeiten konnten. Die Bewertungen der Programmierer weisen darauf hin, dass sie mit dem Editor sehr zufrieden waren. Sie bewerteten den Editor anhand des SUS-Fragebogens als »Good« bis »Excellent«. Die Ergebnisse des ASQ sind sogar noch etwas besser.

Die Ergebnisse hinsichtlich der Probanden außerhalb der Zielgruppe waren hingegen erheblich schlechter. Der Vergleich der beiden Teilaufgaben 2b und 2d hat gezeigt, dass Programmierer schneller ein tieferes Verständnis der Abfragesprache erlangt haben als die Anfänger, und Lerneffekte dadurch deutlich stärker erkennbar waren. Bei den Anfängern konnten hingegen keine statistisch relevanten Verbesserungen festgestellt werden. Es ist daher davon auszugehen, dass einige grundlegende Funktionsweisen der Abfragesprache und des Editors in der kurzen Zeit von den Anfängern nicht verinnerlicht werden konnten. Viele der Anfänger merkten daher auch an, dass sie vermutlich deutlich besser mit SemwidgED hätten arbeiten können, wenn sie sich das Tutorial-Video noch einmal hätten anschauen dürfen. In einem realen Anwendungsszenario, in dem die Nutzer das Tutorial-Video und die weiteren Material der SemwidgED-Webseite nutzen könnten, würden diese vermutlich deutlich bessere Ergebnisse als die Probanden erzielen.

Letztendlich konnten aber auch die Anfänger alle an sie gestellten Aufgaben mit SemwidgED lösen. Dabei nutzten sie unbekannte Daten aus einer Datenbank, mit einer für sie unbekannten Datenstruktur, und fragten diese mit einer für sie unbekannten Abfragesprache ab. Zur Vorbereitung auf diese Aufgaben bekamen sie lediglich ein 7 ½ Minuten langes Tutorial-Video vorgezeigt und ein einseitiges Handout als Gedächtnisstütze ausgehändigt. Auch die Probanden außerhalb der Zielgruppe konnten damit effektiv mit SemwidgED arbeiten. Allerdings haben die Probanden aus der eigentlichen Zielgruppe statistisch signifikant besserer Ergebnisse bezüglich der Effizient erzielt und statistisch signifikant besser Bewertungen bezüglich der Nutzerzufriedenheit abgegeben.

4.5 Diskussion und Fazit

In diesem Kapitel wurde mit SemwidgED ein Online-WYSIWYG-Editor zur Erstellung von Webseiten mit integrierten SemwidgJS-Widgets vorgestellt. Der Editor unterstützt den Nutzer bei der Formulierung des statischen HTML-Textes, der Konfiguration von SemwidgJS, der Einbindung von Widgets, der Formulierung von SemwidgQL-Abfragen und der Generierung von Templates. Des weiteren genügt die Implementierung von SemwidgED (vgl. Abschnitt 4.3) einer Menge von Anforderungen die sich im Rahmen von konzeptionellen Überlegungen (vgl. Abschnitt 4.2) ableiten ließen. Diese fußen wiederum auf allgemeinen Anforderungen die sich aus dem Funktionsumfang von SemwidgJS und SemwidgQL ergaben sowie auf verwandten Arbeiten aus der Industrie und Wissenschaft (vgl. Abschnitt 4.1).

Bei der Betrachtung der verwandten Arbeiten konnten drei Anforderungen als besonders bedeutend herausgearbeitet werden. Der Editor sollte sich erstens ohne fortgeschrittene Programmierkenntnisse nutzen lassen, zweitens den Nutzern viele Freiheiten bei der Gestaltung geben und sie drittens bei der Erstellung der Webseite unterstützen. Auch wenn die erste und dritte Anforderung sehr ähnlich erscheinen, sind sie dennoch nicht identisch und bedingen sich auch nicht gegenseitig. Es sind sowohl Systeme denkbar, die ohne fortgeschrittene Programmierkenntnisse nutzbar sind, Nutzer aber nicht bei der Erstellung der Webseite unterstützen, wie auch der umgekehrte Fall. Keines der fünf untersuchten Systeme konnte alle drei Anforderungen erfüllen. Je mehr Freiraum den Nutzern bei der Gestaltung der Webseite gegeben wurde, umso mehr Wissen mussten sie im Bereich der Webprogrammierung vorweisen können. Insbesondere die freie Gestaltungsmöglichkeit der Darstellung der Daten durch Nutzer und die Nutzbarkeit des Editors ohne Programmierkenntnisse standen sich bei den hier untersuchten Editoren diametral entgegen. Entweder richtete sich einer der untersuchten Editoren an Experten oder an Anfänger. Jedoch nie an beide Gruppen gleichzeitig. SemwidgED wurde so entwickelt, dass es alle drei Anforderungen bedienen kann, indem die Funktionen für Experten und Anfänger gleichberechtigt nebeneinander existieren ohne sich gegenseitig zu behindern.

Dies zeigt sich insbesondere in der Eingabeunterstützung für SemwidgQL-Abfragen. Während sich die textbasierte Eingabe an die erfahrenen Nutzer richtet, können Anfänger eine formularbasierte Eingabe nutzen. Diese erfordert zwar bei der Abfrageerstellung einen höheren zeitlichen Aufwand als die textbasierte Eingabe, unterstützt den Nutzer jedoch stärker und führt ihn gerichtet durch den Formulierungsprozess. Dennoch bildet formularbasierte Eingabe den vollständigen Umfang von SemwidgQL ab und Nutzer können unmittelbar zwischen beiden Varianten der Eingabe wechseln.

Das SemwidgED-Plug-In für den CKEditor und dessen Quelltext wurden auf der Semwidg-Projektseite[29] unter der MIT-Open-Source-Lizenz veröffentlicht und steht allen Interessenten vollumfänglich zur Verfügung. Der CKEditor selbst steht unter der BSD-Lizenz. Zusätzlich wurde ein ausführliches Tutorial erstellt. Dieses existiert sowohl in textueller Form[30] wie auch als kommentiertes Video[31]. Eine verkürzte Version dieses Videos wurde im Rahmen der Nutzerstudie verwendet. Eine Demo-Version des Editors[32] kann auf der Webseite direkt getestet werden und bietet interessierten Nutzern einen leichten Einstieg in das Semwidg-Projekt.

In erster Linie richtet sich das Semwidg-Projekt und damit natürlich auch SemwidgED an Nutzer mit fortgeschrittener Programmiererfahrung, die jedoch keine oder nur wenig Erfahrung mit dem Semantic Web und Linked-Data-Techniken aufweisen können. Dennoch

[29] https://semwidg.org/page/semwidged
[30] https://semwidg.org/page/semwidged-tutorial
[31] https://semwidg.org/page/semwidged-tutorial-video
[32] https://semwidg.org/page/semwidged-editor

wurde es als wünschenswert angesehen, wenn auch Anwender ohne Programmierkenntnisse SemwidgED in zufriedenstellender Weise nutzen können. Die Gebrauchstauglichkeit von SemwidgED wurde in einer empirischen Nutzerstudie evaluiert (vgl. Abschnitt 4.4). Hierbei wurde untersucht, ob die Probanden effektiv und effizient mit dem Editor arbeiten konnten. Zudem wurde die Nutzerzufriedenheit anhand verschiedener Fragebögen ermittelt. An der Studie haben 23 Probanden aus den beiden Gruppen der »Programmierer« und »Anfänger« teilgenommen. Während die Anfänger lediglich mit Hilfe von Website-Baukästen und WYSIWYG-Editoren in der Lage sind, Webseiten zu erstellen, verfügen die Programmierer auch über fortgeschrittene Programmierkenntnisse auf die sie bei der Website-Erstellung zurückgreifen können. Jedoch sind weder die Anfänger noch die Programmierer stärker mit den Techniken und Anwendungen des Linked-Data-Bereichs vertraut. Während die Programmierer, also die eigentlichen Zielgruppe von SemwidgED, sehr gute Ergebnisse bei der Nutzung erzielten, erzielten die Anfänger deutlich schlechtere Ergebnisse. Dementsprechend haben diese den Editor auch hinsichtlich der Nutzerzufriedenheit schlechter bewertet als die Programmierer. Letztendlich konnten aber sämtliche Probanden alle an sie gestellten Aufgaben mit SemwidgED lösen, und das obwohl sie zur Einführung nur ein kurzes Tutorial-Video gesehen und ein einseitiges Handout ausgehändigt bekommen haben.

Es lässt sich festhalten, dass sowohl Anfänger wie auch Programmierer in der Lage sind, mit Hilfe von SemwidgED Webseiten effektiv mittels SemwidgJS-Widgets zu erweitern. Die Effizienz und Nutzerzufriedenheit hängt jedoch stark von der Nutzergruppe ab, wobei die eigentliche Zielgruppe sehr gute Ergebnisse erzielt und Bewertungen abgegeben hat. Die an SemwidgED gestellten Anforderungen können also als erfüllt betrachtet werden.

KAPITEL 5

Abschlussbetrachtungen

In der nachfolgenden Abschlussbetrachtung werden die geleisteten Arbeiten innerhalb des Semwidg-Projekts in ihrer Gesamtheit analysiert. Dazu erfolgt zu Beginn dieses Kapitels eine Zusammenfassung des gesamten Semwidg-Projekts (vgl. Abschnitt 5.1), gefolgt von einer Diskussion der erzielten Ergebnisse sowie einer kritischen Auseinandersetzung mit den Stärken und Limitationen der aus dem Projekt hervorgegangenen Werkzeuge mit Fokus auf die zu Beginn aufgestellten Forschungsfragen (vgl. Abschnitt 5.2). Das Kapitel endet mit einem Ausblick auf potenziell an die hier vorgelegte Arbeit anschließende Forschungsfragen sowie einer Betrachtung von Optimierungsmöglichkeiten von bestimmten Teilaspekten des Semwidg-Projekts (vgl. Abschnitt 5.3).

5.1 Zusammenfassung

Obwohl die Menge der frei verfügbaren, in Graph-Form verlinkten Daten seit den Anfängen des Semantic Webs stetig anwächst, findet eine Nutzung beziehungsweise Mehrfachverwendung dieser Daten durch private Nutzer aber auch semiprofessionelle Webentwickler und -autoren kaum statt. Als Hauptgrund dafür wurde in dieser Arbeit die enorme Komplexität der zur Nutzung benötigten Techniken, Anwendungen und Formate identifiziert.

Ziel dieser Arbeit war es, diese Komplexität zu minimieren und somit Anwendern ohne Fachkenntnisse im Linked-Data-Bereich die Nutzung dieser Daten zu ermöglichen. Dazu wurde hier ein System zur Präsentation und Visualisierung von Linked Open Data in Webseiten mittels Semantic Data Widgets vorgestellt. Diese Semantic Data Widgets kapseln Datenabfragen und präsentationsrelevante Informationen, die benötigt werden, um Daten aus der LOD-Cloud abzurufen, sie zu verarbeiten und anschließend innerhalb einer Webseite anzuzeigen. Gleichzeitig sind sie wiederverwendbar und lassen sich zum Beispiel durch Parametrisierung der Abfrage auf weitere Daten anwenden (vgl. Abschnitt 1.2.2). Zur Umsetzung des Gesamtsystems wurde das Semwidg-Projekt initiiert. Dieses untergliedert sich in die drei Teilprojekte SemwidgQL, SemwidgJS und SemwidgED, welchen jeweils ein Kapitel dieser Arbeit gewidmet wurde.

Mit SemwidgQL (vgl. Kapitel 2) wurde eine leicht zu erlernende Pfadabfragesprache entwickelt, die sich in ihrer Syntax und Semantik an den technischen Vorkenntnissen von Programmierern und Webentwicklern orientiert. SemwidgQL lässt sich clientseitig in SPARQL transkompilieren und kann so mit nahezu allen öffentlichen SPARQL-Endpoints genutzt werden, abhängig von den dort unterstützen SPARQL-Funktionen. Eine empirische Nutzerstudie zeigte, dass die Probanden SemwidgQL mindestens so effektiv nutzen konnten wie SPARQL, sie dabei wesentlich effizienter waren und SemwidgQL leichter erlernen konnten. Zudem präferierte die Mehrzahl der Studienteilnehmer SemwidgQL gegenüber SPARQL als Abfragesprache. Eine weitere Untersuchung konnte zeigen, dass trotz der Einschränkungen von SemwidgQL, fast alle Abfragen, welche im Alltagsbetrieb an einige der großen SPARQL-Endpoints der LOD-Cloud gerichtet wurden, mit SemwidgQL abgebildet werden können.

Mit SemwidgJS (vgl. Kapitel 3) wurde eine Widget-Bibliothek und zugleich ein Framework zur Präsentation und Visualisierung von Daten aus öffentlichen SPARQL-Endpoints auf Basis von JavaScript entwickelt. SemwidgJS integriert SemwidgQL zur vereinfachten Definition von Datenabfragen, erlaubt aber auch zusätzlich die Verwendung von SPARQL als Abfragesprache. Die Widgets ermöglichen die einfache Einbindung von Daten der LOD-Cloud in beliebige Webseiten. In der Regel ist für die Instanziierung eines Widgets lediglich die Angabe einer einzelnen Datenbankabfrage erforderlich. Widgets können untereinander Daten austauschen oder durch weitere JavaScript-Programme manipuliert werden, wodurch sich leicht interaktive Webseiten erstellen lassen. Mittels einer Template-Funktion können ganze Widget-Kompositionen mit abgeänderten und frei definierbaren Parametern wiederverwendet werden. SemwidgJS kam im Rahmen des Eurostars-geförderten Projekts DESUMA (Decision Support Marketplace) zum Einsatz und konnte dort seine Nützlichkeit belegen.

Mit SemwidgED (vgl. Kapitel 4) wurde ein Online-WYSIWYG-Editor entwickelt, der die Nutzer bei der Einbindung von Linked Open Data in ihre Webseiten unterstützt. Ebenso unterstützt der Editor die Nutzer bei der Formulierung von SemwidgQL-Abfragen. Die Gebrauchstauglichkeit von SemwidgED wurde in einer empirischen Nutzerstudie hinsichtlich Effektivität, Effizient und Nutzerzufriedenheit untersucht. Sämtliche Probanden konnten alle an sie gestellten Aufgabe nach Ansicht eines kurzen Tutorial-Videos lösen, wobei die beobachteten Er-

gebnisse jedoch stark von der Nutzergruppe abhingen, aus welcher die Probanden stammten. Die Probanden aus der Hauptzielgruppe des Semwidg-Projekts (vgl. Abschnitt 1.2.3) erzielten sehr gute Ergebnisse und gaben entsprechend gute Bewertungen ab.

Im nachfolgenden Abschnitt werden die Ergebnisse des Semwidg-Projekts diskutiert und es erfolgt eine kritische Auseinandersetzung hinsichtlich der Stärken und Limitationen der daraus hervorgegangenen Werkzeuge.

5.2 Diskussion

In diesem Abschnitt werden die erzielten Ergebnisse bezüglich der zu Beginn aufgestellten Forschungsfragen (vgl. Abschnitt 1.2.1) diskutiert. Ebenso erfolgt eine kritische Auseinandersetzung hinsichtlich der Stärken und Limitationen der aus der Arbeit hervorgegangenen Werkzeuge, die in ausführlicher Form bereits in den jeweiligen Teilkapitel zu SemwidgQL (vgl. Abschnitt 2.6), SemwidgJS (vgl. Abschnitt 3.5) und SemwidgED (vgl. Abschnitt 4.5) vorgenommen wurde.

Forschungsfrage 1: »Wie lassen sich Daten aus der LOD-Cloud auch von Programmierern und Webentwicklern abrufen, die keine Linked-Data-Experten sind?«

Zur Beantwortung der ersten Forschungsfrage wurde mit SemwidgQL eine leicht zu erlernende Pfadabfragesprache entwickelt, die sich in einer empirischen Nutzerstudie gegenüber der RDF-Standardabfragesprache SPARQL behaupten konnte. In einer weiteren Studie konnte dargelegt werden, dass sich fast alle SPARQL-Abfragen des Alltagsbetriebs durch SemwidgQL-Abfragen substituieren lassen. Beide Studien wurden im Rahmen der »16th International Semantic Web Conference (ISWC 2017)« präsentiert und veröffentlicht. Davor haben sie jeweils ein Peer-Review-Verfahren durchlaufen.

Die im Rahmen der vergleichenden Studie hervorgegangene Komplexitätsmetrik für Abfragen im Linked-Data-Bereich, unabhängig einsetzbar von der genutzten Abfragesprache, stellt ein Novum in diesem Forschungsbereich dar und geht dabei deutlich über die Klärung der hier besprochenen Forschungsfrage hinaus. Diese Metrik beschreibt den mentalen Aufwand, den die Formulierung einer Abfrage auf einen RDF-Graphen erfordert und ermöglicht somit den Vergleich verschiedener Abfragesprachen hinsichtlich einer bisher kaum beachteten Variablen. Auch der Vergleich der beiden Abfragesprachen selbst ist in diesem Forschungsbereich neu. Es existiert zwar eine Vielzahl von Abfragesprachen für RDF-Daten, doch konnte auch nach intensiver Recherche keine Publikation gefunden werden, in der verschiedene Abfragesprachen des Linked-Data-Bereichs in einer empirischen Nutzerstudie verglichen wurden. Alle bisherigen Studien beschränken sich auf den Vergleich von Benchmarks oder anderen technischen Werte.

Trotz der guten Studienergebnisse sollte SemwidgQL aber nicht als vollständiger Ersatz für SPARQL verstanden werden, da die vereinfachte Syntax der Sprache einige Limitationen nach sich zieht. Auch wenn, wie sich gezeigt hat, die meisten SPARQL-Abfragen des Alltagsbetriebs in Form einer SemwidgQL-Abfrage formuliert werden können, existieren immer noch unendlich viele SPARQL-Abfragen die nicht durch SemwidgQL substituiert werden können. Sämtliche Abfragen die komplexe UNION-Ausdrücke enthalten und sich nicht umformulieren lassen, können nicht mit SemwidgQL abgebildet werden. Ebenso unterstützt SemwidgQL keine GROUP-BY-Ausdrücke.

Die Stärken von SemwidgQL liegen in der leichteren Erlernbarkeit im Vergleich zu SPARQL, was sie für Einsteiger in den Linked-Data-Bereich besonders attraktiv macht, und in der effizienteren Nutzbarkeit, wodurch sich auch für SPARQL-Experten erhebliche Vorteile bieten.

Forschungsfrage 2: »Mit welchen Mitteln und Techniken lassen sich die abgerufenen Daten in reguläre Webseiten einbinden, ohne einen bereits vorhandenen Publikationsprozess oder sogar die Serverinfrastruktur fundamental umzustrukturieren oder zu ersetzen?«

Zur Beantwortung der zweiten Forschungsfrage wurde auf das Konzept der Semantic Data Widgets zurückgegriffen (vgl. Abschnitt 1.2.2), welches bereits in vorhergehenden Arbeiten entworfen, im Rahmen des Semwidg-Projekts weiterentwickelt wurde und diesem gleichzeitig seinen

Namen verlieh. SemwidgJS setzt dieses Konzept in Form einer Widget-Bibliothek um. Diese enthält vordefinierte Datenpräsentationen und Visualisierungen, die die anzuzeigenden Daten selbstständig durch Angabe einer SemwidgQL- oder auch SPARQL-Abfragen aus einem SPARQL-Endpoint abrufen und verarbeiten. Über zusätzliche Parameter können diese Widgets noch weiter angepasst werden. Ein Templating-Mechanismus ermöglicht die Wiederverwendung von Widget-Kompositionen innerhalb einer Webseite. Widgets können miteinander kommunizieren, wodurch sich Nutzerinteraktionen mit einem Element auf weitere Widgets auswirken können. Hierdurch wird die Erstellung interaktiver Webanwendungen ermöglicht.

Da SemwidgJS all diese Funktionen clientseitig bereitstellt und lediglich die Einbindung der SemwidgJS-Bibliothek in die entsprechende Webseite erfordert, ist auch keine weitere Anpassung der serverseitig genutzten Programme vonnöten, welche im Rahmen des Publikationsprozesses eingesetzt werden. Eine Anpassung der Serverinfrastruktur ist daher ebenfalls nicht notwendig. Die Widgets der SemwidgJS-Bibliothek werden innerhalb des Quelltextes der Webseite in Form von HTML-Elementen deklariert, was jedes CMS und jede Blogging-Plattform ohne Anpassung unterstützen sollte.

Jedoch weist SemwidgJS auch einige Limitationen auf, die die Nutzung in bestimmten Einsatzszenarios erschweren oder gar ausschließen. SemwidgJS verfügt über keine Möglichkeit zum Log-in in passwortgeschützte, nicht öffentliche SPARQL-Endpoints und ist somit auf die Nutzung öffentlicher SPARQL-Endpoints beschränkt. Ebenso hat es mit den gleichen Problemen wie jedes andere Tool zu kämpfen, welches seine Daten aus der LOD-Cloud bezieht, nämlich mit der teilweise unbefriedigende Verarbeitungsgeschwindigkeit der Abfragen und der eingeschränkten Verlässlichkeit der entsprechenden SPARQL-Endpoints. Da SemwidgJS ausschließlich lokal arbeitet, ist es von diesen Problemen stärker betroffen als vergleichbare Serveranwendungen. SemwidgJS verfügt zwar über ein Caching-System zur Zwischenspeicherung einmal erhaltene Abfrageergebnisse, doch liegen diese erst nach der ersten Abfrage vor, welche aufgrund der zuvor beschriebenen Unzulänglichkeiten der SPARQL-Endpoints auch fehlschlagen kann. Serveranwendungen können hier immer auf zwischengespeicherte Daten zurückgreifen, vorausgesetzt ein Autor testet seine Abfragen, bevor er einen Text publiziert.

Dementsprechend ist SemwidgJS auch nicht für den professionellen Einsatz gedacht, sondern wurde von Anfang an für den privaten und semiprofessionellen Einsatz konzipiert. Hier kann es seine Stärken in Form der einfachen Einbindung in bestehende Webseiten und seinen breiten Funktionsumfang ausspielen.

Forschungsfrage 3: »Wie können Anwender bei der Einbindung der Daten in die Webseite unterstützt werden, um den daraus resultierenden Arbeitsaufwand so weit wie möglich zu reduzieren?«

Zur Beantwortung der dritten Forschungsfrage wurde der Online-WYSIWYG-Editor SemwidgED entwickelt, der sich in eine Vielzahl bestehender CMS-Systeme und Blogging-Plattformen integrieren lässt. Entweder indem der bisherige HTML-Editor des Systems komplett durch SemwidgED ersetzt wird oder, wenn dieses den selben Basis-Editor nutzt wie SemwidgED, durch Installation des SemwidgED-Plug-Ins.

SemwidgED unterstützt die Nutzer bei der Konfiguration von SemwidgJS, der Deklaration von Widgets, der Verbindung mehrerer Widgets und der Generierung von Templates. Zudem assistiert der Editor dem Nutzer bei der Formulierung von SemwidgQL-Abfragen. Dabei bietet er zwei Funktionsmodi. Ein formularbasierter Modus führt den Nutzer gerichtet durch den Formulierungsprozess der SemwidgQL-Abfrage. Eine falsche syntaktische Struktur innerhalb

der Abfrage wird somit grundsätzlich vermieden. Mögliche valide Resources und Eigenschaften werden dem Nutzer zur Auswahl vorgeschlagen. Die Auswahl eines solchen Vorschlags garantiert, dass der daraus entstehende Abfragetext mindestens ein Ergebnis bei der Abfrage des SPARQL-Endpoints zurückliefert. Weiterhin verfügt SemwidgED über einen einfachen textbasierten Modus, in welchem dem fortgeschrittenen Nutzer während der Formulierung Vorschläge für die Vervollständigung seiner Eingabe unterbreitet werden, ähnlich der Unterstützung zur Code-Vervollständigung bei der Programmierung mittels einer integrierten Entwicklungsumgebung.

Dass effektives und effizientes Arbeiten mit SemwidgED möglich ist, konnte in einer empirischen Nutzerstudie gezeigt werden. Ebenso wurde die Nutzerzufriedenheit untersucht. Alle Probanden der Studie waren nach Betrachtung eines kurzen Tutorial-Videos in der Lage, sämtliche an sie gestellten Aufgaben in einer zufriedenstellenden Zeit zu erledigen. Allerdings unterschieden sich sowohl die Werte der beiden teilnehmenden Nutzergruppen bezüglich der Effizienz wie auch der subjektiven Nutzerzufriedenheit signifikant voneinander. Während Probanden aus der Hauptzielgruppe des Semwidg-Projekts (vgl. Abschnitt 1.2.3) sehr gute Werte erzielten und Bewertungen abgaben, waren diese im Fall der anderen Gruppe deutlich schlechter aber immer noch zufriedenstellend.

Verglichen mit anderen untersuchten Editoren und Systemen, konnte für SemwidgED dennoch eine gute Balance zwischen der Unterstützung für Nutzer der Hauptzielgruppe und denen der anderen Gruppen gefunden werden. Alle anderen Systeme waren in ihrem Funktionsumfang entweder so eingeschränkt, dass sie zwar auch von absoluten Laien in den Bereichen Web-Programmierung und Linked Data genutzt werden konnten, jedoch keine freie Gestaltungsmöglichkeit der Datendarstellung mehr vorhanden war, oder die Systeme waren so komplex, dass sie nur noch von Experten genutzt werden konnten.

Zusammenführung der drei Teilprojekte

Zusammengenommen bilden alle drei Teilprojekte ein System, das es Nutzern mit einem informatischen Vorwissen und Programmierkenntnissen ermöglicht, und in einem begrenzen Maß auch Nutzern ohne diese Fähigkeiten, bestehende Daten aus der LOD-Cloud in Webseiten einzubinden und somit wiederzuverwenden. Die vordefinierten Semantic Data Widgets der SemwidgJS-Bibliothek erlauben die schnelle und einfache Präsentation und Visualisierung dieser Daten. SemwidgQL ist integraler Bestandteil von SemwidgJS und kann so zur Vereinfachung der Formulierung benötigter Datenabfragen beitragen. Bei Bedarf kann SemwidgED die Nutzer bei der Formulierung dieser Abfragen unterstützen und zusätzlich die Anwendung von SemwidgJS erheblich vereinfachen.

Im Rahmen der einzelnen Untersuchungen des aktuellen Forschungsstands zu den jeweiligen Teilprojekten des Semwidg-Projekts, wurden zahlreiche Eigenschaften unterschiedlicher Systeme identifiziert und betrachtet. Die Zusammenführung der drei Teilprojekte führt dabei zu einem System, welches über eine einmalige Kombination relevanter Eigenschaften verfügt. So ist das Semwidg-System das einzige, das gleichzeitig keine zusätzliche Serversoftware benötigt, ein Widget-basiertes Verfahren zur Darstellung von Daten aus SPARQL-Endpoints bereitstellt, welches sowohl die Datenpräsentation mittels üblicher Web-UI-Elemente, wie auch die Visualisierung von numerischen Daten in Form von Diagrammen ermöglicht, die Erstellung von interaktiven Webseiten unterstützt, eine einfache Wiederverwendung der genutzten Abfragen und abgerufenen Rohdaten erlaubt, eine vereinfachte und leicht zu erlernende Abfrage-

sprache bereitstellt sowie über einen Editor verfügt, der den Nutzer beim Einsatz des Systems unterstützt.

In Anbetracht dessen kann resümiert werden, dass das Semwidg-Projekt einen relevanten Forschungsbeitrag hinsichtlich der Einbindung, Präsentation und Visualisierung von Linked Open Data in Webseiten leisten kann. Die Arbeit ist mit der Hoffnung verbunden, dass das Semwidg-Projekt die Wiederverwendung der Daten der LOD-Cloud in Zukunft steigern kann, und dass es die Einstiegshürden in den Linked-Data-Bereich senken kann, und ihn somit für neue Nutzer erschließbar macht, sodass die Potenziale, die diese bereitstellt, nutzbar werden.

Weiterhin konnten an die Arbeit anschließende Forschungsfragen identifiziert werden, welche im nachfolgenden Abschnitt beschrieben werden. Gleiches gilt für einige Ideen zur Optimierung von Teilen des Semwidg-Projekts, deren Umsetzung zum Teil weit über die hier untersuchten Forschungsfragen hinausgegangen wären.

5.3 Ausblick

Abschließend erfolgt ein Überblick über sich an diese Arbeit anschließende Forschungsfragen, die in weiterführenden Arbeiten untersucht werden könnten, sowie einige Ideen zur potenziellen Weiterentwicklung des Semwidg-Projekts.

5.3.1 SemwigdQL für weiterer Programmiersprachen

Die Syntax von SemwidgQL wurde vollständig in Form einer EBNF-Regel-basierten Grammatik definiert (vgl. Anhang A). Diese ermöglicht dem Parser-Generator ANTLR (vgl. Parr & Fisher, 2011) automatisch einen Parser für beliebige SemwidgQL-Abfragen in einer Vielzahl von Programmiersprachen zu erzeugen, sofern diese von ANTLR unterstützt werden. Zum Zeitpunkt der Entwicklung unterstützte das aktuelle ANTLR v4 noch kein JavaScript, sodass auf das zu diesem Zeitpunkt bereits veraltete ANTLR v3 zurückgegriffen wurde. Beide Versionen unterscheiden sich ihn ihrer internen Arbeitsweise fundamental voneinander, wodurch sich ihr Funktionsumfang ebenfalls unterscheidet.

In der aktuellen Fassung von SemwidgQL wird eine Abfrage eingelesen und aus dieser ein abstrakter Syntaxbaum erzeugt. Dieser wird dann vom SemwidgQL-zu-SPARQL-Transcompiler abgearbeitet, welcher zusätzlich implementiert werden musste. Im Rahmen dieser Arbeit geschah dies für JavaScript und Java. Mit ANTLR v4 sollte es jedoch möglich sein, mittels zusätzlicher Transformationsregeln und String Templates, diesen Transcompiler automatisch mit generieren zu lassen. Ob sich der komplexe Übersetzungsprozess von SemwidgQL zu SPARQL allerdings wirklich damit abbilden ließe, müsste weiter untersucht werden.

Da JavaScript mittlerweile auch von ANTLR v4 unterstützt wird, scheint es erstrebenswert, die SemwidgQL-Grammatik von der alten zur neuen ANTLR-Version zu migrieren und um entsprechende Transformationsregeln und String-Templates zu erweitern. Damit ließen sich direkt weitere SemwidgQL-Transcompiler für eine Vielzahl zusätzlicher Programmiersprachen erzeugen. Unter anderem für C++, C# und Python. Neue Funktionen von SemwidgQL wären dann ebenfalls direkt für alle unterstützten Programmiersprachen verfügbar, ohne sie für jede Programmiersprache aufwendig integrieren zu müssen.

5.3.2 Portal zur Zugänglichmachung neuer Widgets

Die SemwidgJS-Widget-Bibliothek enthält eine Vielzahl vordefinierter Widget-Klassen. Diese bilden die Grundlage für die Präsentation und Visualisierung der Daten der LOD-Cloud. Der Einsatz von SemwidgJS im Rahmen des DESUMA-Projekts (vgl. Abschnitt 3.4) zeigte bereits, dass ein spezifischer Kontext auch spezifische Widgets erfordert. Ebenso existieren zahlreiche Verfahren zur Datenvisualisierung, die durch die Widget-Bibliothek noch nicht abgedeckt werden und ohne die Hilfe weiterer Entwickler nicht implementiert werden können. Um diesen weiteren Entwicklern eine Möglichkeit zur Verbreitung ihrer Widgets zu bieten, könnte auf der Semwidg-Projektseite ein Portal zum Teilen von zusätzlichen Widgets entstehen.

5.3.3 Optimierung von SemwidgED

Die durchgeführte Nutzerstudie zur Evaluation der Gebrauchstauglichkeit lieferte zwar gute Ergebnisse hinsichtlich dieser, doch besteht speziell für Nutzer aus der Gruppe der Anfänger ein gewisses Optimierungspotenzial. So könnten diese beispielsweise stärker durch den Konfigurations- und Widget-Einbindungsprozess geführt werden. Ebenso würden diese, und auch

alle anderen Nutzer, von integrierten Hilfetexten profitieren, die momentan noch vollständig fehlen.

5.3.4 Grafische Eingabe von SemwidgQL-Abfragen

Neben der formular- und textbasierten Eingabe von SemwidgQL-Abfragen bietet sich weiterhin eine grafische Eingabe an, die über die reine Darstellung der Abfrage als (Sub-)Graph hinausgeht, welche bereits in Abschnitt 4.1.2 diskutiert und für ungeeignet befunden wurde. Auf einer höheren Abstraktionsebene lassen sich gegebenenfalls dennoch verständliche Abfragen darstellen (vgl. Haag et al., 2014).

Eine solche Darstellungsart könnte es Nutzern erleichtern, die Abfrage und deren einzelnen Elemente besser zu verstehen. Beispielsweise ließen sich verschiedene Abfragekomponenten grafisch hervorheben oder ausblenden. Speziell längere Abfragen, die in textueller Form schnell unübersichtlich werden, könnten von einer solchen Darstellungsart profitieren. Ob und wie eine grafische Eingabe von SemwidgQL-Abfragen das Verständnis der Nutzer für eben jene verbessern könnte, wäre zu untersuchen.

5.3.5 Ausführliche Evaluation der Komplexitätsmetrik

Auch wenn die im Rahmen der Nutzerstudie zum Vergleich von SemwidgQL und SPARQL entstandene Komplexitätsmetrik nur ein Nebenprodukt der eigentlichen Arbeit darstellt (vgl. Abschnitt 2.4.2), weist sie doch ein erhebliches Potenzial zur weiteren Nutzung durch andere Forscher auf. Bisher liegt der Fokus beim Vergleich verschiedener Abfragesprachen in der Regel auf der Untersuchung der Ausführungsgeschwindigkeit der verarbeitenden Systeme. Durch die hier entwickelte Komplexitätsmetrik könnte der Fokus stärker auf den Nutzer gelenkt werden, der diese Abfragen formulieren muss.

Um eine stärkere Verbreitung zu finden, müsste die Komplexitätsmetrik jedoch genauer evaluiert werden. Dazu bedarf es einer umfangreichen Datenbasis über eine umfassende Menge von verschiedenen Abfragen. Speziell die Gewichtung der möglichen Knotentypen ließe sich dadurch noch verfeinert. Weiterhin wäre zu untersuchen, ob die Komplexität tatsächlich linear mit der Länge einer Abfrage ansteigt oder ob sich die Länge noch auf andere Weise auf die Komplexität auswirkt.

Literaturverzeichnis

Ambrus, O., Möller, K. & Handschuh, S. (2010). *Konduit VQB: a Visual Query Builder for SPARQL on the Social Semantic Desktop*. In: *Workshop on Visual Interfaces to the Social and Semantic Web*. [Seite 125 und 126]

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J. & Vrgoč, D. (2017). *Foundations of Modern Query Languages for Graph Databases*. ACM Computing Surveys (CSUR), 50(5):68:1–68:40. [Seite 21 und 37]

Anicic, D., Fodor, P., Rudolph, S. & Stojanovic, N. (2011). *EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning*. In: *Proceedings of the 20th International Conference on World Wide Web*, WWW 2011, S. 635–644, New York, NY, USA. ACM. [Seite 26]

Artignan, G. & Hascoët, M. (2010). *STOOG: Style-Sheets-Based Toolkit for Graph Visualization*. Technischer Bericht 10009, Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier. [Seite 81]

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R. & Ives, Z. G. (2007). *DBpedia: A Nucleus for a Web of Open Data*. In: Aberer, K., Choi, K., Noy, N. F., Allemang, D., Lee, K., Nixon, L. J. B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G. & Cudré-Mauroux, P. (Hrsg.), *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, Bd. 4825 d. Reihe *Lecture Notes in Computer Science*, S. 722–735. Springer. [Seite 125]

Auer, S., Doehring, R. & Dietzold, S. (2010). *LESS - Template-Based Syndication and Presentation of Linked Data*. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L. & Tudorache, T. (Hrsg.), *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part II*, Bd. 6089 d. Reihe *Lecture Notes in Computer Science*, S. 211–224. Springer. [Seite 12, 82, 119 und 121]

Bailey, J., Bry, F., Furche, T. & Schaffert, S. (2009). *Semantic Web Query Languages*. In: Liu, L. & Özsu, M. T. (Hrsg.), *Encyclopedia of Database Systems*, S. 2583–2586, Boston, MA. Springer US. [Seite 21]

Bangor, A., Kortum, P. & Miller, J. (2009). *Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale*. Journal of Usability Studies, 4(3):114–123. [Seite 148 und 149]

Barbieri, D. F., Braga, D., Ceri, S., Della Valle, E. & Grossniklaus, M. (2009). *C-SPARQL: SPARQL for Continuous Querying*. In: *Proceedings of the 18th International Conference on World Wide Web*, WWW 2009, S. 1061–1062, New York, NY, USA. ACM. [Seite 26]

Battle, S., Wood, D., Leigh, J. & Ruth, L. (2012). *The Callimachus Project: RDFa as a Web Template Language*. In: *Proceedings of the Third International Conference on Consuming Linked Data - Volume 905*, COLD'12, S. 1–14, Aachen, Germany, Germany. CEUR-WS.org. [Seite 82]

Berners-Lee, T., Chen, Y., Chilton, L., Connolly, D., Dhanaraj, R., Hollenbach, J., Lerer, A. & Sheets, D. (2006). *Tabulator: Exploring and analyzing linked data on the semantic web*. In: *Proceedings of the 3rd international semantic web user interaction workshop*, Bd. 2006, S. 159. [Seite 9]

Berners-Lee, T., Hendler, J. & Lassila, O. (2001). *The Semantic Web*. Scientific American, 284(5):34–43. [Seite 1]

Bolles, A., Grawunder, M. & Jacobi, J. (2008). *Streaming SPARQL - Extending SPARQL to Process Data Streams*. In: Bechhofer, S., Hauswirth, M., Hoffmann, J. & Koubarakis, M. (Hrsg.), *The Semantic Web: Research and Applications*, S. 448–462, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 26]

Bonifati, A., Martens, W. & Timm, T. (2017). *An Analytical Study of Large SPARQL Query Logs*. Proceedings of the VLDB Endowment, 11(2):149–161. [Seite 68]

Broekstra, J. & Kampman, A. (2003). *SeRQL : A Second Generation RDF Query Language*. In: *SWAD-Europe Workshop on Semantic Web Storage and Retrieval*. [Seite 23]

Broekstra, J., Kampman, A. & van Harmelen, F. (2002). *Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema*. In: Horrocks, I. & Hendler, J. (Hrsg.), *The Semantic Web — ISWC 2002*, S. 54–68, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 23]

Brooke, J. (1996). *SUS: A »quick and dirty« usability scale*. In: Jordan, P. W., Weerdmeester, B. A., Thomas, B. & McClelland, A. L. (Hrsg.), *Usability Evaluation in Industry*, London. Taylor and Francis. [Seite 142 und 257]

Brophy, M. (2010). *OWL-PL: A presentation language for displaying semantic data on the web*. Masterarbeit, Lehigh University. [Seite 82]

Bry, F. & Schaffert, S. (2002). *The XML Query Language Xcerpt: Design Principles, Examples, and Semantics*. In: Chaudhri, A. B., Jeckle, M., Rahm, E. & Unland, R. (Hrsg.), *Web, Web-Services, and Database Systems*, S. 295–310, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 23]

Brünken, R., Plass, J. L. & Leutner, D. (2003). *Direct Measurement of Cognitive Load in Multimedia Learning*. Educational Psychologist, 38(1):53–61. [Seite 50]

Calbimonte, J., Jeung, H., Corcho, Ó. & Aberer, K. (2012). *Enabling Query Technologies for the Semantic Sensor Web*. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1):43–63. [Seite 26]

Casterella, G. I. & Vijayasarathy, L. (2013). *An experimental investigation of complexity in database query formulation tasks*. Journal of Information Systems Education, 24(3):211. [Seite 51]

Chamberlin, D., Robie, J. & Florescu, D. (2001). *Quilt: An XML Query Language for Heterogeneous Data Sources*. In: Goos, G., Hartmanis, J., van Leeuwen, J., Suciu, D. & Vossen, G. (Hrsg.), *The World Wide Web and Databases*, S. 1–25, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 28]

Champin, P.-A. (2009). *Tal4Rdf: lightweight presentation for the Semantic Web*. In: Auer, S., Bizer, C. & Grimnes, G. A. (Hrsg.), *Proceedings of 5th Workshop on Scripting and Development for the Semantic Web at ESWC 2009, Heraklion, Greece*, Bd. 449 d. Reihe *CEUR Workshop Proceedings*, Aachen. [Seite 82]

Chan, H., Siau, K. & Wei, K.-K. (1997). *The Effect of Data Model, System and Task Characteristics on User Query Performance: An Empirical Study*. SIGMIS Database, 29(1):31–49. [Seite 48 und 49]

Clemmer, A. & Davies, S. (2011). *Smeagol: A »Specific-to-General« Semantic Web Query Interface Paradigm for Novices*. In: Hameurlain, A., Liddle, S. W., Schewe, K.-D. & Zhou, X. (Hrsg.), *Database and Expert Systems Applications. DEXA 2011.*, S. 288–302. Springer, Berlin, Heidelberg. [Seite 126]

Codd, E. F. (1979). *Extending the Database Relational Model to Capture More Meaning*. ACM Transactions on Database Systems, 4(4):397–434. [Seite 24]

Cohen-Boulakia, S., Froidevaux, C. & Pietriga, E. (2006). *Selecting biological data sources and tools with XPR, a path language for RDF*. Pacific Symposium on Biocomputing, S. 116—–127. [Seite 28]

Cohen, J. (1992). *A power primer*. Psychological bulletin, 112(1):155–159. [Seite 57 und 144]

Corby, O. & Faron-Zucker, C. (2015). *STTL: A SPARQL-based Transformation Language for RDF*. In: *11th International Conference on Web Information Systems and Technologies*, Lisbon, Portugal. [Seite 85]

Costabello, L. & Gandon, F. (2014). *Adaptive Presentation of Linked Data on Mobile*. In: *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, S. 243–244, New York, NY, USA. ACM. [Seite 9 und 84]

Deligiannidis, L., Kochut, K. J. & Sheth, A. P. (2007). *RDF data exploration and visualization*. In: *Proceedings of the ACM first workshop on CyberInfrastructure: information management in eScience*, S. 39–46. [Seite 10]

Diaz, G., Arenas, M. & Benedikt, M. (2016). *SPARQLByE: Querying RDF Data by Example*. Proceedings of the VLDB Endowment, 9(13):1533–1536. [Seite 21]

Dzida, W., Hofmann, B., Freitag, R., Redtenbacher, W., Baggen, R., Geis, T., Beimel, J., Zurheiden, C., Hampe-Neteler, W., Hartwig, R. & Peters, H. (2000). *Gebrauchstauglichkeit von Software – ErgoNorm: Ein Verfahren zur Konformitätsprüfung von Software auf der Grundlage von DIN EN ISO 9241 Teile 10 und 11*. In: *Schriftenreihe der Bundesanstalt für Arbeitsschutz und Arbeitsmedizin, Forschung F 1693*, Bremerhaven. Verlag für neue Wissenschaft. [Seite 143 und 258]

Ermilov, I., Martin, M., Lehmann, J. & Auer, S. (2013). *Linked Open Data Statistics: Collection and Exploitation*. In: *Proceedings of the 4th Conference on Knowledge Engineering and Semantic Web*. [Seite 1]

Fafalios, P., Yannakis, T. & Tzitzikas, Y. (2016). *Querying the Web of Data with SPARQL-LD*. In: Fuhr, N., Kovács, L., Risse, T. & Nejdl, W. (Hrsg.), *Research and Advanced Technology for Digital Libraries*, S. 175–187, Cham. Springer International Publishing. [Seite 26]

Fionda, V., Gutierrez, C. & Pirró, G. (2012). *Semantic Navigation on the Web of Data: Specification of Routes, Web Fragments and Actions*. In: *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, S. 281–290, New York, NY, USA. ACM. [Seite 28]

Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P. & Taylor, A. (2018). *Cypher: An Evolving Query Language for Property Graphs*. In: *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, S. 1433–1445, New York, NY, USA. ACM. [Seite 23]

Fuhr, N. & Großjohann, K. (2001). *XIRQL: A Query Language for Information Retrieval in XML Documents*. In: *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, S. 172–180, New York, NY, USA. ACM. [Seite 32]

Furche, T., Linse, B., Bry, F., Plexousakis, D. & Gottlob, G. (2006). *RDF Querying: Language Constructs and Evaluation Methods Compared*. In: Barahona, P., Bry, F., Franconi, E., Henze, N. & Sattler, U. (Hrsg.), *Reasoning Web: Second International Summer School 2006, Lisbon, Portugal, September 4-8, 2006, Tutorial Lectures*, S. 1–52. Springer Berlin Heidelberg. [Seite 21 und 23]

Gassert, H. & Harth, A. (2005). *From graph to GUI: Displaying RDF data from the web with Arago*. In: *Second European Semantic Web Conference/Scripting for the Semantic Web Workshop*. Citeseer. [Seite 9 und 84]

Ghanem, T. M. (2017). *On Teaching Big Data Query Languages*. [Seite 21]

Haag, F., Krüger, R. & Ertl, T. (2016). *VESPa: A Pattern-based Visual Query Language for Event Sequences*. In: *Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 2: IVAPP, (VISIGRAPP 2016)*, S. 48–59. INSTICC, SciTePress. [Seite 126]

Haag, F., Lohmann, S. & Ertl, T. (2014). *SparqlFilterFlow: SPARQL Query Composition for Everyone*. In: Presutti, V., Blomqvist, E., Troncy, R., Sack, H., Papadakis, I. & Tordai, A. (Hrsg.), *The Semantic Web: ESWC 2014 Satellite Events*, S. 362–367, Cham. Springer International Publishing. [Seite 163]

Haag, F., Lohmann, S., Siek, S. & Ertl, T. (2015). *QueryVOWL: A Visual Query Notation for Linked Data*. In: Gandon, F., Guéret, C., Villata, S., Breslin, J., Faron-Zucker, C. & Zimmermann, A. (Hrsg.), *The Semantic Web: ESWC 2015 Satellite Events*, S. 387–402, Cham. Springer International Publishing. [Seite 12, 21 und 126]

Hair, J. F., Ringle, C. M. & Sarstedt, M. (2011). *PLS-SEM: Indeed a silver bullet*. Journal of Marketing theory and Practice, 19(2):139–152. [Seite 63]

Halstead, M. H. (1977). *Elements of software science*, Bd. 7. Elsevier New York. [Seite 49, 51, 62 und 73]

Han, X., Feng, Z., Zhang, X., Wang, X., Rao, G. & Jiang, S. (2016). *On the statistical analysis of practical SPARQL queries*. In: *Proceedings of the 19th International Workshop on Web and Databases*, S. 2. ACM. [Seite 68]

Hartig, O. & Pérez, J. (2015). *LDQL: A Query Language for the Web of Linked Data*. In: Arenas, M., Corcho, Ó., Simperl, E., Strohmaier, M., d'Aquin, M., Srinivas, K., Groth, P., Dumontier, M., Heflin, J., Thirunarayan, K., Thirunarayan, K. & Staab, S. (Hrsg.), *The Semantic Web - ISWC 2015*, S. 73–91, Cham. Springer International Publishing. [Seite 23]

Hastrup, T., Cyganiak, R. & Bojars, U. (2008). *Browsing Linked Data with Fenfire*. In: *LDOW*. [Seite 10]

Heim, P., Ertl, T. & Ziegler, J. (2010). *Facet Graphs: Complex Semantic Querying Made Easy*. In: *Proceedings of the 7th Extended Semantic Web Conference (ESWC 2010)*, Bd. 6088 d. Reihe *LNCS*, S. 288–302, Berlin/Heidelberg. Springer. [Seite 10 und 125]

Heim, P., Hellmann, S., Lehmann, J., Lohmann, S. & Stegemann, T. (2009). *RelFinder: Revealing Relationships in RDF Knowledge Bases*. In: *Proceedings of the 4th International Conference on Semantic and Digital Media Technologies (SAMT 2009)*, S. 182–187, Berlin/Heidelberg. Springer. [Seite 10, 11, 34, 125 und 135]

Hogenboom, F., Milea, V., Frasincar, F. & Kaymak, U. (2010). *RDF-GL: A SPARQL-Based Graphical Query Language for RDF*. In: Chbeir, R., Badr, Y., Abraham, A. & Hassanien, A.-E. (Hrsg.), *Emergent Web Intelligence: Advanced Information Retrieval*, S. 87–116, London. Springer London. [Seite 126]

Hollenbach, J. D. (2010). *A Widget Library for creating Policy-Aware Semantic Web Applications*. Masterarbeit, Massachusetts Institute of Technology. [Seite 86]

Horn, A. (1951). *On sentences which are true of direct unions of algebras*. Journal of Symbolic Logic, 16(1):14–21. [Seite 23]

Horrocks, I. (2002). *DAML+OIL: A Description Logic for the Semantic Web*. Bulletin of the Technical Committee on Data Engineering, 25(1):4–9. [Seite 23]

Huynh, D., Mazzocchi, S. & Karger, D. (2005). *Piggy Bank: Experience the Semantic Web Inside Your Web Browser*. In: Gil, Y., Motta, E., Benjamins, V. R. & Musen, M. A. (Hrsg.), *The Semantic Web – ISWC 2005*, S. 413–430, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 28]

Huynh, D. F., Karger, D. R. & Miller, R. C. (2007). *Exhibit: Lightweight Structured Data Publishing*. In: *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, S. 737–746, New York, NY, USA. ACM. [Seite 86 und 121]

Ikkala, E., Hyvönen, E., Rantala, H. & Koho, M. (2021). *Sampo-UI: A Full Stack JavaScript Framework for Developing Semantic Portal User Interfaces*. Semantic Web – Interoperability, Usability, Applicability. [Seite 125]

Karger, D. R., Ostler, S. & Lee, R. (2009). *The Web Page As a WYSIWYG End-user Customizable Database-backed Information Management Application.* In: *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, S. 257–260, New York, NY, USA. ACM. [Seite 119, 120 und 141]

Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D. & Scholl, M. (2002). *RQL: A Declarative Query Language for RDF*. In: *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, S. 592–603, New York, NY, USA. ACM. [Seite 23]

Kaufmann, E. & Bernstein, A. (2007). *How Useful Are Natural Language Interfaces to the Semantic Web for Casual End-Users?*. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G. & Cudré-Mauroux, P. (Hrsg.), *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, S. 281–294. Springer, Berlin, Heidelberg. [Seite 126]

Kifer, M., Lausen, G. & Wu, J. (1995). *Logical Foundations of Object-oriented and Frame-based Languages*. Journal of the ACM (JACM), 42(4):741–843. [Seite 23]

Kleene, S. C. (1956). *Representation of events in nerve nets and finite automata*. In: Ashby, W., Shannon, C. & McCarthy, J. (Hrsg.), *Automata Studies: Annals of Mathematics Studies. Number 34*, S. 3–41. Princeton University Press. [Seite 24 und 27]

Kobilarov, G., Scott, T., Raimond, Y., Oliver, S., Sizemore, C., Smethurst, M., Bizer, C. & Lee, R. (2009). *Media Meets Semantic Web – How the BBC Uses DBpedia and Linked Data to Make Connections*. In: Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M. & Simperl, E. (Hrsg.), *The Semantic Web: Research and Applications*, S. 723–737, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 1]

Koch, J. & Franz, T. (2008). *LENA - Browsing RDF Data More Complex Than Foaf*. In: Bizer, C. & Joshi, A. (Hrsg.), *Proceedings of the Poster and Demonstration Session at the 7th International Semantic Web Conference (ISWC2008), Karlsruhe, Germany, October 28, 2008*, Bd. 401 d. Reihe *CEUR Workshop Proceedings*. CEUR-WS.org. [Seite 9 und 28]

Lassila, M., Junkkari, M. & Kekäläinen, J. (2015). *Comparison of two XML query languages from the perspective of learners*. Journal of Information Science, 41(5):584–595. [Seite 47, 49, 51 und 57]

Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J. & Hauswirth, M. (2011). *A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data*. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N. & Blomqvist, E. (Hrsg.), *The Semantic Web – ISWC 2011*, S. 370–388, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 26]

Leinberger, M., Scheglmann, S., Lämmel, R., Staab, S., Thimm, M. & Viegas, E. (2014). *Semantic Web Application Development with LITEQ*. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C. A., Vrandecic, D., Groth, P. T., Noy, N. F., Janowicz, K. & Goble, C. A. (Hrsg.), *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II*, Bd. 8797 d. Reihe *Lecture Notes in Computer Science*, S. 212–227. Springer. [Seite 51]

Levenshtein, V. I. (1966). *Binary codes capable of correcting deletions, insertions, and reversals*. In: *Soviet physics doklady*, Bd. 10, S. 707–710. [Seite 137]

Lewis, J. R. (1991). *Psychometric evaluation of an after-scenario questionnaire for computer usability studies: the ASQ*. ACM Sigchi Bulletin, 23(1):78–81. [Seite 142 und 256]

Lewis, J. R. (1995). *IBM computer usability satisfaction questionnaires: Psychometric evaluation and instructions for use*. International Journal of Human–Computer Interaction, 7(1):57–78. [Seite 146]

Lewis, J. R. & Sauro, J. (2009). *The Factor Structure of the System Usability Scale*. In: Kurosu, M. (Hrsg.), *Human Centered Design: First International Conference, HCD 2009, Held as Part of HCI International 2009, San Diego, CA, USA, July 19-24, 2009 Proceedings*, Bd. 5619 d. Reihe *Lecture Notes in Computer Science*, S. 94–103, Berlin, Heidelberg. Springer. [Seite 142]

Lochovsky, F. H. & Tsichritzis, D. C. (1977). *User Performance Considerations in DBMS Selection*. In: *Proceedings of the 1977 ACM SIGMOD International Conference on Management of Data*, SIGMOD '77, S. 128–134, New York, NY, USA. ACM. [Seite 47 und 49]

Lohmann, S., Negru, S., Haag, F. & Ertl, T. (2016). *Visualizing Ontologies with VOWL*. Semantic Web, 7(4):399–419. [Seite 126]

Lopez, V. & Motta, E. (2004). *Ontology-Driven Question Answering in AquaLog*. In: Meziane, F. & Métais, E. (Hrsg.), *Natural Language Processing and Information Systems*, S. 89–102, Berlin, Heidelberg. Springer. [Seite 21]

Luggen, M., Gschwend, A., Bernhard, A. & Cudre-Mauroux, P. (2015). *Uduvudu: a Graph-Aware and Adaptive UI Engine for Linked Data*. In: Bizer, C., Auer, S., Berners-Lee, T. & Heath, T. (Hrsg.), *Proceedings of the Linked Data on the Web Workshop (LDOW2015)*, Nr. 1409 in *CEUR Workshop Proceedings*, Aachen. [Seite 119 und 123]

Martin, E. & Roberts, K. H. (1966). *Grammatical factors in sentence retention*. Journal of Verbal Learning and Verbal Behavior, 5(3):211–218. [Seite 51]

McBride, B. (2002). *Jena: A Semantic Web Toolkit*. IEEE Internet Computing, 6(6):55–59. [Seite 24]

McCabe, T. J. (1976). *A complexity measure*. IEEE Transactions on Software Engineering, 2(4):308–320. [Seite 51]

Mendelzon, A. O. & Wood, P. T. (1995). *Finding Regular Simple Paths in Graph Databases*. SIAM Journal on Computing, 24(6):1235–1258. [Seite 27]

Mika, P. & Potter, T. (2012). *Metadata Statistics for a Large Web Corpus*. In: Bizer, C., Heath, T., Berners-Lee, T. & Hausenblas, M. (Hrsg.), *WWW2012 Workshop on Linked Data on the Web, Lyon, France, 16 April, 2012*, Bd. 937 d. Reihe *CEUR Workshop Proceedings*. CEUR-WS.org. [Seite 1]

Miller, G. A. & Chomsky, N. (1963). *Finitary Models of Language Users*. In: Luce, D. (Hrsg.), *Handbook of Mathematical Psychology*, S. 2–419. John Wiley & Sons. [Seite 50]

Miller, L., Seaborne, A. & Reggiori, A. (2002). *Three Implementations of SquishQL, a Simple RDF Query Language*. In: Horrocks, I. & Hendler, J. (Hrsg.), *The Semantic Web — ISWC 2002*, S. 423–435, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 24]

Nikolaou, C., Dogani, K., Bereta, K., Garbis, G., Karpathiotakis, M., Kyzirakos, K. & Koubarakis, M. (2015). *Sextant: Visualizing time-evolving linked geospatial data*. Journal of Web Semantics, 35:35–52. [Seite 10 und 11]

Nowack, B. (2008). *SPARQL+, SPARQLScript, SPARQL Result Templates: SPARQL Extensions for the Mashup Developer*. In: Bizer, C. & Joshi, A. (Hrsg.), *Proceedings of the Poster and Demonstration Session at the 7th International Semantic Web Conference (ISWC2008), Karlsruhe, Germany, October 28, 2008*, Bd. 401 d. Reihe *CEUR Workshop Proceedings*. CEUR-WS.org. [Seite 86]

Nowack, B. (2009). *Paggr: Linked Data widgets and dashboards*. Journal of Web Semantics, 7(4):272–277. [Seite 86]

Olston, C., Reed, B., Srivastava, U., Kumar, R. & Tomkins, A. (2008). *Pig Latin: A Not-so-foreign Language for Data Processing*. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, S. 1099–1110, New York, NY, USA. ACM. [Seite 21]

Parr, T. & Fisher, K. (2011). *LL(*): The Foundation of the ANTLR Parser Generator*. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, S. 425–436, New York, NY, USA. ACM. [Seite 45 und 162]

Pérez, J., Arenas, M. & Gutierrez, C. (2008). *nSPARQL: A Navigational Language for RDF*. In: Sheth, A., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T. & Thirunarayan, K. (Hrsg.), *The Semantic Web - ISWC 2008*, S. 66–81, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 28]

Perry, M., Jain, P. & Sheth, A. P. (2011). *SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries*. In: Ashish, N. & Sheth, A. P. (Hrsg.), *Geospatial Semantics and the Semantic Web: Foundations, Algorithms, and Applications*, S. 61–86, Boston, MA. Springer US. [Seite 26]

Pietriga, E., Bizer, C., Karger, D. R. & Lee, R. (2006). *Fresnel: A Browser-Independent Presentation Vocabulary for RDF*. In: Cruz, I. F., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M. & Aroyo, L. (Hrsg.), *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, Bd. 4273 d. Reihe *Lecture Notes in Computer Science*, S. 158–171. Springer. [Seite 9, 28, 84 und 119]

Przyjaciel-Zablocki, M., Schätzle, A., Hornung, T. & Lausen, G. (2012). *RDFPath: Path Query Processing on Large RDF Graphs with MapReduce*. In: García-Castro, R., Fensel, D. & Antoniou, G. (Hrsg.), *The Semantic Web: ESWC 2011 Workshops*, S. 50–64, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 28]

Quan, D., Huynh, D. & Karger, D. R. (2003). *Haystack: A Platform for Authoring End User Semantic Web Applications*. In: Fensel, D., Sycara, K. & Mylopoulos, J. (Hrsg.), *The Semantic Web - ISWC 2003*, S. 738–753, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 8 und 84]

Quan, D. & Karger, D. R. (2004a). *How to Make a Semantic Web Browser*. In: *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, S. 255–265, New York, NY, USA. ACM. [Seite 84]

Quan, D. & Karger, D. R. (2004b). *Xenon: An RDF Stylesheet Ontology*. Veröffentlicht über die »general@simile.mit.edu«-Mailingliste. [Seite 84 und 119]

Reisner, P. (1977). *Use of Psychological Experimentation as an Aid to Development of a Query Language*. IEEE Transactions on Software Engineering, 3(3):218–229. [Seite 46]

Reisner, P. (1981). *Human Factors Studies of Database Query Languages: A Survey and Assessment*. ACM Computing Surveys, 13(1):13–31. [Seite 49]

Reisner, P., Boyce, R. F. & Chamberlin, D. D. (1975). *Human factors evaluation of two data base query languages: SQUARE and SEQUEL*. In: *American Federation of Information Processing Societies: 1975 National Computer Conference, 19-22 May 1975, Anaheim, CA, USA*, Bd. 44 d. Reihe *AFIPS Conference Proceedings*, S. 447–452. AFIPS Press. [Seite 46, 49 und 57]

Reynolds, D. (2002). *RDF-QBE: a Semantic Web building block*. Technical Report HPL-2002-327, HP Labs. [Seite 21]

Rico, M., Camacho, D. & Corcho, Ó. (2008). *VPOET: Using a Distributed Collaborative Platform for Semantic Web Applications*. In: Badica, C., Mangioni, G., Carchiolo, V. & Burdescu, D. D. (Hrsg.), *Intelligent Distributed Computing, Systems and Applications*, S. 167–176, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 81]

Rico, M., Camacho, D. & Corcho, Ó. (2010). *A contribution-based framework for the creation of semantically-enabled web applications*. Information Sciences, 180(10):1850–1864. Special Issue on Intelligent Distributed Information Systems. [Seite 81]

Rietveld, L. & Hoekstra, R. (2013). *YASGUI: Not Just Another SPARQL Client*. In: Cimiano, P., Fernández, M., Lopez, V., Schlobach, S. & Völker, J. (Hrsg.), *The Semantic Web: ESWC 2013 Satellite Events*, S. 78–86, Berlin, Heidelberg. Springer Berlin Heidelberg. [Seite 125 und 126]

Rodriguez, M. A. (2015). *The Gremlin Graph Traversal Machine and Language (Invited Talk)*. In: *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, S. 1–10, New York, NY, USA. ACM. [Seite 21]

Saleem, M., Ali, M. I., Hogan, A., Mehmood, Q. & Ngomo, A. N. (2015). *LSQ: The Linked SPARQL Queries Dataset*. In: Arenas, M., Corcho, Ó., Simperl, E., Strohmaier, M., d'Aquin, M., Srinivas, K., Groth, P. T., Dumontier, M., Heflin, J., Thirunarayan, K. & Staab, S. (Hrsg.), *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, Bd. 9367 d. Reihe *Lecture Notes in Computer Science*, S. 261–269. Springer. [Seite 52, 55 und 68]

Sasayama, S. (2016). *Is a 'Complex' Task Really Complex? Validating the Assumption of Cognitive Task Complexity*. The Modern Language Journal, 100(1):231–254. [Seite 50]

Schaffert, S., Bauer, C., Kurz, T., Dorschel, F., Glachs, D. & Fernandez, M. (2012). *The linked media framework: integrating and interlinking enterprise media content and data*. In: Presutti, V. & Pinto, H. S. (Hrsg.), *I-SEMANTICS 2012 - 8th International Conference on Semantic Systems, I-SEMANTICS '12, Graz, Austria, September 5-7, 2012*, S. 25–32. ACM. [Seite 28]

Schandl, T. & Blumauer, A. (2010). *PoolParty: SKOS Thesaurus Management Utilizing Linked Data*. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L. & Tudorache, T. (Hrsg.), *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part II*, Bd. 6089, S. 421–425. Springer. [Seite 1]

Scheider, S., Degbelo, A., Lemmens, R., van Elzakker, C., Zimmerhof, P., Kostic, N., Jones, J. & Banhatti, G. (2017). *Exploratory querying of SPARQL endpoints in space and time*. Semantic web, 8(1):65–86. [Seite 10 und 13]

Schmachtenberg, M., Bizer, C. & Paulheim, H. (2014). *Adoption of the Linked Data Best Practices in Different Topical Domains*. In: *The Semantic Web – ISWC 2014*, Bd. 8796 d. Reihe *Lecture Notes in Computer Science*, S. 245–260. Springer International Publishing. [Seite 1]

Sequeda, J. F. & Corcho, Ó. (2009). *Linked Stream Data: A Position Paper*. In: *Proceedings of the 2nd International Conference on Semantic Sensor Networks - Volume 522*, SSN'09, S. 148–157, Aachen, Germany, Germany. CEUR-WS.org. [Seite 26]

Shepperd, M. & Ince, D. (1994). *A critique of three metrics*. Journal of Systems and Software, 26:197–210. [Seite 51]

Sintek, M. & Decker, S. (2002). *TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web*. In: *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, S. 364–378, Berlin, Heidelberg. Springer-Verlag. [Seite 23]

Skjæveland, M. G. (2012). *Sgvizler: A JavaScript Wrapper for Easy Visualization of SPARQL Result Sets*. In: *9th Extended Semantic Web Conference (ESWC2012)*. [Seite 12, 86 und 108]

Smart, P. R., Russell, A., Braines, D., Kalfoglou, Y., Bao, J. & Shadbolt, N. R. (2008). *A Visual Approach to Semantic Query Design Using a Web-Based Graphical Query Designer*. In: Gangemi, A. & Euzenat, J. (Hrsg.), *Knowledge Engineering: Practice and Patterns. EKAW 2008*, S. 275–291. Springer, Berlin, Heidelberg. [Seite 126]

Stegemann, T., Hussein, T., Gaulke, W. & Ziegler, J. (2011). *Interacting with semantic data by using X3S*. In: Diaz, P., Hussein, T., Lohmann, S. & Ziegler, J. (Hrsg.), *Proceedings of the 1st Workshop on Data-Centric Interactions on the Web*. [Seite 12 und 84]

Stegemann, T. & Ziegler, J. (2014). *SemwidgJS: A Semantic Widget Library for the Rapid Development of User Interfaces for Linked Open Data*. In: Plödereder, E., Grunske, L., Schneider, E. & Ull, D. (Hrsg.), *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, Bd. 232 d. Reihe *LNI*, S. 479–490. GI. [Seite 4]

Stegemann, T. & Ziegler, J. (2017a). *Investigating Learnability, User Performance, and Preferences of the Path Query Language SemwidgQL Compared to SPARQL*. In: d'Amato, C., Fernandez, M., Tamma, V., Lecue, F., Cudré-Mauroux, P., Sequeda, J., Lange, C. & Heflin, J. (Hrsg.), *The Semantic Web – ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21–25, 2017, Proceedings, Part I*, S. 611–627, Cham. Springer International Publishing. [Seite 46, 68, 77 und 78]

Stegemann, T., Ziegler, J., Hussein, T. & Gaulke, W. (2012). *Interactive Construction of Semantic Widgets for Visualizing Semantic Web Data*. In: *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, S. 157–162, New York, NY, USA. ACM. [Seite 4, 84, 119 und 122]

Stegemann, T. & Ziegler, J. (2017b). *Pattern-Based Analysis of SPARQL Queries from the LSQ Dataset*. In: Nikitina, N., Song, D., Fokoue, A. & Haase, P. (Hrsg.), *ISWC 2017 Posters & Demonstrations and Industry Tracks (ISWC-PD-Industry)*, Nr. 1963 in *CEUR Workshop Proceedings*, Aachen. [Seite 68]

Topi, H., Valacich, J. S. & Hoffer, J. A. (2005). *The effects of task complexity and time availability limitations on human performance in database query tasks*. International Journal of Human-Computer Studies, 62(3):349–379. [Seite 48 und 49]

Tullis, T. S. & Stetson, J. N. (2004). *A Comparison of Questionnaires for Assessing Website Usability*. In: *Proceedings of the Usability Professionals Association (UPA) 2004*, S. 1–12. [Seite 142]

Vardi, M. Y. (1982). *The Complexity of Relational Query Languages (Extended Abstract)*. In: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, S. 137–146, New York, NY, USA. ACM. [Seite 37]

Viegas, F. B., Wattenberg, M., van Ham, F., Kriss, J. & McKeon, M. (2007). *ManyEyes: A Site for Visualization at Internet Scale*. IEEE Transactions on Visualization and Computer Graphics, 13(6):1121–1128. [Seite 2]

Wang, M. D. (1970). *The role of syntactic complexity as a determiner of comprehensibility*. Journal of Verbal Learning and Verbal Behavior, 9(4):398–404. [Seite 50 und 52]

Welty, C. & Stemple, D. W. (1981). *Human Factors Comparison of a Procedural and a Nonprocedural Query Language*. ACM Transactions on Database Systems (TODS), 6(4):626–649. [Seite 47, 49 und 57]

Winkler, W. E. (1990). *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*. In: *Proceedings of the Section on Survey Research Methods*. American Statistical Association. [Seite 137]

Wirth, N. (1996). *ISO/IEC 14977:1996(E), Information technology – Syntactic metalanguage – Extended BNF*. Standard, International Organization for Standardization/International Electrotechnical Commission, Genève, Switzerland. [Seite 37]

Wleklinski, F. (2003). *Suche im Semantic Web: Erweiterung des VRP um eine intuitive und RQL-basierte Anfrageschnittstelle*. Diplomarbeit, Universität Frankfurt am Main. [Seite 23]

Wood, R. E. (1986). *Task complexity: Definition of the construct*. Organizational Behavior and Human Decision Processes, 37(1):60–82. [Seite 50]

Yngve, V. H. (1960). *A model and an hypothesis for language structure*. Proceedings of the American philosophical society, 104(5):444–466. [Seite 51]

Ziegler, J. & Stegemann, T. (2014). *The Role of Semantic Data in Engineering Interactive Systems*. HCI Engineering Charting the Way towards Methods and Tools for Advanced Interactive Systems, S. 43. [Seite 4]

Zloof, M. M. (1975). *Query by Example*. In: *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, AFIPS '75, S. 431–438, New York, NY, USA. ACM. [Seite 12, 21 und 125]

Abbildungsverzeichnis

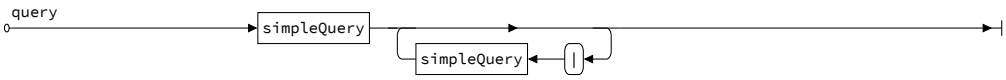
Tabellenverzeichnis

Listingverzeichnis

Diagrammverzeichnis

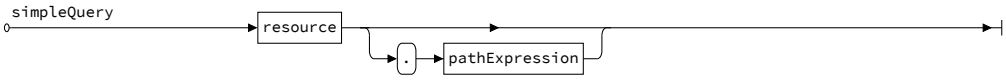
ANHANG A

Formale Spezifikation der SemwidgQL-Syntax



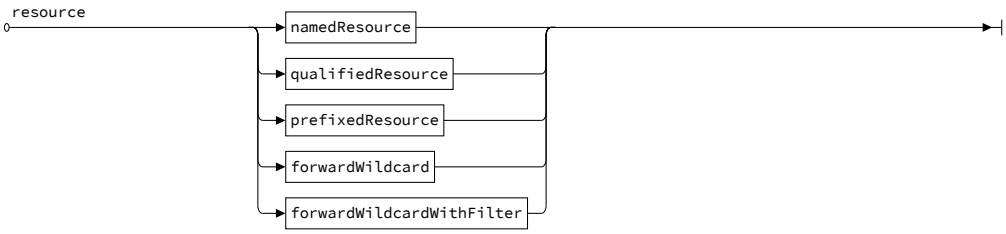

```
query = simpleQuery { '|' simpleQuery } ;
```

Diagramm A.1: Syntaxdiagramm der query-Syntax




```
simpleQuery = resource [ '.' pathExpression ] ;
```

Diagramm A.2: Syntaxdiagramm der simpleQuery-Syntax

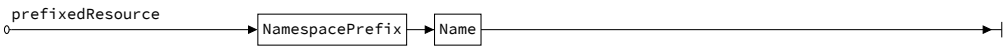



```
resource = namedResource | qualifiedResource | prefixedResource | forwardWildcard | forwardWildcardWithFilter
;
```

Diagramm A.3: Syntaxdiagramm der resource-Syntax


```
namedResource = Name ;
```

Diagramm A.4: Syntaxdiagramm der namedResource-Syntax

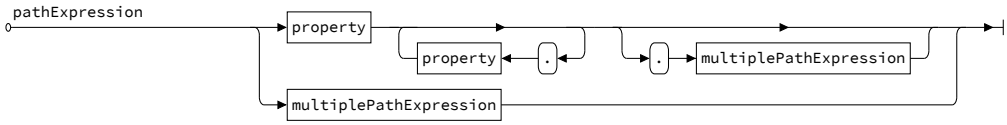



```
prefixedResource = NamespacePrefix Name ;
```

Diagramm A.5: Syntaxdiagramm der prefixedResource-Syntax

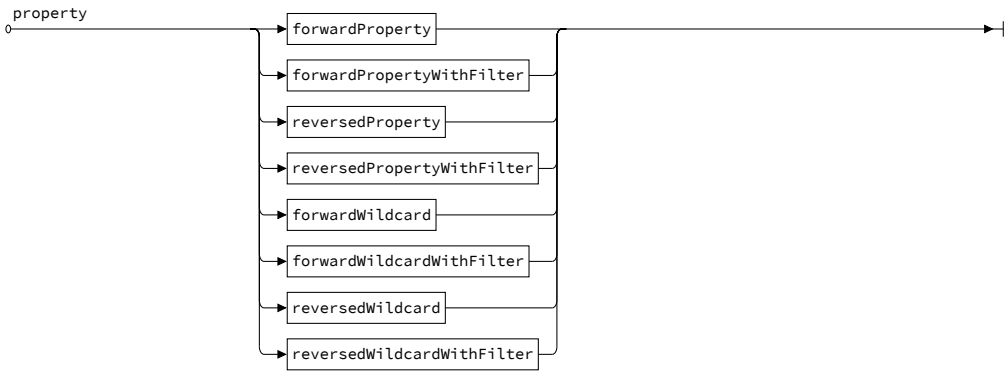

```
qualifiedResource = '<' Iri '>' ;
```

Diagramm A.6: Syntaxdiagramm der qualifiedResource-Syntax



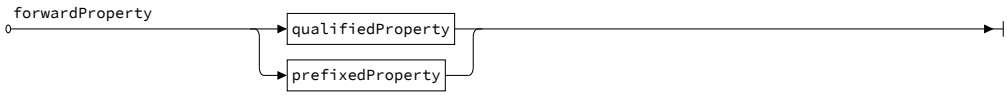

```
pathExpression = property { '.' property } [ '.' multiplePathExpression ] | multiplePathExpression ;
```

Diagramm A.7: Syntaxdiagramm der pathExpression-Syntax



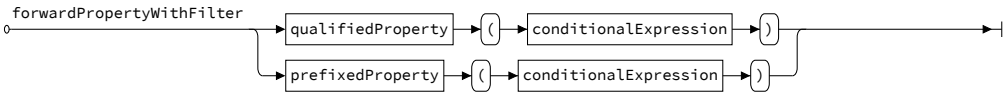

```
property = forwardProperty | forwardPropertyWithFilter | reversedProperty | reversedPropertyWithFilter |
forwardWildcard | forwardWildcardWithFilter | reversedWildcard | reversedWildcardWithFilter ;
```

Diagramm A.8: Syntaxdiagramm der property-Syntax



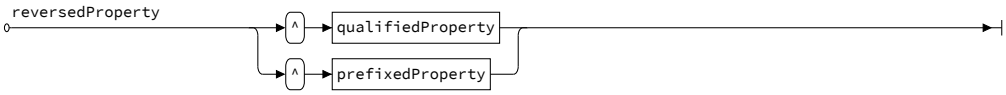

```
forwardProperty = qualifiedProperty | prefixedProperty ;
```

Diagramm A.9: Syntaxdiagramm der forwardProperty-Syntax



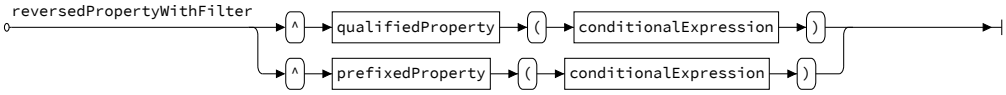

```
forwardPropertyWithFilter = qualifiedProperty '(' conditionalExpression ')' | prefixedProperty '(' conditio-
nalExpression ')' ;
```

Diagramm A.10: Syntaxdiagramm der forwardPropertyWithFilter-Syntax




```
reversedProperty = '^' qualifiedProperty | '^' prefixedProperty ;
```

Diagramm A.11: Syntaxdiagramm der reversedProperty-Syntax

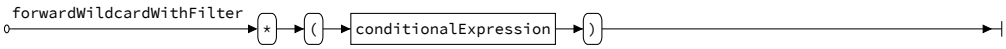



```
reversedPropertyWithFilter = '^' qualifiedProperty '(' conditionalExpression ')' | '^' prefixedProperty '('
conditionalExpression ')' ;
```

Diagramm A.12: Syntaxdiagramm der reversedPropertyWithFilter-Syntax

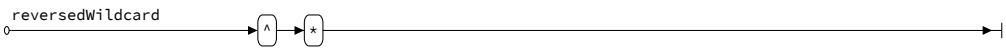

```
forwardWildcard = '*' ;
```

Diagramm A.13: Syntaxdiagramm der forwardWildcard-Syntax



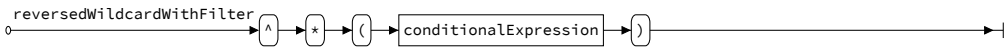

```
forwardWildcardWithFilter = '*' '(' conditionalExpression ')' ;
```

Diagramm A.14: Syntaxdiagramm der forwardWildcardWithFilter-Syntax




```
reversedWildcard = '^' '*' ;
```

Diagramm A.15: Syntaxdiagramm der reversedWildcard-Syntax

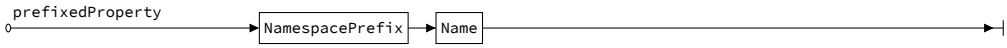



```
reversedWildcardWithFilter = '^' '*' '(' conditionalExpression ')' ;
```

Diagramm A.16: Syntaxdiagramm der reversedWildcardWithFilter-Syntax

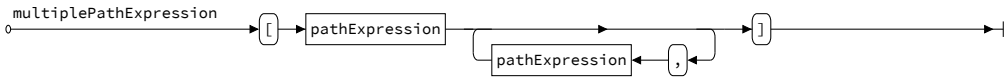

```
qualifiedProperty = '<' Iri '>' ;
```

Diagramm A.17: Syntaxdiagramm der qualifiedProperty-Syntax



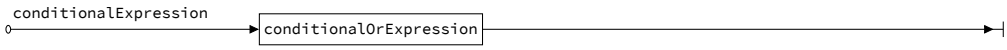

```
prefixedProperty = NamespacePrefix Name ;
```

Diagramm A.18: Syntaxdiagramm der prefixedProperty-Syntax



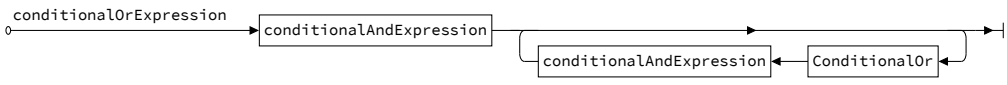

```
multiplePathExpression = '[' pathExpression { ',' pathExpression } ']' ;
```

Diagramm A.19: Syntaxdiagramm der multiplePathExpression-Syntax




```
conditionalExpression = conditionalOrExpression ;
```

Diagramm A.20: Syntaxdiagramm der conditionalExpression-Syntax

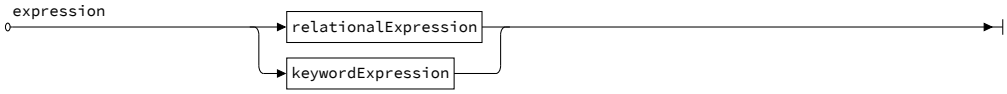



```
conditionalOrExpression = conditionalAndExpression { ConditionalOr conditionalAndExpression } ;
```

Diagramm A.21: Syntaxdiagramm der conditionalOrExpression-Syntax

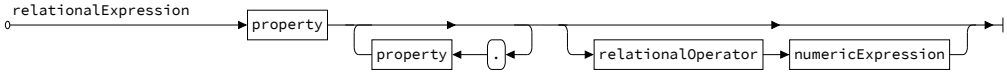

```
conditionalAndExpression = expression { ConditionalAnd expression } ;
```

Diagramm A.22: Syntaxdiagramm der conditionalAndExpression-Syntax




```
expression = relationalExpression | keywordExpression ;
```

Diagramm A.23: Syntaxdiagramm der expression-Syntax

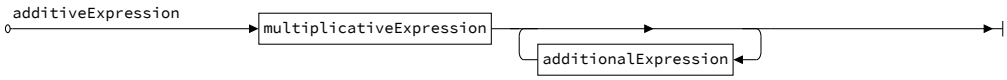



```
relationalExpression = property { '.' property } [ relationalOperator numericExpression ] ;
```

Diagramm A.24: Syntaxdiagramm der relationalExpression-Syntax

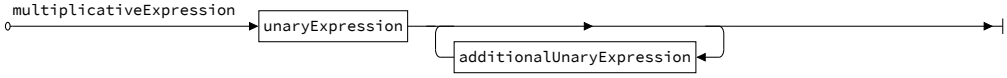

```
numericExpression = additiveExpression ;
```

Diagramm A.25: Syntaxdiagramm der numericExpression-Syntax



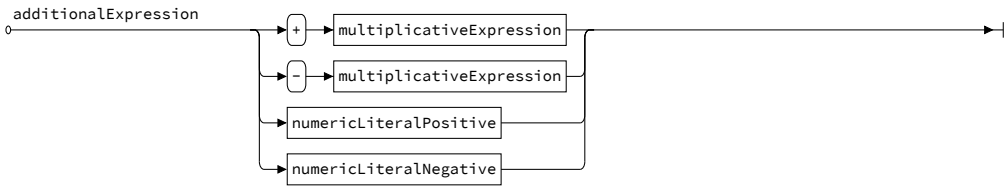

```
additiveExpression = multiplicativeExpression { additionalExpression } ;
```

Diagramm A.26: Syntaxdiagramm der additiveExpression-Syntax



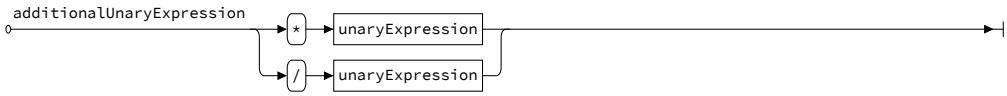

```
multiplicativeExpression = unaryExpression { additionalUnaryExpression } ;
```

Diagramm A.27: Syntaxdiagramm der multiplicativeExpression-Syntax



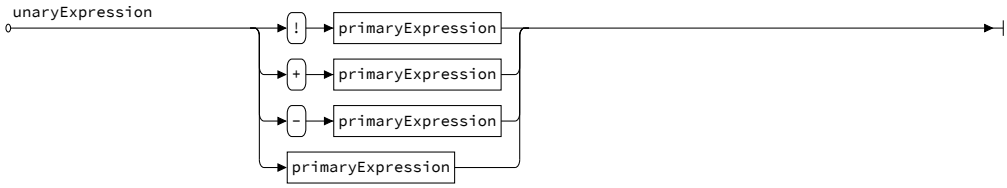

```
additionalExpression = '+' multiplicativeExpression | '-' multiplicativeExpression | numericLiteralPositive
| numericLiteralNegative ;
```

Diagramm A.28: Syntaxdiagramm der additionalExpression-Syntax



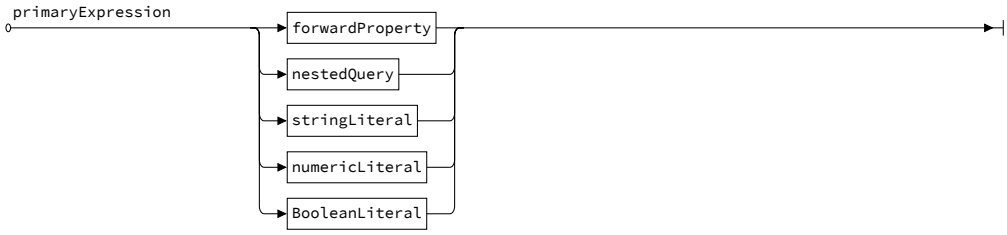

```
additionalUnaryExpression = '*' unaryExpression | '/' unaryExpression ;
```

Diagramm A.29: Syntaxdiagramm der additionalUnaryExpression-Syntax



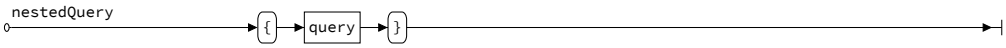

```
unaryExpression = '!' primaryExpression | '+' primaryExpression | '-' primaryExpression | primaryExpression
;
```

Diagramm A.30: Syntaxdiagramm der unaryExpression-Syntax




```
primaryExpression = forwardProperty | nestedQuery | stringLiteral | numericLiteral | BooleanLiteral ;
```

Diagramm A.31: Syntaxdiagramm der primaryExpression-Syntax




```
nestedQuery = '{' query '}' ;
```

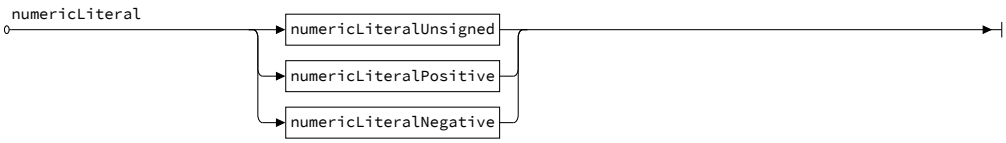
Diagramm A.32: Syntaxdiagramm der nestedQuery-Syntax


```
stringLiteral = StringLiteral1 | StringLiteral2 ;
```

Diagramm A.33: Syntaxdiagramm der stringLiteral-Syntax


```
booleanLiteral = Boolean ;
```

Diagramm A.34: Syntaxdiagramm der booleanLiteral-Syntax

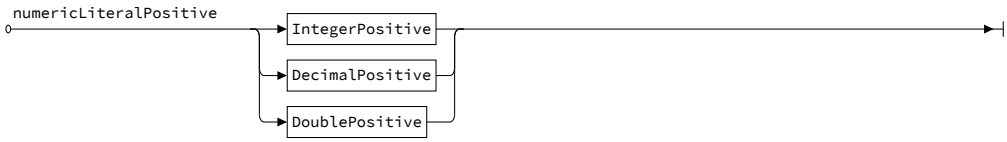



```
numericLiteral = numericLiteralUnsigned | numericLiteralPositive | numericLiteralNegative ;
```

Diagramm A.35: Syntaxdiagramm der numericLiteral-Syntax

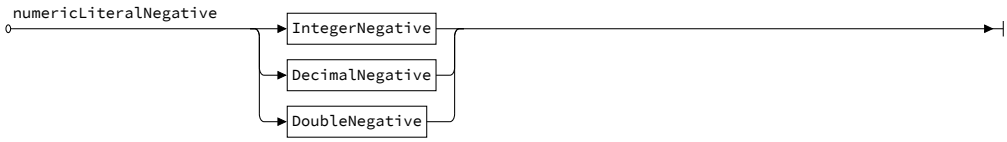

```
numericLiteralUnsigned = Integer | Decimal | DoubleValue ;
```

Diagramm A.36: Syntaxdiagramm der numericLiteralUnsigned-Syntax



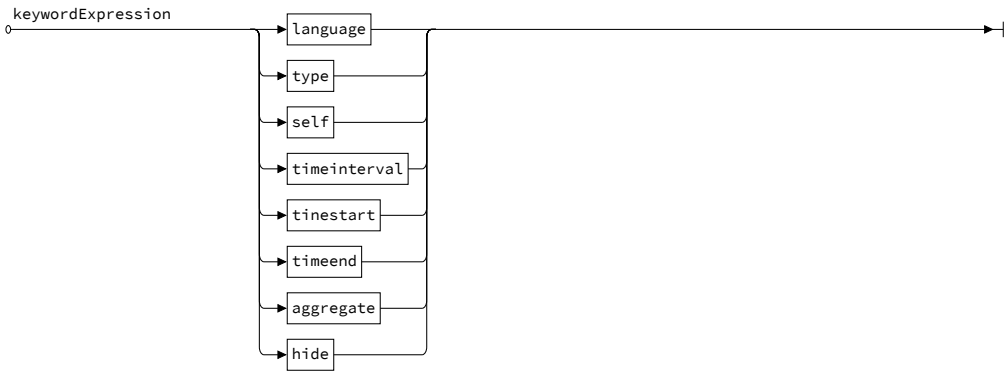

```
numericLiteralPositive = IntegerPositive | DecimalPositive | DoublePositive ;
```

Diagramm A.37: Syntaxdiagramm der numericLiteralPositive-Syntax



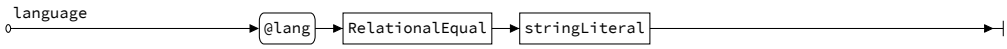

```
numericLiteralNegative = IntegerNegative | DecimalNegative | DoubleNegative ;
```

Diagramm A.38: Syntaxdiagramm der numericLiteralNegative-Syntax



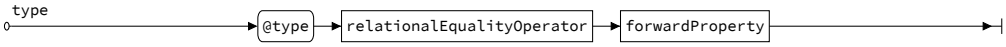

```
keywordExpression = language | type | self | timeinterval | tinestart | timeend | aggregate | hide ;
```

Diagramm A.39: Syntaxdiagramm der keywordExpression-Syntax



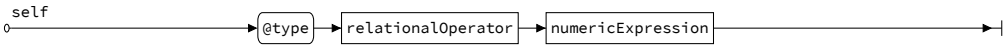

```
language = '@lang' RelationalEqual stringLiteral ;
```

Diagramm A.40: Syntaxdiagramm der language-Syntax



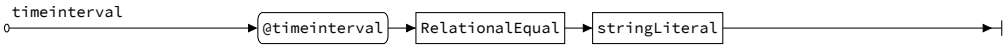

```
type = '@type' relationalEqualityOperator forwardProperty ;
```

Diagramm A.41: Syntaxdiagramm der type-Syntax



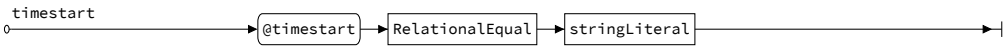

```
self = '@type' relationalOperator numericExpression ;
```

Diagramm A.42: Syntaxdiagramm der self-Syntax



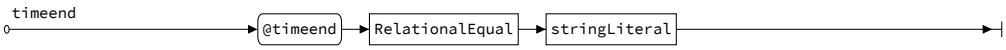

```
timeinterval = '@timeinterval' RelationalEqual stringLiteral ;
```

Diagramm A.43: Syntaxdiagramm der timeinterval-Syntax



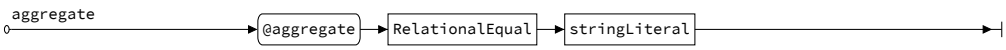

```
timestart = '@timestart' RelationalEqual stringLiteral ;
```

Diagramm A.44: Syntaxdiagramm der timestart-Syntax



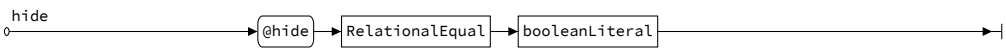

```
timeend = '@timeend' RelationalEqual stringLiteral ;
```

Diagramm A.45: Syntaxdiagramm der timeend-Syntax



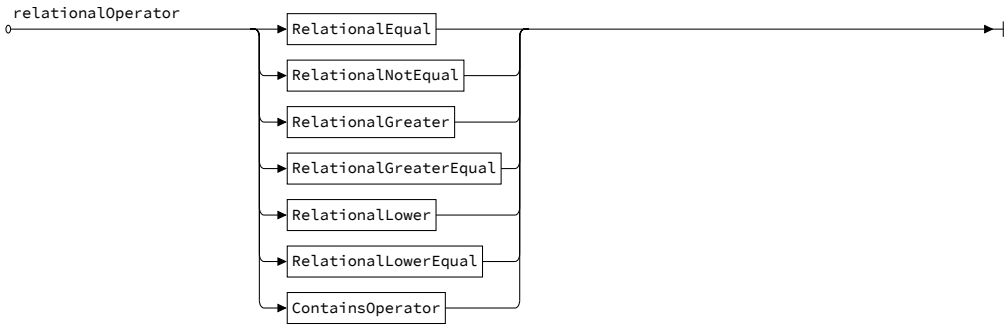

```
aggregate = '@aggregate' RelationalEqual stringLiteral ;
```

Diagramm A.46: Syntaxdiagramm der aggregate-Syntax



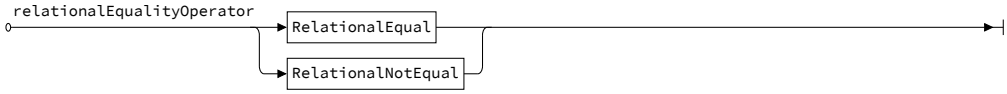

```
hide = '@hide' RelationalEqual booleanLiteral ;
```

Diagramm A.47: Syntaxdiagramm der hide-Syntax



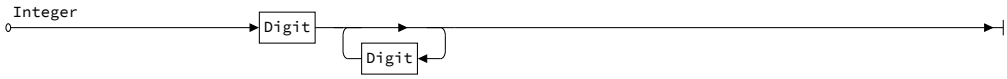

```
relationalOperator = RelationalEqual | RelationalNotEqual | RelationalGreater | RelationalGreaterEqual |
RelationalLower | RelationalLowerEqual | ContainsOperator ;
```

Diagramm A.48: Syntaxdiagramm der relationalOperator-Syntax



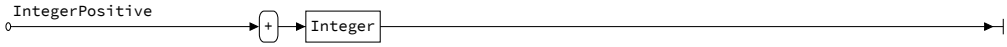

```
relationalEqualityOperator = RelationalEqual | RelationalNotEqual ;
```

Diagramm A.49: Syntaxdiagramm der relationalEqualityOperator-Syntax



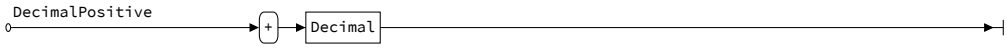

```
Integer = Digit { Digit } ;
```

Diagramm A.50: Syntaxdiagramm der Integer-Syntax




```
IntegerPositive = '+' Integer ;
```

Diagramm A.51: Syntaxdiagramm der IntegerPositive-Syntax

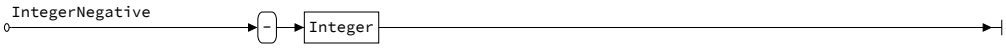



```
DecimalPositive = '+' Decimal ;
```

Diagramm A.52: Syntaxdiagramm der DecimalPositive-Syntax


```
DoublePositive = '+' DoubleValue ;
```

Diagramm A.53: Syntaxdiagramm der DoublePositive-Syntax




```
IntegerNegative = '-' Integer ;
```

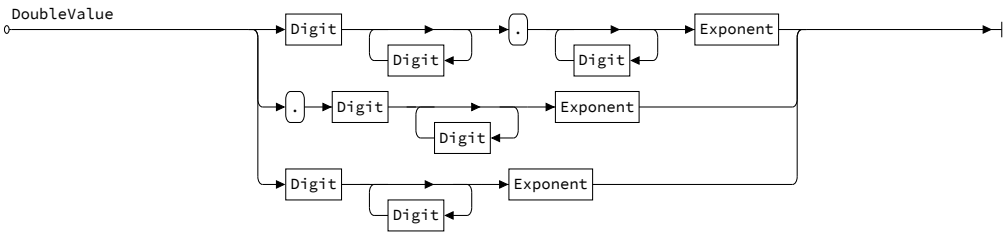
Diagramm A.54: Syntaxdiagramm der IntegerNegative-Syntax


```
DecimalNegative = '-' Decimal ;
```

Diagramm A.55: Syntaxdiagramm der DecimalNegative-Syntax

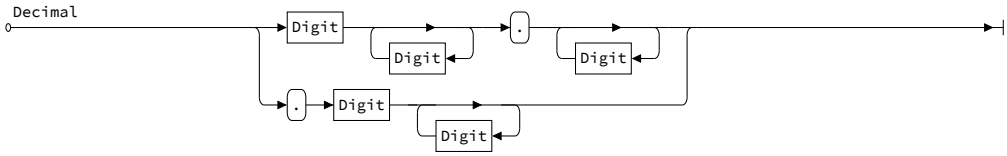

```
DoubleNegative = '-' DoubleValue ;
```

Diagramm A.56: Syntaxdiagramm der DoubleNegative-Syntax



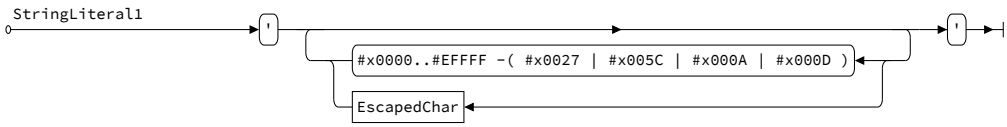

```
DoubleValue = Digit { Digit } '.' {Digit} Exponent | '.' Digit { Digit } Exponent | Digit { Digit } Exponent
;
```

Diagramm A.57: Syntaxdiagramm der DoubleValue-Syntax



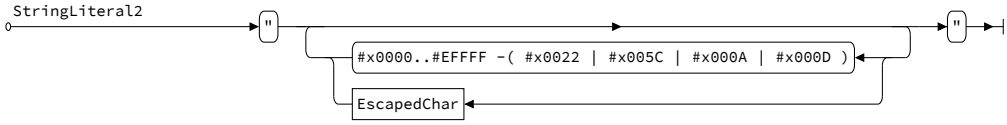

```
Decimal = Digit { Digit } '.' {Digit} | '.' Digit { Digit } ;
```

Diagramm A.58: Syntaxdiagramm der Decimal-Syntax



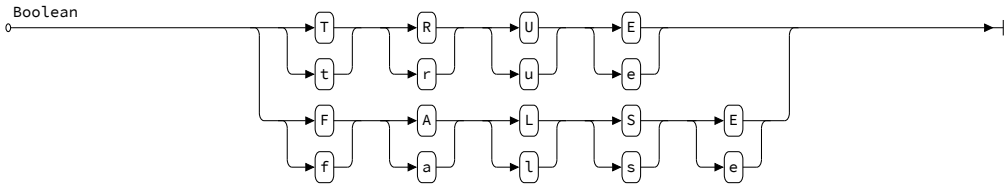

```
StringLiteral1 = "'" { ( "#x0000..#EFFFF -( #x0027 | #x005C | #x000A | #x000D )" ) | EscapedChar } "'" ;
```

Diagramm A.59: Syntaxdiagramm der StringLiteral1-Syntax



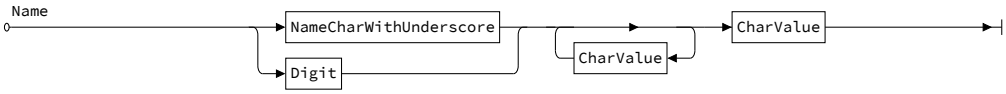

```
StringLiteral2 = '"' { ( "#x0000..#EFFFF -( #x0022 | #x005C | #x000A | #x000D )" ) | EscapedChar } '"' ;
```

Diagramm A.60: Syntaxdiagramm der StringLiteral2-Syntax




```
Boolean = ('T'|'t') ('R'|'r') ('U'|'u') ('E'|'e') | ('F'|'f') ('A'|'a') ('L'|'l') ('S'|'s') ('E'|'e') ;
```

Diagramm A.61: Syntaxdiagramm der Boolean-Syntax

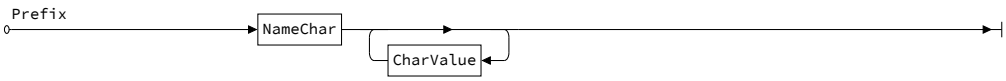



```
Name = ( NameCharWithUnderscore | Digit ) ( { CharValue } CharValue ) ;
```

Diagramm A.62: Syntaxdiagramm der Name-Syntax

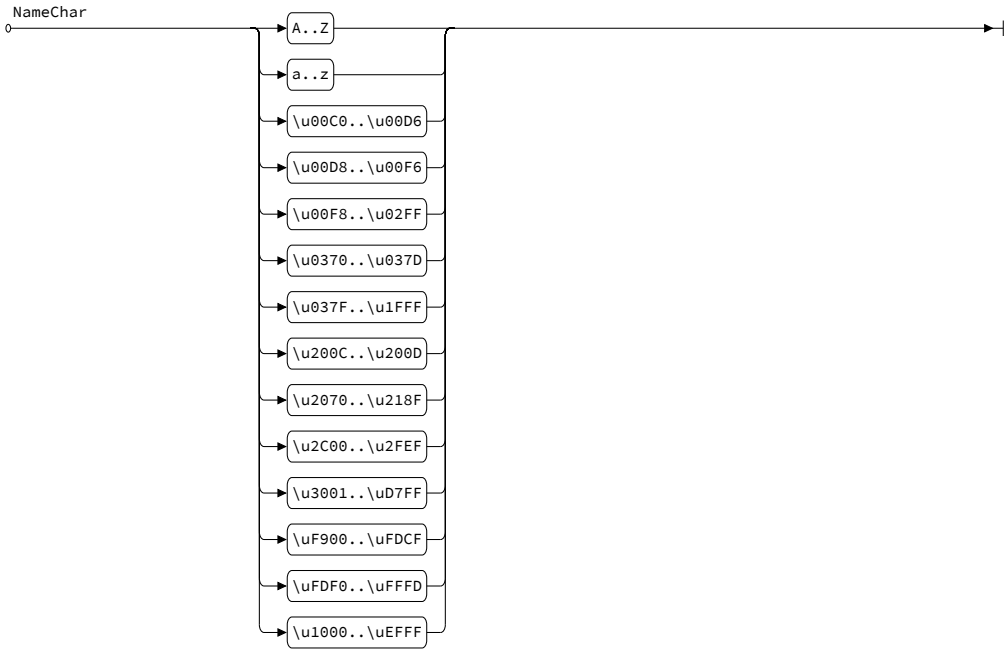

```
NamespacePrefix = Prefix ':' ;
```

Diagramm A.63: Syntaxdiagramm der NamespacePrefix-Syntax



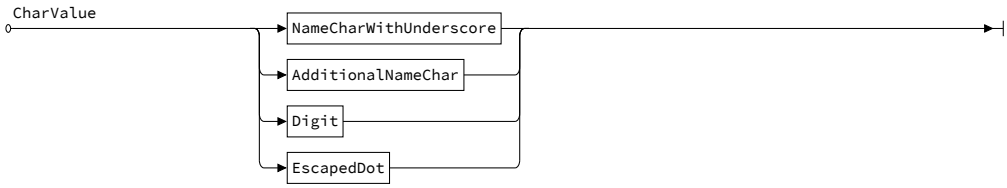

```
Prefix = NameChar { CharValue } ;
```

Diagramm A.64: Syntaxdiagramm der Prefix-Syntax



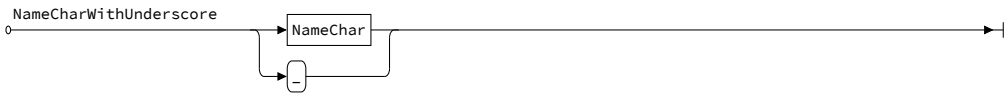

```
NameChar = 'A..Z' | 'a..z' | '\u00C0..\u00D6' | '\u00D8..\u00F6' | '\u00F8..\u02FF' | '\u0370..\u037D'
| '\u037F..\u1FFF' | '\u200C..\u200D' | '\u2070..\u218F' | '\u2C00..\u2FEF' | '\u3001..\uD7FF' |
'\uF900..\uFDCF' | '\uFDF0..\uFFFD' | '\u1000..\uEFFF' ;
```

Diagramm A.65: Syntaxdiagramm der NameChar-Syntax



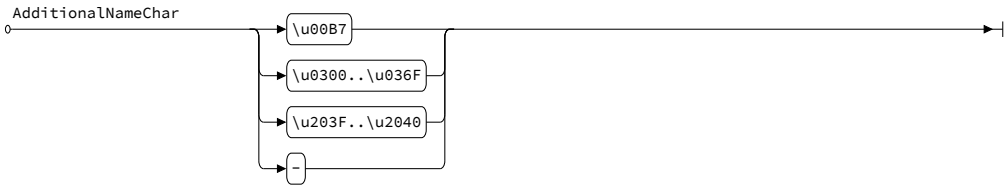

```
CharValue = NameCharWithUnderscore | AdditionalNameChar | Digit | EscapedDot ;
```

Diagramm A.66: Syntaxdiagramm der CharValue-Syntax



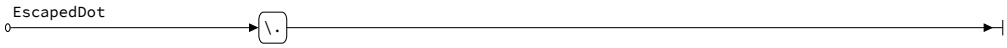

```
NameCharWithUnderscore = NameChar | '_' ;
```

Diagramm A.67: Syntaxdiagramm der NameCharWithUnderscore-Syntax



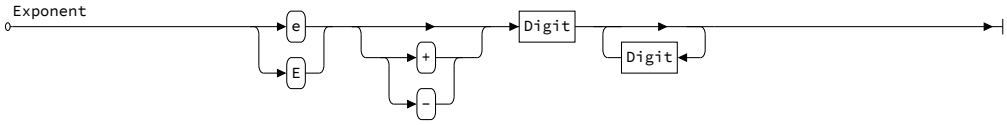

```
AdditionalNameChar = '\u00B7' | '\u0300..\u036F' | '\u203F..\u2040' | '-' ;
```

Diagramm A.68: Syntaxdiagramm der AdditionalNameChar-Syntax



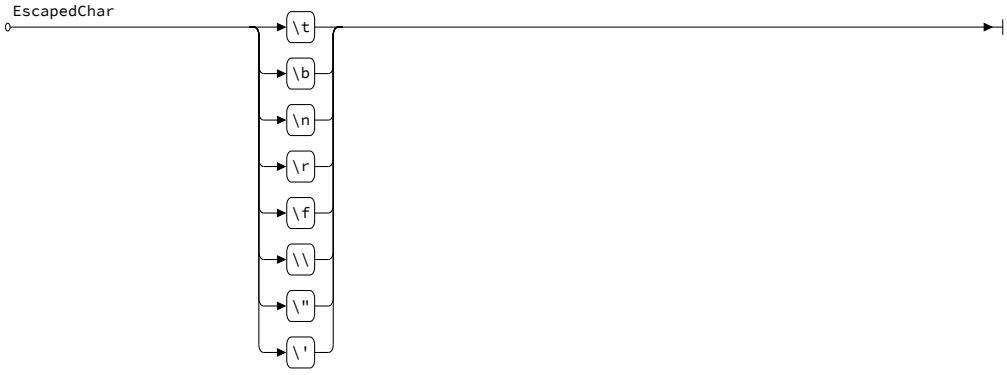

```
EscapedDot = '\.' ;
```

Diagramm A.69: Syntaxdiagramm der EscapedDot-Syntax




```
Exponent = ( 'e' | 'E' ) [ '+' | '-' ] Digit { Digit } ;
```

Diagramm A.70: Syntaxdiagramm der Exponent-Syntax




```
EscapedChar = '\t' | '\b' | '\n' | '\r' | '\f' | '\\' | '\"' | '\'' ;
```

Diagramm A.71: Syntaxdiagramm der EscapedChar-Syntax


```
ConditionalOr = '|' | '||' ;
```

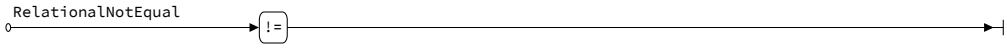
Diagramm A.72: Syntaxdiagramm der ConditionalOr-Syntax


```
ConditionalAnd = '&' | '&&' ;
```

Diagramm A.73: Syntaxdiagramm der ConditionalAnd-Syntax


```
RelationalEqual = '=' | '==' ;
```

Diagramm A.74: Syntaxdiagramm der RelationalEqual-Syntax




```
RelationalNotEqual = '!=' ;
```

Diagramm A.75: Syntaxdiagramm der RelationalNotEqual-Syntax


```
RelationalGreater = '>' ;
```

Diagramm A.76: Syntaxdiagramm der RelationalGreater-Syntax


```
RelationalGreaterEqual = '>=' ;
```

Diagramm A.77: Syntaxdiagramm der RelationalGreaterEqual-Syntax


```
RelationalLower = '<' ;
```

Diagramm A.78: Syntaxdiagramm der RelationalLower-Syntax


```
RelationalLowerEqual = '<=' ;
```

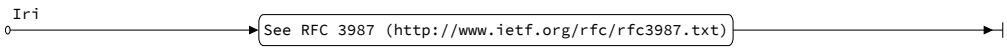
Diagramm A.79: Syntaxdiagramm der RelationalLowerEqual-Syntax


```
ContainsOperator = '~' ;
```

Diagramm A.80: Syntaxdiagramm der ContainsOperator-Syntax


```
Digit = '0..9' ;
```

Diagramm A.81: Syntaxdiagramm der Digit-Syntax




```
Iri = 'See RFC 3987 (http://www.ietf.org/rfc/rfc3987.txt)' ;
```

Diagramm A.82: Syntaxdiagramm der Iri-Syntax

ANHANG B

Implementierung des SemwidgQL-zu-SPARQL-Transcompilers in Pseudocode

Listing B.1: Pseudocode der toSparql-Funktion

```
1  FUNCTION ARRAY toSparql ( STRING semwidgQuery, ARRAY namespaces, ARRAY
       ↪ namedResources ) {
2
3    NODE ast = getAbstractSyntaxTreeFromQuery( semwidgQuery );
4    ast = replaceNamedResourcesWithURIs( ast, namedResources );
5
6    ARRAY queries = [];
7
8    WALK ast RECURSIVELY AND SAVE CURRENT NODE IN node {
9
10     IF node IS A SimpleQuery {
11
12       STRING sparqlQuery = buildQuery( node, namespaces );
13       queries.push( sparqlQuery );
14
15     }
16   }
17
18   RETURN queries;
19 }
```

Listing B.2: Pseudocode der buildQuery-Funktion

```
1  FUNCTION STRING buildQuery ( NODE simpleQuery, ARRAY namespaces ) {
2
3    STRING query = "";
4
5    query += buildPrefixPart( simpleQuery, namespaces );
6    query += buildSelectPart( simpleQuery );
7    query += buildWherePart( simpleQuery );
8    query += buildGroupPart( simpleQuery );
9
10   RETURN query;
11 }
```

Listing B.3: Pseudocode der buildPrefixPart-Funktion

```
1  FUNCTION STRING buildPrefixPart ( NODE simpleQuery, ARRAY namespaces ) {
2
3    STRING query = "";
4
5    FOR EVERY USED NAMESPACE ns IN simpleQuery {
6
7      query += "PREFIX " + ns + ": <" + namespaces[ns] + ">";
8
9    }
10
11   RETURN query;
12 }
```

Listing B.4: Pseudocode der buildSelectPart-Funktion

```
1  FUNCTION STRING buildSelectPart ( NODE simpleQuery ) {
2
```

```
3     STRING query = "SELECT DISTINCT ";
4
5     ARRAY resourcesAndProperties = getUsedResourcesAndProperties(
          ↪ simpleQuery );
6     ARRAY hiddenElements = getHiddenElements( simpleQuery );
7
8     BOOLEAN aggregate = IS getGroupExpressions( simpleQuery ) NOT EMPTY;
9
10    FOR EACH NODE element IN resourcesAndProperties {
11
12      BOOLEAN hidden = IS element PART OF hiddenElements;
13
14      IF NOT hidden {
15
16        STRING variableName = convertQueryElementToVariableName( element );
17        STRING aggregateFunction = getAggregateExpressions( element );
18
19        IF element IS ANY Property OR Wildcard {
20
21          IF aggregate {
22            query += aggregateFunction + " (" + variableName + ") AS " +
                  ↪ variableName;
23          } ELSE {
24            query += variableName;
25          }
26
27        } ELSE {
28
29          STRING resourceURI = element.getUri();
30
31          IF aggregate {
32            query += aggregateFunction + " (<" + resourceURI + ">) AS " +
                  ↪ variableName;
33          } ELSE {
34            query += "(<" + resourceURI + ">) AS " + variableName;
35          }
36        }
37      }
38    }
39
40    RETURN query;
41  }
```

Listing B.5: Pseudocode der buildWherePart-Funktion

```
1   FUNCTION STRING buildWherePart ( NODE simpleQuery ) {
2
3     STRING query = "WHERE {" + buildPathExpression( simpleQuery.getResource
          ↪ (), simpleQuery.getPathExpression() ) + "}";
4
5     RETURN query;
6   }
```

Listing B.6: Pseudocode der buildPathExpression-Funktion

```
1   FUNCTION STRING buildPathExpression ( NODE previousElement, NODE
        ↪ pathExpression ) {
```

```
2
3     STRING query = "";
4
5     IF pathExpression IS NOT EMPTY {
6
7       IF pathExpression IS A MultiplePathExpression {
8
9         FOR EACH NODE pathEx IN pathExpression {
10
11          query += buildPathExpression( previousElement, pathEx );
12
13        }
14
15      } ELSE {
16
17        NODE prop = pathExpression.getFirstProperty();
18
19        IF previousElement IS A Resource {
20
21          query += buildFirstTriple( previousElement, prop );
22          query += buildFilter( previousElement );
23
24        } ELSE {
25
26          query += buildTriple( previousElement, prop );
27
28        }
29
30        query += buildFilter( prop );
31      }
32
33    } ELSE {
34
35      IF previousElement IS A ForwardWildcard {
36
37        STRING variableName = convertQueryElementToVariableName(
              ↪ previousElement );
38
39        query += variableName + " ?p ?o . ";
40      }
41    }
42
43    RETURN query;
44  }
```

Listing B.7: Pseudocode der buildFirstTriple-Funktion

```
1   FUNCTION STRING buildFirstTriple ( NODE elem1, NODE elem2 ) {
2
3     STRING query = "";
4
5     STRING value1;
6
7     IF elem1 IS A Wildcard {
8
9       value1 = convertQueryElementToVariableName( elem1 );
10
```

```
11    } ELSE {
12
13      value1 = elem1.getValue();
14
15    }
16
17    IF elem2 IS A ForwardProperty OR ForwardPropertyWithFilter {
18
19      query += value1 + " " + elem2.getValue() + " " +
            ↪ convertQueryElementToVariableName( elem2 ) + " . ";
20
21    } ELSE IF elem2 IS A ReversedProperty OR ReversedPropertyWithFilter {
22
23      query += convertQueryElementToVariableName( elem2 ) + " " + elem2.
            ↪ getValue() + " " + value1 + " . ";
24
25    } ELSE IF elem2 IS A ForwardWildcard OR ForwardWildcardWithFilter {
26
27      STRING propetyName = convertQueryElementToVariableName( elem1 );
28
29      query += value1 + " " + getUniqueVariableName( elem1, elem2 ) + " " +
            ↪ convertQueryElementToVariableName( elem2 ) + " . ";
30
31    } ELSE IF elem2 IS A ReversedWildcard OR ReversedWildcardWithFilter {
32
33      STRING propetyName = convertQueryElementToVariableName( elem2 );
34
35      query += convertQueryElementToVariableName( elem2 ) + " " +
            ↪ getUniqueVariableName( elem2, elem1 ) + " " + value1 + " . ";
36    }
37
38    RETURN query;
39  }
```

Listing B.8: Pseudocode der buildFilter-Funktion

```
1   FUNCTION STRING buildFilter ( NODE property ) {
2
3     STRING query = "";
4
5     IF property HAS FILTER EXPRESSION {
6
7       ARRAY filterExpressions = property.getFilter();
8       // filter array has form [filter expression, conditional operator,
            ↪ filter expression, conditional operator, ..., filter expression
            ↪ ]
9
10      FOR EACH NODE filterExpression IN filterExpressions {
11
12        query += buildFilterTriples( property, filterExpression );
13
14      }
15
16      ARRAY realFilters = getRealFilters( filterExpressions );
17      ARRAY pseudoFilters = getPseudoFilters( filterExpressions );
18
19      IF realFilters IS NOT EMPTY {
```

```
20
21        query += "FILTER (";
22
23        FOR EACH NODE filterExpression, conditionalOperator IN realFilters {
24
25          query += buildFilterExpression( property, filterExpression );
26          query += conditionalOperator.getValue();
27
28        }
29
30        query += ") . ";
31
32      }
33
34      FOR EACH NODE filterExpression IN pseudoFilters {
35
36        IF filterExpression IS A Timeinterval {
37
38          STRING variableName = convertQueryElementToVariableName( property
                ↪ );
39
40          query += "BIND(FLOOR((xsd:dateTime(" + variableName + ") - xsd:
                ↪ dateTime("1970-01-01T00:00:00")) / (" + filterExpression.
                ↪ getTimeintervalInSeconds() + ")) AS " + variableName + "
                ↪ _timeinterval)";
41
42        }
43      }
44    }
45
46    RETURN query;
47  }
```

Listing B.9: Pseudocode der buildFilterTriples-Funktion

```
1   FUNCTION STRING buildFilterTriples ( NODE property, NODE filterExpression
        ↪ ) {
2
3     STRING query = "":
4
5     IF filterExpression IS A RelationalExpression {
6
7       ARRAY properties = [ property ].concat( filterExpression.getProperties
            ↪ () );
8
9       FOR EACH prop1, prop2 IN properties {
10
11        query += buildTriple( prop1, prop2 );
12
13      }
14
15      ARRAY values = filterExpression.getValue();
16
17      FOR EACH value IN values {
18
19        IF value IS A SimpleQuery {
20
```

```
21          query += buildPathExpression( value.getResource(), value.
                ↪ getPathExpression() );
22
23        }
24
25      }
26
27    } ELSE IF filterExpression IS A Self {
28
29      ARRAY values = filterExpression.getValue();
30
31      FOR EACH value IN values {
32
33        IF value IS A SimpleQuery {
34
35          query += buildPathExpression( value.getResource(), value.
                ↪ getPathExpression() );
36
37        }
38
39      }
40
41    } ELSE IF filterExpression IS A Type {
42
43      STRING variableName = convertQueryElementToVariableName( property );
44
45      query += variableName + " rdf:type ?TypeOf_" + variableName.substr( 1
            ↪ ) " . ";
46      // .substr( 1 ) return the string without the first char ("?")
47    }
48
49    RETURN query;
50  }
```

Listing B.10: Pseudocode der buildFilterExpression-Funktion

```
1   FUNCTION STRING buildFilterExpression ( NODE property, NODE
        ↪ filterExpression ) {
2
3     STRING query = "";
4
5     STRING variableName = convertQueryElementToVariableName( property );
6
7     IF filterExpression IS A Language {
8
9       query += buildFilterExpressionLanguage( filterExpression, variableName
            ↪  );
10
11    } ELSE IF filterExpression IS A Type {
12
13      query += buildFilterExpressionType( filterExpression, variableName );
14
15    } ELSE IF filterExpression IS A Self {
16
17      query += buildFilterExpressionSelf( filterExpression, variableName );
18
19    } ELSE IF filterExpression IS A RelationalExpression {
```

```
20
21      query += buildFilterExpressionRelationalExpression( filterExpression,
            ↪ variableName );
22
23    }  ELSE IF filterExpression IS A Timestart {
24
25      query += buildFilterExpressionTimestart( filterExpression,
            ↪ variableName );
26
27    } ELSE IF filterExpression IS A Timeend {
28
29      query += buildFilterExpressionTimeend( filterExpression, variableName
            ↪ );
30
31    }
32
33    RETURN query;
34  }
```

Listing B.11: Pseudocode der buildFilterExpressionLanguage-Funktion

```
1   FUNCTION STRING buildFilterExpressionLanguage ( NODE filterExpression,
        ↪ STRING variableName ) {
2
3     STRING query = "";
4
5     query += "lang(" + variableName + ") = "" || langMatches(lang(" +
          ↪ variableName + "), " + filterExpression.getLanguage() + ")";
6
7     RETURN query;
8   }
```

Listing B.12: Pseudocode der buildFilterExpressionType-Funktion

```
1   FUNCTION STRING buildFilterExpressionType ( NODE filterExpression, STRING
        ↪ variableName ) {
2
3     STRING query = "";
4
5     query += "?TypeOf_" + variableName.substr( 1 ) + " " + filterExpression.
          ↪ getOperator + " " + filterExpression.getType();
6
7     RETURN query;
8   }
```

Listing B.13: Pseudocode der buildFilterExpressionSelf-Funktion

```
1   FUNCTION STRING buildFilterExpressionSelf ( NODE filterExpression, STRING
        ↪ variableName ) {
2
3     STRING query = "";
4
5     ARRAY values = filterExpression.getValue();
6     ARRAY lastProperties = [];
7
8     FOR EACH value IN values {
```

```
 9
10      IF value IS A SimpleQuery {
11
12        IF value.getPathExpression() IS NOT EMPTY {
13
14          lastProperties.push( getLastPropertiesOfPathExpression( value.
                ↪ getPathExpression() ) );
15
16        }
17      }
18    }
19
20    IF lastProperties IS EMPTY {
21    // filterExpression does not contain nested queries with path
          ↪ expressions
22
23      IF filterExpression.getOperator() IS "~" {
24      // Contains Operator
25
26        query += "CONTAINS(LCASE(STR(" + variableName + ")), LCASE(STR(";
27
28        FOR EACH NODE value IN values {
29
30          IF value IS ANY Resource {
31
32            query += convertQueryElementToVariableName( value.getResource()
                  ↪ );
33
34          } ELSE {
35
36            query += value + " ";
37
38          }
39        }
40
41        query += ")))";
42
43      } ELSE {
44
45        IF value[0] IS A String {
46
47          query += "STR(" + variableName + ") ";
48
49        } ELSE {
50
51          query += variableName + " ";
52
53        }
54
55        query += filterExpression.getOperator() + " ";
56
57        FOR EACH NODE value IN values {
58
59          IF value IS A SimpleQuery {
60
61            query += value.getResource().getValue() ;
62
```

```
63          } ELSE {
64
65            query += value + " ";
66
67          }
68        }
69      }
70
71    } ELSE {
72    // filterExpression contains nested queries with path expressions
73
74      // if the nested query contains (another nested query with) a
            ↪ MultiplePathExpression, the number of filters must be
            ↪ calculated. Filter expressions are concatenated with a logical
            ↪ disjunction ("||").
75      INT numberOfFilters = 1;
76
77      FOR EACH ARRAY property IN lastProperties {
78        numberOfFilters *= property.length;
79      }
80
81      ARRAY propertyIndices = [];
82      INT loops = lastProperties.length;
83      FOR i = 0 TO loops {
84        propertyIndices.push( 0 );
85      }
86
87      FOR i = 0 TO numberOfFilters {
88
89        IF filterExpression.getOperator() IS "~" {
90        // Contains Operator
91
92          query += "CONTAINS(LCASE(STR(" + variableName + ")), LCASE(";
93
94          INT simpleQueryIndex = 0;
95          FOR EACH NODE value IN values {
96
97            IF value IS A SimpleQuery {
98
99              ARRAY simpleQuerysProperties = lastProperties[simpleQueryIndex
                    ↪ ];
100             NODE simpleQuerysProperty = simpleQuerysProperties[
                    ↪ propertyIndices[simpleQueryIndex]];
101             simpleQueryIndex++;
102             query += convertQueryElementToVariableName(
                    ↪ simpleQuerysProperty ) + " ";
103
104           } ELSE {
105
106             query += value + " ";
107
108           }
109         }
110
111         query += "))";
112
113       } ELSE {
```

```
114
115           IF values CONTAINS A StringLiteral {
116
117             query += "STR(" + variableName + ") ";
118
119           } ELSE {
120
121             query += variableName + " ";
122
123           }
124
125           query += filterExpression.getOperator() + " ";
126
127           INT simpleQueryIndex = 0;
128           FOR EACH NODE value IN values {
129
130             IF value IS A SimpleQuery {
131
132               ARRAY simpleQuerysProperties = lastProperties[simpleQueryIndex
                      ↪ ];
133               NODE simpleQuerysProperty = simpleQuerysProperties[
                      ↪ propertyIndices[simpleQueryIndex]];
134               simpleQueryIndex++;
135               query += convertQueryElementToVariableName(
                      ↪ simpleQuerysProperty ) + " ";
136
137             } ELSE {
138
139               query += value + " ";
140
141             }
142           }
143         }
144
145         // Define last properties indices for next iteration
146         BOOLEAN increaseNext = true;
147         FOR j = 0 TO loops {
148
149           INT val = propertyIndices[j];
150           INT length = lastProperties[j].length;
151
152           IF increaseNext IS true {
153             propertyIndices[j] = (val + 1) % length;
154
155             IF val + 1 IS propertyIndices[j] {
156               increaseNext = false;
157             } ELSE {
158               increaseNext = true;
159             }
160           }
161         }
162
163         IF THIS IS NOT LAST ITERATION {
164           query += " || ";
165         }
166
167       }
```

```
168     }
169
170     RETURN query;
171   }
```

Listing B.14: Pseudocode der buildFilterExpressionRelationalExpression-Funktion

```
1   FUNCTION STRING buildFilterExpressionRelationalExpression ( NODE
        ↪ filterExpression, STRING variableName ) {
2
3     STRING query = "";
4
5     ARRAY values = filterExpression.getValue();
6     ARRAY lastProperties = [];
7
8     FOR EACH value IN values {
9
10      IF value IS A SimpleQuery {
11
12        IF value.getPathExpression() IS NOT EMPTY {
13
14          lastProperties.push( getLastPropertiesOfPathExpression( value.
                ↪ getPathExpression() ) );
15
16        }
17      }
18    }
19
20    ARRAY properties = filterExpression.getProperties();
21    NODE lastProperty = LAST ELEMENT OF properties;
22    STRING lastPropertyName = convertQueryElementToVariableName(
          ↪ lastProperty );
23
24    IF lastProperties IS EMPTY {
25    // filterExpression does not contain nested queries with path
          ↪ expressions
26
27      IF filterExpression.getOperator() IS "~" {
28      // Contains Operator
29
30        query += "CONTAINS(LCASE(STR(" + lastPropertyName + ")), LCASE(STR("
              ↪ ;
31
32        FOR EACH NODE value IN values {
33
34          IF value IS ANY Resource {
35
36            query += convertQueryElementToVariableName( value.getResource()
                  ↪ );
37
38          } ELSE {
39
40            query += value + " ";
41
42          }
43        }
44
```

```
45        query += ")))";
46
47      } ELSE {
48
49        IF values CONTAINS A StringLiteral {
50
51          query += "STR(" + lastPropertyName + ") ";
52
53        } ELSE {
54
55          query += lastPropertyName + " ";
56
57        }
58
59        query += filterExpression.getOperator() + " ";
60
61        FOR EACH value IN values {
62
63          IF value IS A SimpleQuery {
64
65            IF value.getPathExpression() IS NOT EMPTY {
66
67              query += value.getResource().getValue() + " ";
68
69            } ELSE {
70
71              query += value + " ";
72
73            }
74          }
75        }
76      }
77
78    } ELSE {
79    // filterExpression contains nested queries with path expressions
80
81      // if the nested query contains (another nested query with) a
          ↪ MultiplePathExpression, the number of filters must be
          ↪ calculated. Filter expressions are concatenated with a logical
          ↪ disjunction ("||").
82      INT numberOfFilters = 1;
83
84      FOR EACH ARRAY property IN lastProperties {
85        numberOfFilters *= property.length;
86      }
87
88      ARRAY propertyIndices = [];
89      INT loops = lastProperties.length;
90      FOR INT i = 0 TO loops {
91        propertyIndices.push( 0 );
92      }
93
94      FOR INT i = 0 TO numberOfFilters {
95
96        IF filterExpression.getOperator() IS "~" {
97        // Contains Operator
98
```

```
99          query += "CONTAINS(LCASE(STR(" + lastPropertyName + ")), LCASE(";
100
101         INT simpleQueryIndex = 0;
102         FOR EACH NODE value IN values {
103
104           IF value IS A SimpleQuery {
105
106             ARRAY simpleQuerysProperties = lastProperties[simpleQueryIndex
                    ↪ ];
107             NODE simpleQuerysProperty = simpleQuerysProperties[
                    ↪ propertyIndices[simpleQueryIndex]];
108             simpleQueryIndex++;
109             query += convertQueryElementToVariableName(
                    ↪ simpleQuerysProperty ) + " ";
110
111           } ELSE {
112
113             query += value + " ";
114
115           }
116         }
117
118         query += "))";
119
120       } ELSE {
121
122         IF values CONTAINS A StringLiteral {
123
124           query += "STR(" + lastPropertyName + ") ";
125
126         } ELSE {
127
128           query += lastPropertyName + " ";
129
130         }
131
132         query += filterExpression.getOperator() + " ";
133
134         INT simpleQueryIndex = 0;
135         FOR EACH NODE value IN values {
136
137           IF value IS A SimpleQuery {
138
139             ARRAY simpleQuerysProperties = lastProperties[simpleQueryIndex
                    ↪ ];
140             NODE simpleQuerysProperty = simpleQuerysProperties[
                    ↪ propertyIndices[simpleQueryIndex]];
141             simpleQueryIndex++;
142             query += convertQueryElementToVariableName(
                    ↪ simpleQuerysProperty ) + " ";
143
144           } ELSE {
145
146             query += value + " ";
147
148           }
149         }
```

```
150         }
151
152         // Define last properties indices for next iteration
153         BOOLEAN increaseNext = true;
154         FOR INT j = 0 TO loops {
155
156           INT val = propertyIndices[j];
157           INT length = lastProperties[j].length;
158
159           IF increaseNext IS true {
160             propertyIndices[j] = (val + 1) % length;
161
162             IF val + 1 IS propertyIndices[j] {
163               increaseNext = false;
164             } ELSE {
165               increaseNext = true;
166             }
167           }
168         }
169
170         IF THIS IS NOT LAST ITERATION {
171           query += " || ";
172         }
173       }
174     }
175
176     RETURN query;
177   }
```

Listing B.15: Pseudocode der buildFilterExpressionTimestart-Funktion

```
1   FUNCTION STRING buildFilterExpressionTimestart ( NODE filterExpression,
        ↪ STRING variableName ) {
2
3     STRING query = "";
4
5     query += "xsd:dateTime(" + variableName + ") >= " + readDateExpression(
          ↪ filterExpression.getTimestart() );
6
7     RETURN query;
8   }
```

Listing B.16: Pseudocode der buildFilterExpressionTimeend-Funktion

```
1   FUNCTION STRING buildFilterExpressionTimeend ( NODE filterExpression,
        ↪ STRING variableName ) {
2
3     STRING query = "";
4
5     query += "xsd:dateTime(" + variableName + ") <= " + readDateExpression(
          ↪ filterExpression.getTimeend() );
6
7     RETURN query;
8   }
```

Listing B.17: Pseudocode der buildTriple-Funktion

```
1  FUNCTION STRING buildTriple = ( NODE elem1, NODE elem2 ) {
2
3    STRING query = "";
4
5    STRING elem1Name = convertQueryElementToVariableName( elem1 );
6    STRING elem2Name = convertQueryElementToVariableName( elem2 );
7
8    IF elem2 IS A ForwardProperty OR ForwardPropertyWithFilter {
9
10     query += elem1Name + " " + elem2.getValue() + " " + elem2Name + " . ";
11
12   } ELSE IF elem2 IS A ReversedProperty OR ReversedPropertyWithFilter {
13
14     query += elem2Name + " " + elem2.getValue() + " " + elem1Name + " . ";
15
16   } ELSE IF elem2 IS A ForwardWildcard OR ForwardWildcardWithFilter {
17
18     query += elem1Name + " " + "?PropertyOf_" + elem1Name.substr(1) + "__"
         ↪  + elem2Name.substr(1) + " " + elem2Name + " . ";
19
20   } ELSE IF elem2 IS A ReversedWildcard OR ReversedWildcardWithFilter {
21
22     query += elem2Name + " " + "?PropertyOf_" + elem2Name.substr(1) + "__"
         ↪  + elem1Name.substr(1) + " " + elem1Name + " . ";
23
24   }
25
26   RETURN query;
27 };
```

Listing B.18: Pseudocode der buildGroupPart-Funktion

```
1  FUNCTION STRING buildGroupPart ( NODE simpleQuery ) {
2
3    STRING query = "";
4
5    ARRAY groupExpressions = getGroupExpressions( simpleQuery );
6
7    IF groupExpressions IS NOT EMPTY {
8      NODE groupExpression;
9
10     FOR EACH MAP<STRING, NODE> groupExpression IN groupExpressions {
11
12       IF groupExpression.get( "filter" ) IS A Timeinterval {
13
14         STRING name = convertQueryElementToVariableName( groupExpression.
             ↪ get( "element" ) ) + "_timeinterval";
15
16         query += "GROUP BY " + name;
17         query += "ORDER BY DESC(" + name + ")";
18       }
19     }
20   }
21
22   RETURN query;
```

```
23  };
```

Listing B.19: Pseudocode der getGroupExpressions-Funktion

```
1  FUNCTION ARRAY getGroupExpressions ( NODE node ) {
2
3    // Returns an array of hashmaps that contain a node's and all its child
         ↪ nodes' filter expressions of type TimeInterval. The hashmap also
         ↪ contains a pointer to the respective node
4
5    // MAP<STRING, NODE> groupExpression;
6    // groupExpression.map( "filter", filter );
7    // groupExpression.map( "element", node );
8    // groupExpressions.push( groupExpression );
9  };
```

Listing B.20: Pseudocode der getUsedResourcesAndProperties-Funktion

```
1  FUNCTION ARRAY getUsedResourcesAndProperties ( NODE simpleQuery ) {
2
3    // Returns an array of all elements in simpleQuery that are not part of
         ↪ a filter expression
4
5    // Example:
6    // {r1.ns:p1(ns:p2 < {r2.ns:p3}).[ns:p4(ns:p5 < 10), ns:p6.ns:p7]}
7    //   +  +     -        -  -        +     -            +     +
8    // returns an array containing the corresponding elements to r1, ns:p1,
         ↪ ns:p4, ns:p6 and ns:p7
9  }
```

Listing B.21: Pseudocode der getHiddenElements-Funktion

```
1  FUNCTION ARRAY getHiddenElements ( NODE simpleQuery ) {
2
3    // Returns an array of all elements in simpleQuery where the keyword
         ↪ @hide has the value TRUE
4
5    // Example:
6    // {r1.ns:p1(@hide = false).[ns:p2(@hide = true), ns:p3.ns:p4(@hide =
         ↪ true)]}
7    //   -  -                    +                    -     +
8    // returns an array containing the corresponding elements to ns:p2 and
         ↪ ns:p4
9  }
```

Listing B.22: Pseudocode der replaceNamedResourcesWithURIs-Funktion

```
1  FUNCTION NODE replaceNamedResourcesWithURIs ( NODE ast, ARRAY
       ↪ namedResources ) {
2
3    // Walks the Abstract Syntax Tree recursively and replaces every
         ↪ NamedResource with a QualifiedResource or a PrefixedResource,
         ↪ respectively, depending on the corresponding value in
         ↪ namedResources. The resulting Abstract Syntax Tree is returned.
4  }
```

Listing B.23: Pseudocode der getAbstractSyntaxTreeFromQuery-Funktion

```
1  FUNCTION NODE getAbstractSyntaxTreeFromQuery ( STRING semwidgQuery ) {
2
3    // Returns a revised version of the Abstract Syntax Tree of the
         ↪ SemwidgQL query which is defined by the SemwidgQL EBNF Syntax. To
         ↪  simplify the work with the AST, its nodes are consolidated and
         ↪ converted into objects (or data structures, depending on the used
         ↪  programming language. This pseudo code assumes that an object
         ↪ orientated programming language is used).
4  }
```

Listing B.24: Pseudocode der getLastPropertiesOfPathExpression-Funktion

```
1  FUNCTION ARRAY getLastPropertiesOfPathExpression ( NODE pathExpression) {
2
3    // Returns an array of the last property of a PathExpression. If the
         ↪ pathExpression ends with a MultiplePathExpression, the array
         ↪ contains the last properties of the path expressions of the
         ↪ MultiplePathExpression.
4  }
```

Listing B.25: Pseudocode der convertQueryElementToVariableName-Funktion

```
1  FUNCTION STRING convertQueryElementToVariableName ( NODE node ) {
2
3    // Returns a unique but reproducible variable name for a node of the
         ↪ queries AST, starting with a question mark ("?"). The reference
         ↪ implementation uses the name of the node plus a double underscore
         ↪  "__" plus a suffix that depends on the node's postion in the AST
         ↪ .
4
5    // Node name:
6    // If the node is defined by a fully quallified URI, enclosing angle
         ↪ brackets ("<", ">") are removed, as well as a possible trailing
         ↪ slash ("/"). Then, the part after the last slash or hash ("#") is
         ↪  used as name.
7
8    // <http://domain.tld/resource/MyResource/> -> MyResource
9    // <http://domain.tld/property#MyProperty> -> MyProperty
10
11   // If the node is defined by a prefixed URI, the part after the last
         ↪ colon (":") is used as name.
12   // ns:MyProperty -> MyProperty
13
14   // If the node is a wildcard resource or property ("*"), the name is "
         ↪ wildcard".
15   // * -> wildcard
16
17   // If the node is a reversed resource or property an "Of" is appended to
         ↪  the name.
18   // ^ns:MyProperty -> MyPropertyOf
19
20   // Suffix:
```

```
21    // The suffix starts with an R if the node is a resource or a P if the
          ↪ node is a property. The algorithm walks the AST from top to the
          ↪ regarding node for which the suffix should be created. If the
          ↪ currently visited node is a "Resource", "Property", "
          ↪ PathExpression", "MultiplePathExpression", or a "SimpleQuery", an
          ↪  underscore ("_") plus the index of the node in the list of its
          ↪ siblings is appended.
22
23    // r1.ns:p1.^ns:p2 -> SimpleQuery (0)  (irrelevant nodes were omitted)
24    //                      |           |
25    //               Resource (0)    PathExpression (1)
26    //                   |             |           |
27    //                 "r1"       Propery (0)  Property (1)
28    //                               |           |
29    //                             "ns:p1"      "ns:p2"
30
31    // r1 -> R_0_0, ns:p1 -> P_0_1_0, ns:p2 -> P_0_1_1
32
33    // Complete name:
34    // r1.ns:p1.^ns:p2
35    // Name of ns:p2 would be: ?p2Of__P_0_1_1
36  }
```

Listing B.26: Pseudocode der getRealFilters-Funktion

```
1   FUNCTION ARRAY getRealFilters ( ARRAY filterExpressions ) {
2
3     // Returns an array of filter expressions whose types are not Aggregate,
          ↪  Hide, or Timeinterval
4   }
```

Listing B.27: Pseudocode der getPseudoFilters-Funktion

```
1   FUNCTION ARRAY getPseudoFilters ( ARRAY filterExpressions ) {
2
3     // Returns an array of filter expressions whose types are Aggregate,
          ↪ Hide, or Timeinterval
4   }
```

Listing B.28: Pseudocode der readDateExpression-Funktion

```
1   FUNCTION STRING readDateExpression ( STRING expressionString ) {
2
3     // Returns a SPARQL compliant version of the date expression
4
5     // DateExpression : ("now"|Iso8601DateExpression) ( "+" | "-" ) (Number
          ↪ ("s"|"m"|"h"|"d"|"w") Character*)*
6
7     // "now" is converted into "now()", a date expression in ISO 8601 format
          ↪  is encapsulated into a xsd:dateTime typecasting ("xsd:dateTime(
          ↪ Iso8601DateExpression)").
8
9     // The last parts are converted into seconds each and summed up. Instead
          ↪  of only "s", "m", "h", "d", or "w", sec or seconds etc. are also
          ↪  valid units. (s = 1, m = 60, h = 3600, d = 86400, w = 604800,
          ↪ everything else = 0)
```

```
10
11      // Example:
12      // now - 1h 30m -> now() - 5400
13      // 2016-01-01T12:00:00 - 60 min -> xsd:dateTime("2016-01-01T12
            ↪ :00:00.000") - 3600
14  }
```

ANHANG C

Empirische Nutzerstudie zum Vergleich von SemwidgQL und SPARQL

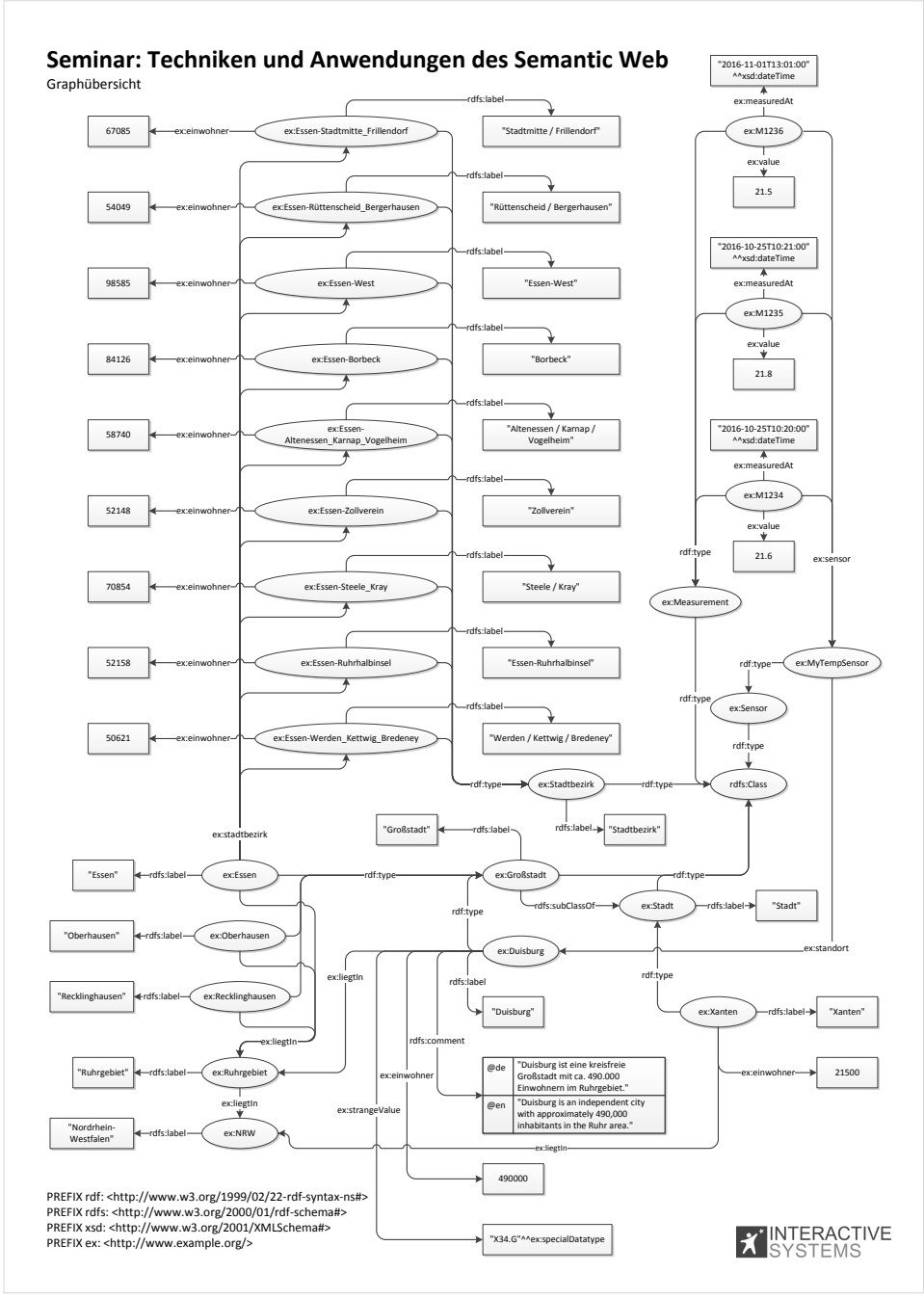
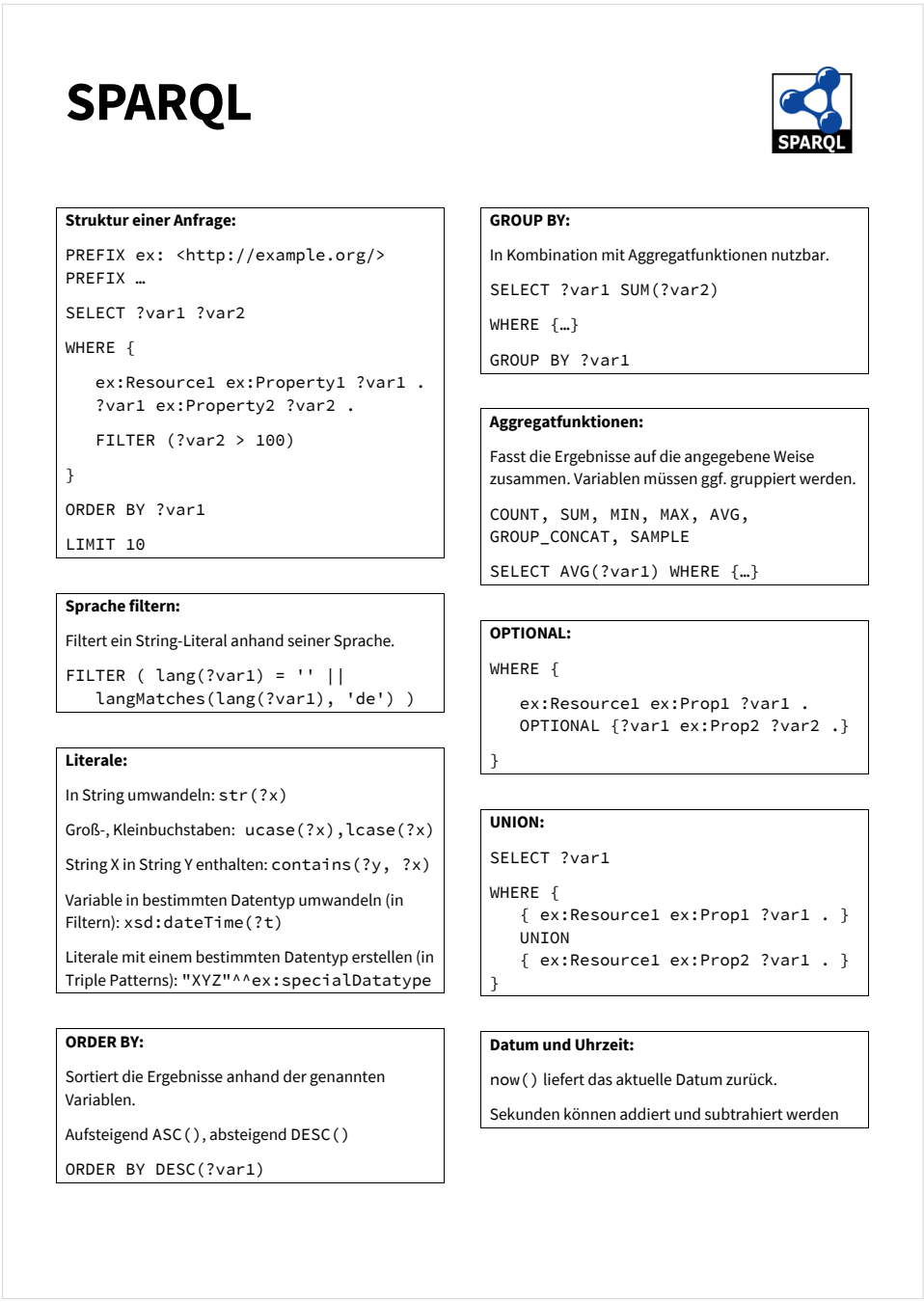


Abbildung C.1: Handout aus dem Seminar »Techniken und Anwendungen des Semantic Web«. Der Graph zeigt die Daten, welche für die Beispiele und Aufgaben aus dem Vortrag und der Evaluation genutzt wurden.



SPARQL

Struktur einer Anfrage:

```
PREFIX ex: <http://example.org/>
PREFIX …

SELECT ?var1 ?var2

WHERE {

   ex:Resource1 ex:Property1 ?var1 .
   ?var1 ex:Property2 ?var2 .

   FILTER (?var2 > 100)

}

ORDER BY ?var1

LIMIT 10
```

Sprache filtern:

Filtert ein String-Literal anhand seiner Sprache.

```
FILTER ( lang(?var1) = '' ||
   langMatches(lang(?var1), 'de') )
```

Literale:

In String umwandeln: `str(?x)`

Groß-, Kleinbuchstaben: `ucase(?x)`,`lcase(?x)`

String X in String Y enthalten: `contains(?y, ?x)`

Variable in bestimmten Datentyp umwandeln (in Filtern): `xsd:dateTime(?t)`

Literale mit einem bestimmten Datentyp erstellen (in Triple Patterns): `"XYZ"^^ex:specialDatatype`

ORDER BY:

Sortiert die Ergebnisse anhand der genannten Variablen.

Aufsteigend `ASC()`, absteigend `DESC()`

```
ORDER BY DESC(?var1)
```

GROUP BY:

In Kombination mit Aggregatfunktionen nutzbar.

```
SELECT ?var1 SUM(?var2)

WHERE {…}

GROUP BY ?var1
```

Aggregatfunktionen:

Fasst die Ergebnisse auf die angegebene Weise zusammen. Variablen müssen ggf. gruppiert werden.

```
COUNT, SUM, MIN, MAX, AVG,
GROUP_CONCAT, SAMPLE

SELECT AVG(?var1) WHERE {…}
```

OPTIONAL:

```
WHERE {

   ex:Resource1 ex:Prop1 ?var1 .
   OPTIONAL {?var1 ex:Prop2 ?var2 .}

}
```

UNION:

```
SELECT ?var1

WHERE {
   { ex:Resource1 ex:Prop1 ?var1 . }
   UNION
   { ex:Resource1 ex:Prop2 ?var1 . }
}
```

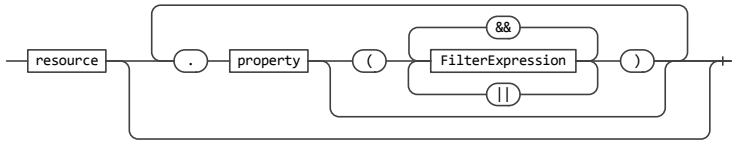
Datum und Uhrzeit:

`now()` liefert das aktuelle Datum zurück.

Sekunden können addiert und subtrahiert werden

Abbildung C.2: Handout aus dem Seminar »Techniken und Anwendungen des Semantic Web«. Das Handout enthält alle SPARQL-Befehle, die für die Lösung der Evaluationsaufgaben relevant waren.

SemwidgQL



Struktur einer Anfrage:

```
ex:Resource1.ex:Prop1.ex:Prop2
```

Multiple Property Selection

```
ex:Resource1.[ex:Prop1, ex:Prop2]
```

Wildcards

Als Resource oder als Property.

```
*.ex:Prop1, ex:Resource1.*
```

Inverse Properties

Umdrehen der Subjekt-Prädikat-Objekt-Relation. Liefert das Subjekt dieser Relation zurück, statt des Objekts.

```
ex:Resource1.^ex:Prop1
```

Filter:

Beziehen sich auf eine Eigenschaft des zu filternden Objekts.

```
*(ex:Prop1 > 100)
```

Zusätzlicher Contains-Operator:

```
*(ex:Prop1 ~ 'xyz')
```

Filter Expressions:

`@lang` – Filterung der Sprache.
`(@lang = 'de')`

`@self` – Zugriff auf den Wert des Filter-Properties.
`(@self > 10)`

`@type` – Abkürzung für `rdf:type`.
`(@type = ex:MyClass)`

`@timestart / @timeend` – Filterung zeitbezogender Daten. Relative Zeitangaben möglich.
`(@timestart = "now - 2 hours")`

Pseudo-Filter

`@aggregate` – Fasst die Ergebnisse auf die angegebene Weise zusammen. Mögliche Werte sind `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`, `GROUP_CONCAT`, und `SAMPLE`. Ggf. müssen andere Werte mit `@hide` ausgeblendet werden.
`(@aggregate = 'SUM')`

`@hide` – Entfernt einzelne Teile der Anfrage aus dem Ergebnis.
`(@hide = true)`

`@optional` – markiert einzelne Properties als optional.
`(@optional = true)`

`@predicate` – Fügt das entsprechende Prädikat der S-P-O-Relation dem Ergebnis hinzu.
`(@predicate = true)`

`@timeinterval` – aggregiert zeitbezogene Daten innerhalb des angegebenen Intervalls.
`(@timeinterval = '10 mins')`

Abbildung C.3: Handout aus dem Seminar »Techniken und Anwendungen des Semantic Web«. Das Handout enthält alle SemwidgQL-Befehle, die für die Lösung der Evaluationsaufgaben relevant waren.

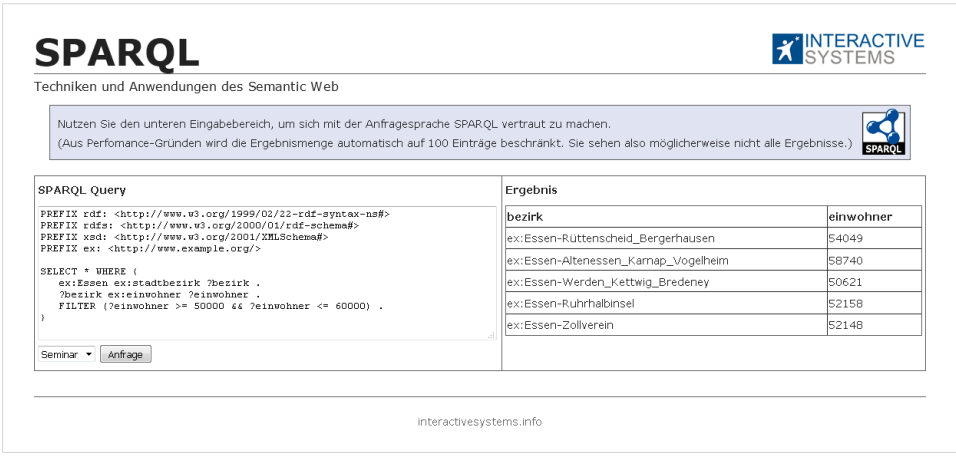


Abbildung C.4: Webseite zur Eingabe von SPARQL-Queries. Der SPARQL-Endpoint und die Präfixe waren vordefiniert.

Abbildung C.5: Webseite zur Bearbeitung der Evaluationsaufgaben.

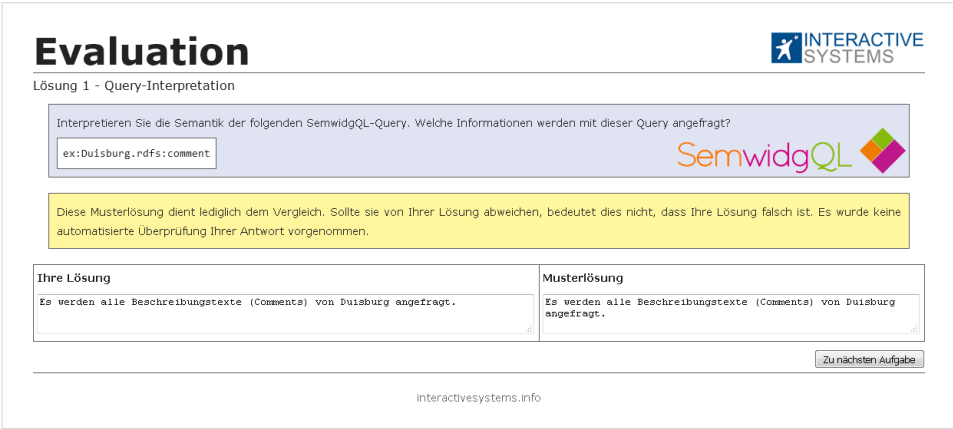


Abbildung C.6: Webseite mit Musterlösung einer der Evaluationsaufgaben.

Tabelle C.1: Aufgabe 1 der Evaluation.

Aufgabe 1:	Interpretieren Sie die Semantik der folgenden SPARQL- / SemwidgQL-Query. Welche Informationen werden mit dieser Query abgefragt?

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT ?x`
	`WHERE {`
	`  ex:Duisburg rdfs:comment ?x .`
	`}`

SemwidgQL:	`ex:Duisburg.rdfs:comment`
Musterlösung:	Es werden alle Beschreibungstexte (Comments) von Duisburg abgefragt.

Tabelle C.2: Aufgabe 2 der Evaluation.

Aufgabe 2:	Interpretieren Sie die Semantik der folgenden SPARQL- / SemwidgQL-Query. Welche Informationen werden mit dieser Query abgefragt?

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT ?x ?y`
	`WHERE {`
	`  ex:Essen ex:stadtbezirk ?x .`
	`  ?x rdfs:label ?y .`
	`}`

SemwidgQL:	`ex:Essen.ex:stadtbezirk.rdfs:label`
Musterlösung:	Es werden alle Labels für alle Stadtbezirke von Essen angefragt.

Tabelle C.3: Aufgabe 3 der Evaluation.

Aufgabe 3:	Interpretieren Sie die Semantik der folgenden SPARQL- / SemwidgQL-Query. Welche Informationen werden mit dieser Query abgefragt?

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT DISTINCT ?y ?z`
	`WHERE {`
	`  ?x ex:sensor ex:MyTempSensor .`
	`  ?x ex:value ?y .`
	`  FILTER (`
	`  ?y > 21.7`
	`  ) .`
	`  ?x ex:measuredAt ?z .`
	`}`

SemwidgQL:	`ex:MyTempSensor.^ex:sensor.` `  [ex:value(@self > 21.7), ex:measuredAt]`
Musterlösung:	Es werden alle Sensorwerte von »ex:MyTempSensor« abgefragt, die höher als 21,7 °C waren. Zusätzlich wird der dazugehöriger Messzeitpunkt abgefragt.

Tabelle C.4: Aufgabe 4 der Evaluation.

Aufgabe 4:	Fragen Sie die Einwohnerzahl von Duisburg ab.

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT ?pop`
	`WHERE {`
	`  ex:Duisburg ex:einwohner ?pop .`
	`}`

SemwidgQL:	`ex:Duisburg.ex:einwohner`

Tabelle C.5: Aufgabe 5 der Evaluation.

Aufgabe 5:	Fragen Sie alle Stadtteilen von Essen und ihre Einwohnerzahlen ab.

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT ?bezirk ?pop`
	`WHERE {`
	`  ex:Essen ex:stadtbezirk ?bezirk .`
	`  ?bezirk ex:einwohner ?pop .`
	`}`

SemwidgQL:	`ex:Essen.ex:stadtbezirk:ex.einwohner`

Tabelle C.6: Aufgabe 6 der Evaluation.

Aufgabe 6:	Fragen Sie das Label und die Beschreibung (rdfs:comment) von Duisburg ab.

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT ?label ?comment`
	`WHERE {`
	`  ex:Duisburg rdfs:label ?label .`
	`  ex:Duisburg rdfs:comment ?comment .`
	`}`

SemwidgQL:	`ex:Duisburg.[rdfs:label, rdfs:comment]`

Tabelle C.7: Aufgabe 7 der Evaluation.

Aufgabe 7:	Fragen Sie die deutschsprachige Beschreibung (rdfs:comment) von Duisburg ab.

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT ?comment`
	`WHERE {`
	`  ex:Duisburg rdfs:comment ?comment .`
	`  FILTER (`
	`  langMatches(lang(?comment), 'de')`
	`  ) .`
	`}`

SemwidgQL:	`ex:Duisburg.rdfs:comment(@lang = 'de')`

Tabelle C.8: Aufgabe 8 der Evaluation.

Aufgabe 8:	Fragen Sie die Labels von allen Städten im Ruhrgebiet ab.

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT ?label`
	`WHERE {`
	`  ?cities ex:liegtIn ex:Ruhrgebiet .`
	`  ?cities rdfs:label ?label .`
	`}`

SemwidgQL:	`ex:Ruhrgebiet.^ex:liegtIn.rdfs:label`

Tabelle C.9: Aufgabe 9 der Evaluation.

Aufgabe 9:	Fragen Sie die Labels von allen Essener Stadtbezirken ab, die mehr als 60.000 Einwohner haben.

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT ?label`
	`WHERE {`
	`  ex:Essen ex:stadtbezirk ?stadtbezirke .`
	`  ?stadtbezirke ex:einwohner ?pop .`
	`  FILTER (`
	`  ?pop > 60000`
	`  ) .`
	`  ?stadtbezirke rdfs:label ?label .`
	`}`

SemwidgQL:	`ex:Essen.ex:stadtbezirk(ex:einwohner > 60000).rdfs:label`

Tabelle C.10: Aufgabe 10 der Evaluation.

Aufgabe 10:	Fragen Sie die Labels von allen Großstädten ab.

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT ?label`
	`WHERE {`
	`  ?cities rdf:type ex:Großstadt .`
	`  ?cities rdfs:label ?label .`
	`}`

SemwidgQL:	`*(@type = ex:Großstadt).rdfs:label`

Tabelle C.11: Aufgabe 11 der Evaluation.

Aufgabe 11:	Fragen Sie die Labels von allen Großstädten ab, die die Zeichenkette »hausen« im Namen haben.

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT ?label`
	`WHERE {`
	`  ?cities rdf:type ex:Großstadt .`
	`  ?cities rdfs:label ?label .`
	`  FILTER (`
	`  CONTAINS( LCASE(STR(?label)), LCASE(STR('hausen')) )`
	`  ) .`
	`}`

SemwidgQL:	`*(@type = ex:Großstadt).rdfs:label(@self ~ 'hausen')`

Tabelle C.12: Aufgabe 12 der Evaluation.

Aufgabe 12:	Fragen Sie alle Messwerte ab, die innerhalb der letzten Woche aufgenommen wurde.

SPARQL:	`PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>;`
	`PREFIX ex: <http://www.example.org/>;`
	`SELECT DISTINCT ?measurement`
	`WHERE {`
	`  ?measurement rdf:type ex:Measurement .`
	`  ?measurement ex:measuredAt ?measuredAt .`
	`  FILTER (`
	`  xsd:dateTime(?measuredAt) >= now() - 60*60*24*7`
	`  ) .`
	`}`

SemwidgQL:	`*(@type = ex:Measurement).ex:measuredAt(@timestart =` `  'now - 1 week' && @hide = true)`

ANHANG D

Dokumentation der SemwidgJS-Elemente

D.1 Konfigurationselemente

Jedes Konfigurationselement (vgl. Abschnitt 3.3.2) wird anhand mehrerer data-*-Attribute eines HTML-Elements beschrieben. Typischerweise werden hierfür nicht-sichtbarer HTML-Elemente wie `div` oder `span` genutzt. Die Objekt-Klasse (oder der Typ) des Konfigurationselements wird durch das data-*-Attribut `data-sw-config` und den Klassennamen bestimmt. Nutzbare Klassennamen sind `endpoint`, `resource`, `namespace`, `mapping` und `proxy`. Die verfügbaren Klassen-spezifischen Attribute hängen von der jeweiligen Objekt-Klasse des Konfigurationselements ab und beginnen mit dem Präfix `data-swp-`. Einige dieser Attribute sind verpflichtend anzugeben.

D.1.1 Endpoint

Ein Endpoint-Konfigurationselement enthält die Informationen, die benötigt werden, um eine Verbindung zu einem SPARQL-Endpoint aufzubauen. Innerhalb einer Webseite können beliebig viele Endpoints definiert werden. Einer dieser Endpoints kann als Standard-Endpoint ausgewählt werden. Wird in einer Abfrage oder einem Widget kein spezifischer Endpoint angegeben, wird die Abfrage an diesen Standard-Endpoint gestellt.

Objekt-Klasse:

Attribut	Wert
`data-sw-config`	`endpoint`

Attribute:

Attribut	Pflicht	Beschreibung
`data-swp-id`	*	Ein einzigartiger Wert, der dieses Konfigurationselement identifiziert. Resource-Elemente können diesen Wert nutzen, um auf den zu nutzenden SPARQL-Endpoint zu verlinken.
`data-swp-url`	*	Die URL des SPARQL-Endpoints. Für gewöhnlich endet diese URL mit `/sparql`. Der Protokollteil der URL (`http`, `https`) sollte nicht mit anzugeben, damit der Browser das korrekte Protokoll selbst auswählen kann. Die Nutzung des falschen Protokolls führt ansonsten zu CORS-Fehlern.
`data-swp-default-graph-uri`		Der `default-graph-uri`, der in die SPARQL-Abfrage eingebettet werden soll.
`data-swp-default-lang`		Die Standardsprache des Endpoints. Dieser Wert kann von SemwidgED genutzt werden um geeignete Beispielwerte für die Abfragevervollständigung anzuzeigen (vgl. Abschnitt 4.3.4).
`data-swp-proxy`		Die ID des Proxy-Elements (vgl. Anhang D.1.5) oder die Werte `none` oder `default`.
`data-swp-default`		Ein boolescher Wert (`true`, `false`), der angibt, ob der Endpoint als Standard-Endpoint genutzt werden soll, wenn kein Abfrage- oder Widget-spezifischer Endpoint angegeben wurde.

Beispiel:

```
<div data-sw-config="endpoint"
     data-swp-id="dbpedia"
     data-swp-url="//dbpedia.org/sparql"
     data-swp-default-graph-uri="http://dbpedia.org"
     data-swp-default-lang="en"
     data-swp-proxy="none"
     data-swp-default="true">
</div>
```

Zusätzliche Informationen

Vor der Ausführung einer Abfrage sucht SemwidgJS nach dem zu nutzenden SPARQL-Endpoint. Die möglichen Fundstellen sind wie folgt priorisiert:

1. Der Endpoint, der mit dem Haupt-Resource-Element der Abfrage (vgl. Abschnitt 3.3.2: *Resources*) verknüpft ist, falls es sich bei dieser Abfrage um eine SemwidgQL-Abfrage handelt.

2. Der Widget-Eigenschaft-spezifische Endpoint (vgl. Anhang D.2.3), der der Widget-Eigenschaft untergeordnet ist, in welcher die Abfrage formuliert wurde.

3. Der Widget-spezifische Endpoint (vgl. Anhang D.2.1), der zu dem Widget gehört, in welcher die Abfrage formuliert wurde.

4. Der ausgewählte Standard-Endpoint der Webseite.

D.1.2 Resource

Ein Resource-Konfigurationselement erzeugt eine benannte Resource (vgl. Abschnitt 2.3.1: *Benannte Resources*), die für die Formulierung von SemwidgQL-Abfragen genutzt werden kann, anhand eines Namens und einer Resource-URI, und verknüpft dieses Element mit einem Endpoint-Konfigurationselement. Die Verknüpfung erfolgt durch Angabe des ID-Wertes des Endpoint-Konfigurationselements. Benannte Resources sind für die Formulierung von SemwidgQL-Abfragen nicht zwingend nötig, erleichtern die Verwendung von SemwidgJS jedoch erheblich.

Objekt-Klasse:

Attribut	Wert
`data-sw-config`	`resource`

Attribute:

Attribut	Pflicht	Beschreibung
`data-swp-id`	*	Ein einzigartiger Wert, der dieses Konfigurationselement identifiziert. Dieser Wert kann innerhalb einer SemwidgQL-Abfrage als benannte Resource (vgl. Abschnitt 2.3.1: *Benannte Resources*) genutzt werden.
`data-swp-uri`	*	Die URI dieser Resource.
`data-swp-endpoint`	*	Die ID des zu verwendenden SPARQL-Endpoints. Da dieses Konfigurationselement einem Endpoint zugeordnet ist, muss dieser dann nicht als zusätzlicher Parameterwert innerhalb der Widget-Definition angegeben werden.

Beispiel:

```
1 <div data-sw-config="resource"
2      data-swp-id="physicist"
3      data-swp-uri="http://dbpedia.org/resource/Albert_Einstein"
4      data-swp-endpoint="dbpedia">
5 </div>
```

D.1.3 Namespace

Namespace-Konfigurationselemente beschreiben Namespaces beziehungsweise Präfixe, die es Anwendern erlauben, präfigierte URIs für Resources oder Eigenschaften innerhalb von SemwidgQL-Abfragen zu nutzen (vgl. Abschnitt 2.3.1: *Präfixdefinition*). Da diese nicht direkt in der Abfrage definiert werden können, wie es beispielsweise in SPARQL der Fall ist, müssen Präfixe über Namespace-Konfigurationselemente deklariert werden und können sie so von verschiedenen Abfragen wiederverwendet werden.

Objekt-Klasse:

Attribut	Wert
`data-sw-config`	`namespace`

Attribute:

Attribut	Pflicht	Beschreibung
`data-swp-id`	*	Ein einzigartiger Wert, der dieses Konfigurationselement identifiziert und innerhalb von SemwidgQL-Abfragen als Prefix-Bezeichnung fungiert.
`data-swp-uri`	*	Die `Uri` des Namespaces.

Beispiel:

```
1  <div data-sw-config="namespace"
2       data-swp-id="dbpr"
3       data-swp-uri="http://dbpedia.org/resource/">
4  </div>
```

D.1.4 Mapping

Mappings dienen als globale Variablen, die in jeden Parameterwert eines SemwidgJS-Widgets eingebettet werden können, unabhängig davon, ob dieser Parameterwert statisch ist oder eine Datenbankabfrage beschreibt. Ein Mapping-Konfigurationselemente enthält einen Mapping-Bezeichner und einen Datenwert. Damit ein SemwidgJS erkennen kann, dass ein Parameterwert ein Mapping enthält, muss der Mapping-Bezeichner durch doppelte Dollarzeichen eingeschlossen werden.

Objekt-Klasse:

Attribut	Wert
`data-sw-config`	`mapping`

Attribute:

Attribut	Pflicht	Beschreibung
`data-swp-id`	*	Ein einzigartiger Wert, der dieses Konfigurationselement identifiziert und innerhalb von `query`-Attributen (vgl. Anhang D.2.2) als Platzhalter für den `value`-Wert dienen. der Mapping-Bezeichner durch doppelte Dollarzeichen eingeschlossen werden (z. B. $$language$$).
`data-swp-value`	*	Der Wert, durch der der Platzhalter bei der Verarbeitung des Widgets ersetzt werden soll.

Beispiel:

```
1 <div data-sw-config="mapping"
2       data-swp-id="language"
3       data-swp-value="en">
4 </div>
```

D.1.5 Proxy

Damit SemwidgJS mit einem externen SPARQL-Endpoint kommunizieren kann, ist es erforderlich, dass dieser das CORS-Protokoll (Cross-Origin Resource Sharing)[1] unterstützt, um somit Same-Origin-Policy des Browsers umgehen zu dürfen. Da eine Vielzahl der SPARQL-Endpoints der LOD-Cloud CORS nicht implementieren, erlauben Proxy-Konfigurationselemente die Abfrage über einen anderen Server zu leiten, welcher CORS implementieren. Wird ein Endpoint-Konfigurationselemente mit einem Proxy-Konfigurationselemente verknüpft, wird automatisch jede Abfrage, die an diesen Endpoint gerichtet ist, über den Proxy umgeleitet.

Objekt-Klasse:

Attribut	Wert
`data-sw-config`	`proxy`

Attribute:

Attribut	Pflicht	Beschreibung
`data-swp-id`	*	Ein einzigartiger Wert, der dieses Konfigurationselement identifiziert.
`data-swp-value`	*	Die URL des Proxy-Servers.

Beispiel:

```
1  <div data-sw-config="proxy"
2       data-swp-id="myProxy"
3       data-swp-url="https://example.com/proxy.php">
4  </div>
```

Zusätzliche Informationen

Wenn die Abfrage eines SPARQL-Endpoints die Nutzung eines Proxyservers erfordert, kann SemwidgJS diese Abfrage über einen solchen Server leiten. Die umgeleitete Abfrage ähnelt der regulären Abfrage an den SPARQL-Endpoint. Zusätzlich wird jedoch noch ein Parameter `url` an die Liste der POST-Parameter anfügt. Dieser Parameter enthält die Url des SPARQL-Endpoints, an den der Proxyserver die Anforderung weiterleiten soll.

[1] `https://www.w3.org/TR/cors/`

D.2 Widgets

Jedes Widget wird durch mehrere data-*-Attribute eines HTML-Elements definiert. In der Regel ist dieses HTML-Element ein `div`- oder `span`-Element. Je nach Widget-Klasse kann es auch sinnvoll sein, andere Elemente als Basis zu wählen (z. B. `h1`-Elemente), um deren vordefinierten CSS-Regeln zu erben. Die Klasse des Widgets wird durch das data-*-Attribut `data-sw-widget` und die entsprechende Widget-Bezeichnung (vgl. Anhang D.2.4) als Wert definiert (z. B. `SwText`, `SwImage`, `SwOpenStreetMap`). Widget-Eigenschaften werden ebenfalls durch data-*-Attribute definiert. Welche Eigenschaften verfügbar sind, hängt vom der verwendeten Widget-Klasse ab. Eine Reihe von Standardeigenschaften ist jedoch für jedes Widget verfügbar.

D.2.1 Standardeigenschaften von Widgets

Jedes Widget verfügt über die nachfolgend aufgelisteten data-*-Attribute, welche einem bestimmten Eigenschaftentyp besitzen. Dieser Eigenschaftentyp wird in Anhang D.2.2 näher beschrieben. Im Gegensatz zu den Konfigurationselementen, gibt es keine verpflichtend anzugebenden Attribute.

Attribut	Typ	Beschreibung
`data-swp-id`	`fixed`	Ein einzigartiger Wert, der dieses Widget identifiziert.
`data-swp-endpoint`	`static`	Der SPARQL-Endpoint, der genutzt werden soll, wenn weder eine SemwidgQL-Abfrage noch die dazugehörige zusätzliche Query-Eigenschaft einen Endpoint vorgibt.
`data-swp-styleclass`	`query`	CSS-Klassenname(n), die in dem erzeugten HTML-Element hinzugefügt werden soll(en).
`data-swp-style`	`query`	Eine Zeichenkette, die dem erzeugten HTML-Element als Wert für das style-Attribut hinzugefügt werden soll.
`data-swp-title`	`query`	Eine Zeichenkette, die dem erzeugten HTML-Element als Wert für das title-Attribut hinzugefügt werden soll. Diese Zeichenkette wird von den meisten Browsern als Mouse-Over-Text angezeigt.
`data-swp-other`	`query`	Eine Zeichenkette, die alle anderen möglichen HTML-Attribute und ihre Werte beschreibt. Der Wert dieses `data-swp-other`-Attributs wird so wie er eingegeben wurde in das HTML-Element eingefügt. Bei Bedarf können über dieses Attribut auch zusätzliche JavaScript-Befehle eingefügt werden.
`data-swp-refresh`	`static`	Das Intervall, mit dem die abgerufenen Daten automatisch aktualisiert werden sollen. SemwidgJS nutzt zum Parsen des Intervalls die JavaScript-Bibliothek Later.js.
`data-swp-limit`	`static`	Die Anzahl der maximal zurückzuliefernden Ergebnisse bei der Abfrage eines SPARQL-Endpoints. Der Wert wird an die SemwidgQL- oder SPARQL-Abfrage angehängt oder ersetzt ihn.
`data-swp-cache`	`static`	Die Dauer, für die die abgefragten Daten zwischengespeichert werden sollen. Als Einheiten können `s(econds)`, `m(inutes)`, `h(ours)` und `d(ays)` genutzt werden. Eine Kombination aus mehren Werten ist ebenfalls möglich. Diese werden entsprechend umgerechnet und aufaddiert. Minimum: `1 minute`, Maximum: `7 days`.
`data-swp-value`	`query`	Der Wert, der standardmäßig die Daten beschreibt, welche das Widget anzeigen soll. Dieses Attribut enthält in der Regel eine SemwidgQL- oder SPARQL-Abfrage. In den meisten Fällen ist es ausreichend lediglich diese Eigenschaft zu nutzen, um die gewünschte Datenpräsentation zu erzeugen.

D.2.2 Eigenschaftentypen von Widgets

Widget-Eigenschaften werden anhand von vier Eigenschaftentypen klassifiziert, und beschreiben, wie eine Eigenschaft verarbeitet wird. Jede Eigenschaft des Typs `query` erzeugt automatisch zusätzliche weitere Eigenschaften, die in Anhang D.2.3 beschrieben werden.

Typ	Beschreibung
`fixed`	Die Werte von Eigenschaften dieses Typs können nachträglich nicht mehr verändert werden. Dieser Eigenschaftentyp wird beispielsweise für die IDs (`data-swp-id`) von sämtlichen SemwidgJS-Elementen genutzt.
`static`	Die Werte von Eigenschaften dieses Typs werden nur als statische Zeichenketten interpretiert. Der Wert kann jedoch während der Laufzeit geändert werden und andere Elemente werden über diese Änderung informiert. Dieser Eigenschaftentyp wird beispielsweise für die Verknüpfung über IDs zwischen den verschiedenen SemwidgJS-Elementen genutzt..
`selector`	Selektoren können von interaktiven Elementen genutzt werden, die es ermöglichen, die Daten von anderen Widgets zu manipulieren. Die Werte von Eigenschaften dieses Typs beschreiben, welches konkrete Element und welche Eigenschaft manipuliert werden soll.
`query`	Die Werte von Eigenschaften dieses Typs können SemwidgQL- und SPARQL-Abfragen, aber auch statische Zeichenketten oder eine beliebige Kombination aus diesen enthalten. Dies ist der Standardtyp der meisten Eigenschaften. Um zu kennzeichnen, dass Teile eines Attributwerts als SemwidgQL- oder SPARQL-Abfrage zu interpretieren sind, müssen diese in einfache geschweifte Klammern (SemwidgQL) oder in doppelte geschweifte Klammern (SPARQL) eingeschlossen werden. Eigenschaft dieses Typs erzeugt automatisch zusätzliche weitere Eigenschaften, die in Anhang D.2.3 beschrieben werden.

D.2.3 Zusätzliche Query-Eigenschaften

Sämtliche Eigenschaften des Typs `query` erzeugen automatisch eine Reihe weiterer Eigenschaften. Diese Eigenschaften verwenden den selben Eigenschaftennamen wie die übergeordnete Eigenschaft, welche diese erzeugt hat, zuzüglich eines Suffixes (z. B. `data-swp-value-endpoint`). Alle zusätzlichen Query-Eigenschaften sind vom Eigenschaftentyp `static`.

Attribut	Beschreibung
`data-swp-*-endpoint`	Der SPARQL-Endpoint, der genutzt werden soll, wenn eine (möglicherweise genutzte) SemwidgQL-Abfrage durch ihr Resource-Element keinen Endpoint vorgibt.
`data-swp-*-order`	Da SemwidgQL-Abfragen keine Sortierung der Ergebnisliste zulässt, lässt sich eine Sortierung nachträglich anhand dieser Eigenschaft festlegen. Die Syntax für den Wert der Eigenschaft ähnelt der Syntax von SQL, in der zuerst die Variable und dann dessen Sortierreihenfolge (aufsteigen `asc` oder absteigend `desc`) angegeben werden. Da die Variablennamen innerhalb der SPARQL-Abfrage automatisch generiert werden, sind sie dem Nutzer jedoch nicht bekannt. Daher lassen sich die Variablen nicht über ihren Namen, sondern über ihren Index innerhalb der SPARQL-Ergebnisliste auswählen. Zur einfacheren Anwendung wurden hier die beiden Schlüsselwörter `first` und `last` eingeführt. Zusätzlich lassen sich vereinfachte arithmetische Ausdrücke (`+-*/`) einfügen, die auch mit den beiden Schlüsselwörtern kombiniert werden können. Wird keine Sortierreihenfolge angegeben, wird automatisch aufsteigend sortiert. Mittels Kommata getrennt, können mehrere Variablen zur Sortierung hintereinander angegeben werden. Beispiel: `last - 1 asc, last desc`.
`data-swp-*-selector`	Sämtliche SemwidgJS-Widgets besitzen Zugriff auf eine Hilfsfunktion, die ihnen die Auswahl des anzuzeigenden Wert aus der Ergebnisliste vereinfacht. Standardmäßig wählt diese Funktion immer das erste Ergebnis der Ergebnisliste und daraus die Eigenschaft mit dem höchsten Index – also die Variable, die in der (übersetzten) SPARQL-Abfrage als letztes im SELECT-Ausdruck aufgeführt wurde. Soll dieses Verhalten geändert werden, kann über diese Eigenschaft eine Auswahlfunktion definiert werden. Dieser wird die SPARQL-Ergebnisliste in vorverarbeiteter Form übergeben und muss einen String zurückliefern. Der Quelltext der Auswahlfunktion kann entweder direkt innerhalb dieser Eigenschaft definiert werden oder es kann auf eine global verfügbare Funktion verwiesen werden.
`data-swp-*-transform`	Soll das Ergebnis vor der Anzeige manipuliert werden, kann mittels dieser Eigenschaft eine Transformationsfunktion definiert werden. Eine solche Funktion kann beispielsweise dafür genutzt werden einen abgefragten Datumswert im ISO-Format (z. B. »1984-07-12«) in eine lokalisierte Form zu überführen (»12.07.1984«). Auch hier kann der Quelltext der Funktion entweder direkt innerhalb dieser Eigenschaft definiert werden oder es kann auf eine global verfügbare Funktion verwiesen werden.
`data-swp-*-refresh`	Das Intervall, mit dem die abgerufenen Daten automatisch aktualisiert werden sollen. SemwidgJS nutzt zum Parsen des Intervalls die JavaScript-Bibliothek Later.js.
`data-swp-*-limit`	Die Anzahl der maximal zurückzuliefernden Ergebnisse bei der Abfrage eines SPARQL-Endpoints. Der Wert wird an die SemwidgQL- oder SPARQL-Abfrage angehängt oder ersetzt ihn.
`data-swp-*-cache`	Die Dauer, für die die abgefragten Daten zwischengespeichert werden sollen. Als Einheiten können `s(econds)`, `m(inutes)`, `h(ours)` und `d(ays)` genutzt werden. Eine Kombination aus mehren Werten ist ebenfalls möglich. Diese werden entsprechend umgerechnet und aufaddiert. Minimum: `1 minute`, Maximum: `7 days`.

D.2.4 Widget-Klassen

Dieser Abschnitt beschreibt die innerhalb der SemwidgJS-Bibliothek vordefinierten Widget-Klassen. Diese lassen sich in GUI-Elemente zur Datenpräsentation sowie Diagramme zur Datenvisualisierung kategorisieren.

Objekt-Klasse:

Attribut	Wert
`data-sw-widget`	Widget-Bezeichnung der Widget-Klasse, die instanziiert werden soll.

GUI-Elemente (Datenpräsentation)

Ausgabe		
Name	Widget-Bezeichnung	Beschreibung
Debug	`SwDebug`	Gibt die vollständige Ergebnisliste einer Abfrage inklusiver aller Eigenschaften in Form einer Tabelle aus. Zusätzlich werden die ursprüngliche Eingabe und alle daraus abgeleiteten SPARQL-Abfragen angezeigt.
Image	`SwImage`	Interpretiert ein Abfrageergebnis als URL einer Bilddatei und zeigt dieses an.
Image Grid	`SwImageGrid`	Interpretiert alle Abfrageergebnisse als URLs von Bilddateien und zeigt diese an.
(Hyper-) Link	`SwLink`	Erzeugt aus einer abgefragten URL und einem Text einen HTML-Hyper-Link.
Map	`SwOpenStreetMap`	Zeigt einen Standort und/oder eine Region anhand der abgefragten Koordinaten auf einer Weltkarte an.
SPARQL Result Table	`SwResultTable`	Gibt die vollständige Ergebnisliste einer Abfrage inklusiver aller Eigenschaften in Form einer Tabelle aus.
Simple List	`SwSimpleList`	Erzeugt eine Liste aus den abgerufenen Daten.
Table	`SwTable`	Gibt ausgewählte Eigenschaften der Ergebnisliste einer Abfrage in Form einer Tabelle aus.
Text	`SwText`	Zeigt ein Abfrageergebnis als Text an.

Eingabe		
Name	Widget-Bezeichnung	Beschreibung
Drop-Down List	`SwDropDown`	Erzeugt ein Drop-Down-Element aus einem Abfrageergebnis. Das ausgewählte Eintrag des Elements kann automatisch an eine weiteres SemwidgJS-Element weitergeleitet werden.
Text Input Field	`SwInputText`	Erzeugt ein einfaches Eingabefeld mit einem zusätzlichen »Ok«-Button. Der Wert der Eingabe kann vordefiniert werden und der geänderte Wert kann automatisch an ein weiteres SemwidgJS-Element weitergeleitet werden.

Diagramme (Datenvisualisierung)

einzelne Datenreihe		
Name	Widget-Bezeichnung	Beschreibung
Doughnut Chart	`ChartsDoughnutChart`	Stellt die abgefragten Daten in Form eines Ringdiagramms dar. Die abgefragten Daten müssen numerische Werte sein.
Pie Chart	`ChartsPieChart`	Stellt die abgefragten Daten in Form eines Kreisdiagramms dar. Die abgefragten Daten müssen numerische Werte sein.
Polar Area Chart	`ChartsPolarAreaChart`	Stellt die abgefragten Daten in Form eines Polar-Area-Diagramm dar. Die abgefragten Daten müssen numerische Werte sein.

mehrere Datenreihen		
Name	Widget-Bezeichnung	Beschreibung
Bar Chart	`ChartsBarChart`	Stellt die abgefragten Daten in Form eines Säulendiagramms dar. Die abgefragten Daten müssen numerische Werte sein. Das Widgets kann mehrer Datenkategorien gleichzeitig anzeigen.
Line Chart	`ChartsLineChart`	Stellt die abgefragten Daten in Form eines Liniendiagramms dar. Die abgefragten Daten müssen numerische Werte sein. Das Widgets kann mehrer Datenkategorien gleichzeitig anzeigen.
RadarChart	`ChartsRadarChart`	Stellt die abgefragten Daten in Form eines Netzdiagramms dar. Die abgefragten Daten müssen numerische Werte sein. Das Widgets kann mehrer Datenkategorien gleichzeitig anzeigen.

D.3 Templates

Templates ermöglichen die Wiederverwendung von Kompositionen aus Widgets und anderem HTML-Inhalten. Eine Template-Definition enthält HTML-Quelltext, welche wiederum Widgets enthalten kann. Teile diese Inhalte können durch Platzhalter ersetzt werden. Durch Instanziierung dieser Template-Definitionen mit entsprechenden Parameterwerten für die Platzhalter können Template-Instanzen erstellt werden.

D.3.1 Template-Definitionen

Wie auch die Konfigurationselemente, werde Template-Definitionen mittels generischer HTML-Elemente (`div`, `span`) erzeugt, erhalten im Gegensatz zu diesen aber nicht nur Attribute, sondern auch einen inneren HTML-Text. Innerhalb der Attribute wird eine kommaseparierte Liste mit Parameternamen angegeben, die später bei der Instanziierung des Templates genutzt werden können. Der Inhalt des Definitionselements kann aus beliebigen HTML-Text bestehen und natürlich auch Widgets enthalten. Teile des Inhalts können durch Platzhalter ersetzt werden, deren Bezeichner den zuvor definierten Parameternamen entsprechen und in doppelte Prozentzeichen eingeschlossen sein müssen.

Objekt-Klasse:

Attribut	Wert
`data-sw-config`	`template`

Attribute:

Attribut	Pflicht	Beschreibung
`data-swp-id`	*	Ein einzigartiger Wert, der diese Template-Definition identifiziert.
`data-swp-parameter`		Eine kommaseparierte Liste von Parameternamen. Diese Parameternamen können, eingeschlossen in doppelte Prozentzeichen, innerhalb des inneren Html-Textes als Platzhalter genutzt werden. Bei der Instanziierung eines Templates werden sie durch die dort übergebenen Werte ersetzt.

Beispiel:

```
1  <div data-sw-config="template"
2       data-swp-id="myTemplateDefinition"
3       data-swp-parameter="resource, language"
4       style="display: none;">
5
6       <div data-sw-widget="SwText"
7            data-swp-value="{%%resource%%.rdfs:label(@lang = '%%language%%'}">
```

```
8         </div>
9
10    </div>
```

Zusätzliche Informationen

Es wird empfohlen, die Sichtbarkeit von Template-Definitionen mittels CSS-Regeln auszuschalten (`style="display: none;"`). SemwidgJS wird dies zwar automatisch ebenfalls tun, doch können längere Ladezeiten dazu führen, dass dies erst lange nach dem ersten Anzeigen der Webseite geschieht und dem Endanwender bis dahin Fragmente der Template-Definition angezeigt werden.

D.3.2 Template-Instanzen

Template-Instanzen werden ähnlich wie Widgets deklariert. Anstatt auf einer Widget-Klasse, basieren sie jedoch auf einer Template-Definition, welche anhand ihrer ID ausgewählt wird. Die Attribute, die bei der Deklaration eine Template-Instanz angegeben werden können, entsprechen den Parameternamen, die bei der Erstellung der entsprechenden Template-Definition angegeben wurden. Es existieren keine verpflichtend anzugebenden Attribute.

Objekt-Klasse:

Attribut	Wert
`data-sw-template`	ID der Template-Definition, die instanziiert werden soll.

Attribute:

Attribut	Beschreibung
`data-swp-id`	Ein einzigartiger Wert, der diese Template-Instanz identifiziert.
`data-swp-*`	Ein Wert, der den Platzhalter mit dem selben Namen in der Template-Definition ersetzen soll. (`*` = Parametername)

Beispiel:

```
1  <div data-sw-template="myTemplateDefinition"
2       data-swp-id="myTemplateInstance"
3       data-swp-resource="dbr:Albert_Einstein"
4       data-swp-language="en">
5  </div>
```

⇓

```
1  <div data-sw-template="myTemplateDefinition"
2       data-swp-id="myTemplateInstance"
3       data-swp-resource="dbr:Albert_Einstein"
4       data-swp-language="en">
5
6       <div data-sw-widget="SwText"
7            data-swp-value="{dbr:Albert_Einstein.rdfs:label(@lang = 'en'}">
8       </div>
9
10 </div>
```

ANHANG E

Empirische Nutzerstudie zur Evaluation der Gebrauchstauglichkeit von SemwidgED

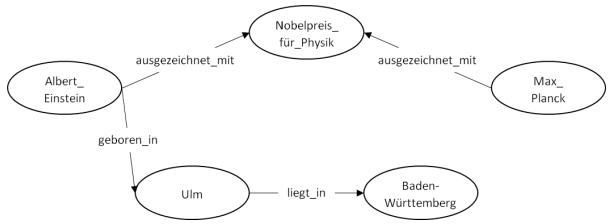
Ziel der Studie und Anwendungszweck

In dieser Studie soll die Usability eines Online-Editors evaluiert werden. Mit diesem Editor können Daten aus sogenannten SPARQL-Endpoints in normale Websites eingebunden werden. Im Gegensatz zu Daten in relationalen Datenbanken, sind die Daten in SPARQL-Endpoints nicht in Tabellenform gespeichert, sondern in Graphform.

Die Daten im SPARQL-Endpoint beschreiben Aussagen in „Subjekt-Prädikat-Objekt"-Form. Z. B.:

Subjekt	*Prädikat*	*Objekt*
Albert_Einstein	ausgezeichnet_mit	Nobelpreis_für_Physik
Albert_Einstein	wurde_geboren_in	Ulm
Ulm	liegt in	Baden-Württemberg
Max_Planck	ausgezeichnet_mit	Nobelpreis_für_Physik

Oder in Graphform:



Die Subjekte und Objekte werden in der Community in der Regel als Resources und die Prädikate als Properties bezeichnet.

Als Abfragesprache für diese Daten nutzt der Editor die Sprache SemwidgQL. Die Syntax der Sprache ähnelt der von Objektorientieren Programmiersprachen (Java, C# etc.). Stellen sie sich die Resources des SPARQL-Endpoints als Objekte in einem Java-Programm vor und die Properties als Eigenschaften dieser Objekte.

Wenn Sie nun eine Variable „Albert_Einstein" in ihrem Java-Programm haben und ausgeben möchten, womit er ausgezeichnet wurde, würden Sie diesen Wert abfragen mittels:

```
Albert_Einstein.ausgezeichnet_mit
```

Und auf dieselbe Weise können Sie mit SemwidgQL die Daten des SPARQL-Endpoints abfragen. Diese Anfragen können zusätzlich noch verkettet werden und Werte können bei Bedarf anhand der verknüpften Sprache gefiltert werden (siehe SemwidgED-Handout).

Im Rahmen der Evaluation nutzen wir den SPARQL-Endpoint „DBpedia" welcher strukturierte Daten aus Wikipedia enthält. Insgesamt umfasst der Endpoint ca. 630 Million Aussagen in „Subjekt-Prädikat-Objekt"-Form.

Abbildung E.1: Handout aus der Nutzerstudie zur Evaluation der Gebrauchstauglichkeit von SemwidgED. Der Text enthält eine Einführung in die Linked-Data-Thematik und SemwidgQL.

SemwidgED

SemwidgEd

Erste Schritte

SemwidgJS konfigurieren:

- ➤ „Config Wizard“
- ➤ Endpoint aus Liste auswählen und hinzufügen
- ➤ Ressourcen suchen und hinzufügen

Widgets

Widgets erstellen:

- ➤ „Add: Widget“
- ➤ passenden Widget-Typ auswählen
- ➤ SemwidQL-Anfrage für gewünschtes Attribut (in den meisten Fällen „value“) eintragen

SemwidgQL-Anfragen erstellen:

Variante 1:

- ➤ Cursor in Eingabefeld platzieren
- ➤ „Start new SemwidgQL query“
- ➤ Anfrage formulieren (dabei Auto-Complete nutzen)

Variante 2:

- ➤ Cursor in Eingabefeld platzieren
- ➤ „Start new SemwidgQL query“
- ➤ Query-Editor über Lupensymbol öffnen
- ➤ Abschnitte der Anfrage einzeln formulieren (dabei Auto-Complete nutzen)
- ➤ neue Abschnitte mittel Zahnradsymbol hinzufügen

Widgets stylen:

- ➤ im Widget-Dialog zu „Basic Properties“ wechseln
- ➤ für Attribut „Style“ CSS-Code eingeben

Templates

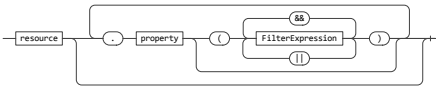
Mehrere Widgets in ein Template umwandeln:

- ➤ Widgets markieren (nicht die Konfigurations-elemente)
- ➤ „Create Template“
- ➤ „Template Modifications“
- ➤ Resource durch Parameter (z. B. %%value%%) ersetzen

Template-Instanzen erstellen:

- ➤ „Add: Template Instance“
- ➤ passendes Template auswählen
- ➤ Wert für Parameter eingeben (z. B. eine Ressource)

SemwidgQL-Anfragen



Struktur einer Anfrage:

```
ex:Resource1.ex:Prop1.ex:Prop2
```

Inverse Properties

Umdrehen der Subjekt-Prädikat-Objekt-Relation. Liefert das Subjekt dieser Relation zurück, statt des Objekts.

```
ex:Resource1.^ex:Prop1
```

Filter-Ausdrücke:

Filterung der Sprache: `(@lang = 'de')`

Abbildung E.2: Handout aus der Nutzerstudie zur Evaluation der Gebrauchstauglichkeit von SemwidgED. Das Handout enthält eine Übersicht über die Funktionen von SemwidgED und die Syntax von SemwidgQL.

Tabelle E.1: Aufgaben der Evaluation.

Aufgabe 1:	Schauen Sie sich das Tutorial-Video an.
Aufgabe 2:	Erstellen Sie eine Website über drei ihrer Lieblingsschauspieler mit Daten von DBpedia. Bewerten Sie die Funktionen des Editors nach jeder Teilaufgabe anhand drei kurzer Aussagen:
a:[a]	Konfigurieren Sie die Website so, dass sie Daten von DBpedia abruft und legen Sie ein Resource-Element für einen ihrer Lieblingsschauspieler an.
b:[a]	Erstellen Sie ein Widget, das den deutschen Beschreibungstext (*comment*) dieses Schauspielers anzeigt.
c:[a]	Erstellen Sie ein Widget, das ein Bild (*thumbnail*) dieses Schauspielers mit einer maximalen Höhe von 200 Pixeln anzeigt.
d:[a]	Erstellen Sie ein Widget, das das deutschsprachige Label des Geburtsorts (*birthplace*, *label*) anzeigt.
e:[a]	Erstellen Sie ein Widget, das den Geburtsort (*birthplace*, *lat*/*long*) in Form einer Karte anzeigt (*Zoomstufe 5*).
f:[a]	Wandeln Sie die bisher erstellte Widget-Sammlung in ein Template um. Ersetzen Sie dabei den Schauspieler durch einen variablen Parameter.
g:[a]	Erstellen Sie drei Instanzen des Templates mit drei verschiedenen Schauspielern.
Aufgabe 3:	Füllen Sie den Online-Fragebogen aus.

[a] (Nach Beendigung der Teilaufgabe füllten die Probanden jeweils einen ASQ aus; vgl. Tabelle E.2)

Tabelle E.2: After Scenario Questionnaire (ASQ) nach Lewis (1991).

	Starke Zustimmung						Starke Ablehnung
	1	2	3	4	5	6	7
1) Insgesamt bin ich damit zufrieden, wie leicht die Aufgaben in diesem Szenario zu lösen war.	☐	☐	☐	☐	☐	☐	☐
2) Insgesamt bin ich damit zufrieden, wie viel Zeit ich für die Lösung der Aufgaben in diesem Szenario aufwenden musste.	☐	☐	☐	☐	☐	☐	☐
3) Insgesamt bin ich mit den unterstützenden Informationen bei der Bearbeitung des Szenarios zufrieden.	☐	☐	☐	☐	☐	☐	☐

Tabelle E.3: System Usability Scale (SUS) nach Brooke (1996).

	stimme gar nicht zu				stimme voll zu
	1	2	3	4	5
1) Ich kann mir sehr gut vorstellen, das System regelmäßig zu nutzen.	☐	☐	☐	☐	☐
2) Ich empfinde das System als unnötig komplex.	☐	☐	☐	☐	☐
3) Ich empfinde das System als einfach zu nutzen.	☐	☐	☐	☐	☐
4) Ich denke, dass ich technischen Support brauchen würde, um das System zu nutzen.	☐	☐	☐	☐	☐
5) Ich finde, dass die verschiedenen Funktionen des Systems gut integriert sind.	☐	☐	☐	☐	☐
6) Ich finde, dass es im System zu viele Inkonsistenzen gibt.	☐	☐	☐	☐	☐
7) Ich kann mir vorstellen, dass die meisten Leute das System schnell zu beherrschen lernen.	☐	☐	☐	☐	☐
8) Ich empfinde die Bedienung als sehr umständlich.	☐	☐	☐	☐	☐
9) Ich habe mich bei der Nutzung des Systems sehr sicher gefühlt.	☐	☐	☐	☐	☐
10) Ich musste eine Menge Dinge lernen, bevor ich mit dem System arbeiten konnte.	☐	☐	☐	☐	☐

Tabelle E.4: ErgoNorm-Fragebogen nach Dzida et al. (2000). Zu jedem Item konnte eine der Optionen »Ja«, »Nein« und »Frage trifft nicht zu« gewählt werden. Bei negativ bewerteten Item sollte eine Begründung im Freitext angegeben werden. Zusätzlich konnte dann die Option »Ich empfinde dies als sehr störend« markiert werden. Die ursprüngliche Zeitform der Fragen wurde vom Präsens ins Präteritum übersetzt.

Aufgaben-angemessenheit	AA1)	Enthielt das Programm alle für Ihre Aufgabe benötigten Funktionen?
	AA2)	Mussten Sie Eingaben oder Dialogschritte machen, die eigentlich überflüssig waren?
	AA3)	War es Ihnen möglich, das wiederholte Eingeben von Daten oder Texten zu vereinfachen?
	AA4)	Fanden Sie, dass der erforderliche Aufwand für Ihr Arbeitsergebnis jeweils angemessen war?
	AA5)	Hatten Sie das Gefühl, dass Sie Arbeiten machen mussten, die besser das Programm erledigen soll?
	AA6)	Mussten Sie Werte und Texte eingeben, die der Computer eigentlich wissen könnte?
	AA7)	Mussten Sie sich mit Umwegen oder Tricks behelfen, um Ihre Arbeitsergebnisse so zu erzielen, wie Sie diese haben möchten?
	AA8)	Fanden Sie in dem Programm Hilfetexte, die Ihnen auch tatsächlich weitergeholfen haben?
	~~AA9)~~	~~Passte das Programm zu Ihren Formularen und bisherigen Formaten?~~[a]
Selbstbeschreibungs-fähigkeit	SF1)	Waren die Informationen, die zur Erledigung der Aufgabe notwendig waren, auf dem Bildschirm übersichtlich verfügbar?
	SF2)	Konnten Sie bei der Arbeit mit dem Programm erkennen, welche Eingabe als nächstes von Ihnen erwartet wurden?
	SF3)	Waren die Meldungen des Systems für Sie immer verständlich?
	SF4)	Wurden Sie vor Aktionen, die nicht rückgängig gemacht werden können, von der Software gewarnt?
	~~SF5)~~	~~Half Ihnen die Hilfefunktion wirklich weiter, wenn einmal ein Dialogschritt oder Menüpunkt nicht ganz klar war?~~[a]
	SF6)	Mussten Sie oft das SemwidgED-Handout konsultieren, um weiterarbeiten zu können?[b]
Steuer-barkeit	ST1)	Konnten Sie Ihre Arbeitsschritte in der Reihenfolge erledigen, die Ihnen am sinnvollsten erschien?
	ST2)	Machte das Programm manchmal etwas, ohne dass Sie es zu dem Zeitpunkt wollten?
	ST3)	Konnten Sie bei Bedarf eine Aufgabe unterbrechen und später wieder fortsetzen, ohne alles neu eingeben zu müssen?
	ST4)	Konnten Sie einen Arbeitsschritt wieder zurücknehmen, wenn es für Ihre Aufgabenerledigung zweckmäßig war?
	ST5)	Fühlten Sie sich in Ihrem Arbeitstempo durch das Programm manchmal gebremst, z. B. durch zu lange Wartezeiten?
Erwartungs-konformität	EK1)	Fanden Sie Menüpunkte oder Funktionen dort, wo sie Ihrer Meinung nach auch sein sollten?
	EK2)	Waren Sie sich bei Wartezeiten immer noch sicher, ob das Programm weiterarbeitet?
	EK3)	Waren Sie manchmal überrascht, wie das Programm auf Ihre Eingabe reagierte?
Fehler-toleranz	FT1)	Bekamen Sie bei fehlerhaften Eingaben Korrekturhinweise?
	FT2)	Konnten Sie die Folgen einer fehlerhaften Eingabe mit geringem Aufwand beheben?
	FT3)	Arbeitete das Programm während der Ausführung Ihrer Aufgabe immer stabil und zuverlässig?
Individualisierbarkeit	IN1)	Konnten Sie am Computer alles so einstellen, dass Ihnen das Lesen und Arbeiten leichter fiel?
Lernförderlichkeit	LF1)	Ermöglichte Ihnen das Programm, auch einmal etwas gefahrlos auszuprobieren?

[a] (Die Frage wurde nicht gestellt, da sie nicht zum Szenario passte oder die abgefragte Funktion nicht implementiert war)
[b] (Die Frage wurde an das Szenario angepasst; Ursprünglich wurde nach Kollegen und dem Handbuch gefragt)

DOI: 10.17185/duepublico/77037
URN: urn:nbn:de:hbz:465-20230523-144347-6