DIPLOMARBEIT

Performante Anfragen in partizipativen semantischen Portalen

von

Sebastian Döweling

eingereicht am 10.11.2006 beim
Institut für Angewandte Informatik
und Formale Beschreibungsverfahren
der Universität Karlsruhe

Referent: Prof. Dr. Rudi Studer
Betreuer: Dipl.-Inform. Max Völkel

Anschrift:
Goethestr. 2
76135 Karlsruhe

Zusammenfassung

Portale sind im Internet von großer Bedeutung. Insbesondere partizipative Portale wie die Online-Enzyklopädie *Wikipedia* oder das Video-Portal *YouTube* genießen derzeit hohe Aufmerksamkeit.

Ein Manko dieser Portale ist jedoch die rein visuelle Darstellung, welche von Maschinen nicht verwendet werden kann. Dieses Problem wird von den partizipativen, semantischen Portalen adressiert. Diese erweitern partizipative Portale um Elemente des *Semantic Web*.

Im Rahmen dieser Arbeit soll das Szenario partizipativer, semantischer Portale am Anwendungsfall der *Semantic Wikipedia* untersucht werden. Die Probleme, die bei der Erweiterung eines partizipativen Portals wie *Wikipedia* zum partizipativen, semantischen Portal entstehen, werden dabei analysiert und eine Lösung wird entwickelt.

Diese Lösung abstrahiert vom konkreten Anwendungsfall und eignet sich allgemein als Zwischenspeicher für SPARQL-Anfragen an alle RDF-Datenspeicher, welche RDFS-kompatible Ontologiesprachen einsetzen.

Inhaltsverzeichnis

1 Motivation

Portalen im Internet wird bereits seit Ende der 90er Jahre hohe Bedeutung beigemessen; so weist etwa die IT-Zeitschrift Computerwoche bereits ab 1998 auf die hohe Marktbedeutung hin ([Com98], [Com00a] und [Com00b]) und auch die *Gartner Group* widmet diesem Thema einen Artikel ([Bar99]). Die Entwicklung hat sich bis zum heutigen Tage dergestalt fortgesetzt, dass 15 der 20 meistbesuchten Seiten Portale sind (laut [AI06]). Bei den verbleibenden 5 handelt es sich um Suchmaschinen, welche größtenteils personalisierbare Suchseiten anbieten, die sich etwa mit Nachrichten, E-Mail-Diensten oder Aktienkursen zum benutzerspezifischen Portal ergänzen lassen.

Partizipative Portale Gerade im Rahmen der kürzlichen Prägung des Begriffs „*Web 2.0*“ [O'R05] wurde dieser Effekt noch einmal verstärkt. So entstanden Projekte wie Flickr[1], MySpace[2] oder YouTube[3], die allesamt nicht nur Inhalte in einem Portal aggregieren, sondern diese Inhalte vor allem durch ihre Benutzer erst erhalten und deren Information sich teilweise durch gemeinsames Hinzufügen sogenannter *Tags* durch jene bestimmt. Diese Portale werden als partizipative Portale bezeichnet.

Wikis Eine wichtige Klasse der partizipativen Portale sind die *Wikis*. Dieser Begriff wurde durch die von 1994 bis -95 von Ward Cunningham entwickelte WikiWikiWeb-Software [LC01] geprägt. Diese Systeme ermöglichen es allen Nutzer, ihr Wissen einzubringen und gemeinsam an der Schaffung einer Wissensbasis zu partizipieren.

Dieses Prinzip wurde unter anderem von der freien Enzklopädie Wikipedia[4] aufgegriffen, die zunächst auf der UseModWiki-Software[5] basierte und später mit *Mediawiki*[6] eine eigens entwickelte Grundlage erhielt, die den steigenden Anforderungen gewachsen war.

Mit etwa 315 Millionen von Google indizierten Referenzen und, laut Alexa Traffic Rankings, einem Platz unter den 20 meist aufgerufenen Seiten kann Wikipedia als einer der derzeit wichtigste Anwendungsfälle für partizipative Portale gelten.

Semantische Portale Ein generelles Manko der Informationspräsentation im World Wide Web der aktuellen Generation ist die rein visuelle Darstellung von Informationen. Während sich die Bedeutung dieser Informationen dem menschlichen Leser einfach erschließt, fehlen weitere Daten, um diese auch für maschinelle Interpretation zu erschließen. Diese Lücke schließt das Semantic Web, das bereits Teil der ursprünglichen Vision des World Wide Web Erfinders Tim Berners-Lee war ([BLHL01], [BL03], [BL00b]).

Eine Vielzahl von Projekten ([VS06]) entwickelt derzeit Lösungen, welche Portale um semantische Elemente anreichern. Unter diesen soll hier besonderer Augenmerk auf das Projekt *Semantic Wikipedia* ([VKV+06a]) gerichtet werden.

[1] Flickr, `http://www.flickr.com/`
[2] MySpace, `http://www.myspace.com/`
[3] YouTube, `http://www.youtube.com/`
[4] Wikipedia, `http://www.wikipedia.org`
[5] UseMod, `http://www.usemod.com/cgi-bin/wiki.pl`
[6] Mediawiki, `http://www.mediawiki.org`

Im Rahmen dieses Projektes entsteht eine Erweiterung der klassischen *MediaWiki*-Software, auf der die *Wikipedia* basiert. Diese nennt sich *Semantic MediaWiki* und ermöglicht es, *Wiki*-Seiten semantische Informationen hinzuzufügen. Dabei wird vor allem der Tatsache Rechnung getragen, dass es sich üblicherweise bei den Editoren eines *Wikis*, speziell eines gemeinhin bekannten wie *Wikipedia*, nicht um wissenschaftlich im Bereich des *Semantic Web* geschulte Personen handelt.

Die *Semantic Wikipedia* ermöglicht es, Anfragen über die semantischen Informationen in die Portalseiten zu integrieren und aus diesen automatisch Listen oder Tabellen zu erzeugen.

1.1 Problemstellung

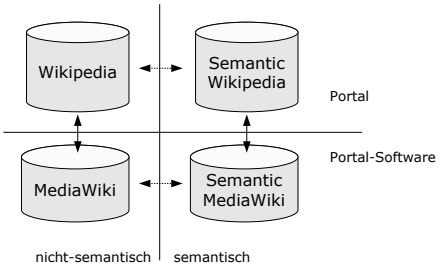

Abbildung 1: Portale und Portal-Software

Am Beispiel der *Semantic Wikipedia* soll untersucht werden, inwiefern sich die Möglichkeit, semantische Informationen zu speichern und durch integrierte semantische Suchanfragen in Portalseiten einzubinden, auf ein bestehendes partizipatives Portal wie *Wikipedia* negativ auswirkt.

Die Architektur der beiden Portale soll im Vergleich analysiert und neu hinzugekommene Leistungsengpässe herausgearbeitet werden. Im Anschluss soll das Problem dieser Leistungsengpässe gelöst und die Leistungsfähigkeit der erarbeiteten Maßnahmen untersucht werden. Dabei ist besonderer Augenmerk auf die Integration in die vorhandene Architektur der *Wikipedia* zu richten.

2 Bestehende Technologien und Systeme

Dieses Kapitel untersucht die für die Arbeit relevanten existierenden Technologien und Systeme. Dabei wird im ersten Teil eine kurze Zusammenfassung bestehender Techniken der Anfrageoptimierung aus dem Bereich der klassischen Datenbanksystem gegeben. Anschließend wird das *Semantic Web*, speziell die hier relevanten Technologien *Resource Descrion Framework (RDF)* und *SPARQL Protocol And RDF Query Language (SPARQL)*, kurz beschrieben.

Im zweiten Teil wird schrittweise die Definition eines partizipativen semantischen Portals erarbeitet. Dabei wird zunächst der klassische Portalbegriff zum Begriff des partizipativen Portals weiterentwickelt. *Wikis* werden als wichtige Klasse partizipativer Portal eingeführt und der hier relevante Anwendungsfall der Online-Enzyklopädie *Wikipedia* vorgestellt. Abgeschlossen wird das Kapitel von der Defintion eines partizipativen semantischen Portals - für dieses repräsentiert *Semantic Wikipedia* den entsprechenden Anwendungsfall.

2.1 Möglichkeiten der Anfrageoptimierung

Ziel der Anfrageoptimierung ist es, die zur Beantwortung einer Anfrage nötige Zeit und den benötigten Speicherplatz gegenüber einem Fall ohne Optimierung zu reduzieren. Gleichzeitig soll eine größtmögliche Aktualität der angefragten Daten sichergestellt werden. Diese Ziele laufen einander teilweise zuwider, so dass ein Kompromiss zwischen ihnen angestrebt wird, der auf den Anwendungsfall oder ein bestimmtes Szenario abgestimmt ist.

2.1.1 Ablauf der Anfrage-Bearbeitung in einem (R)DBMS

Um eine grobe Einordnung der Ebene (Anwendungslogik, Datenbanklogik, Dateiebene), auf der eine Optimierungsmethode wirken kann, vornehmen zu können, soll hier zunächst eine grobe Beschreibung der verschiedenen Module eines DBMS gegeben werden, die bei der Beantwortung einer Anfrage durchlaufen werden (siehe auch [Ioa96]). Dies sind:

Der Anfrage Parser (*query parser*) prüft die syntaktische Gültigkeit der gestellten Anfrage und übersetzt sie in eine interne Form. Im Falle eines RDBMSs ist dies die relationale Algebra.

Der Anfrageoptimierer (*query optimizer*) prüft die *Ablaufpläne (access plans)* aller relational-algebraisch äquivalenten Ausdrücke mittels systemabhängiger Bewertungsfunktionen. Die konkreten Kosten hängen von mehreren Faktoren, wie der Organisation der Dateistruktur, den verfügbaren und verwendeten Index-Typen sowie der Art der Bewertung (rein syntaxabhängig (z.B. Oracle Version 6) oder kostenbasiert (z.B. Oracle ab Version 7, DB2)) ab (siehe auch [HN95]). Unter diesen wählt der Anfrageoptimierer denjenigen mit dem besten Bewertungsresultat aus.

Der Anweisungsgenerator (*code generator*) transformiert den Ablaufplan in Aufrufe an die nächsten Schicht, den „Anfrageprozessor“.

Der Anfrageprozessor (*query processor*) führt die Anfrage tatsächlich aus und lädt die Informationen aus den entsprechenden Dateien.

2.1.2 Umschreiben von Anfragen (*Query rewriting*)

Bei der Bestimmung des im Hinblick auf die Berechnungskomplexität optimalen Weges, eine Anfrage zu beantworten, handelt es sich um ein NP-vollständiges Probleme (siehe u.a. [Sca01]).

Jedoch verfügt jede (relationale) Datenbank über Approximations-Strategien, um aus den vorhandenen Ablaufplänen zur Beantwortung der Anfrage den optimalen auszuwählen. Zur Erfüllung dieser Aufgabe dient ein Modul, das üblicherweise mit „Anfrageoptimierer“ (*query optimizer*) bezeichnet wird (s.o.). Dieser Vorgang wird als „Umschreiben der Anfragen“ (*query rewriting*) bezeichnet. Wie etwa in [Ioa96] beschrieben, können verschiedene Ablaufpläne massiven Einfluss auf die zur Beantwortung nötige Zeit sowie den Speicherbedarf für Zwischenergebnisse. Eine allgemeine Maxime ist es dabei, Zwischenergebnisse klein zu halten, was sowohl Zeit- als auch Speicherkomplexität reduziert.

Die Aktualität der Ergebnisse bleibt bei dieser Optimierungsvariante unberührt. Auch der dauerhaft benötigte Speicher des DBMS erhöht sich hier nicht. Zu beachten ist, dass das Umschreiben der Anfrage und die korrespondierenden Ablaufpläne stets vom verwendeten DBMS abhängig sind (siehe Erläuterungen zum Anfrageoptimierer).

2.1.3 Optimierte Index-Strukturen

Ein Datenbankindex[7] ist eine separate sortierte Zeiger-Struktur, welches es dem DBMS ermöglicht, auf die entsprechenden Datenbank-Einträge in $O(\log n)$ statt wie üblich $O(n)$ beim sequentiellen Suchen zuzugreifen. Dazu werden in der Regel B^* oder B^+-Bäume eingesetzt.

Nachteil eines Index ist, dass er den permanenten Speicherplatzbedarf deutlich erhöht. Wie der Index optimalerweise gestaltet werden sollte, ist dabei von der konkreten Anwendung abhängig. In [HD05] wird die Verwendung von Index Strukturen für RDF Anfragen im Rahmen der Entwicklung eines eigenen RDF Datenspeichers thematisiert.

Die Aktualität der Daten wird durch einen Datenbankindex nicht verändert.

2.1.4 Zwischenspeicher (*Caches*)

Ein Zwischenspeicher (*Cache*) im engeren Sinne ist eine Speichereinheit, deren Zugriffszeit deutlich (in der Regel um eine oder mehrere Größenordnungen) unter derjenigen für den zugrunde liegenden Sekundärspeicher liegt. Die Speicherkapazität ist üblicherweise ebenfalls entsprechend geringer ([PDTU00]). Dieses Konzept findet maßgeblich im Bereich der Mikrorechner-Architektur Verwendung, wo es eingesetzt wird, um die Unterschiede in den Zugriffszeiten zwischen den schnellen Prozessor-Registern und dem vergleichsweise langsamen Hauptspeicher zu kompensieren. Ebenso findet sich oft in der Steuerlogik des Festplattenspeichers ein Cache.

Die Cache-Speicherverwaltung (der *Cache Controller*) sorgt dabei dafür, dass sich im Cache möglichst jene Daten befinden, die als nächste angefordert werden (man nutzt dabei aus, dass viele Programme . Ist dies der Fall, spricht man von einem Cache-Treffer (*Cache Hit*), andernfalls von einem Cache-Fehlzugriff (*Cache Miss*). In letzterem Fall muss das gewünschte Datum aus dem

[7] siehe u.a.: `http://de.wikipedia.org/wiki/Datenbankindex`

Sekundärspeicher angefordert und im Cache abgelegt werden. Der Cache Controller hat außerdem die Aufgabe, die Kohärenz der Daten im Cache und im Sekundärspeicher sicherzustellen.

Aufgrund der beschränkten Größe des Caches ist es bei assoziativen Caches nötig, eine Verdrängungsstrategie zu definieren, welche diejenigen älteren Einträge selektiert, deren Platz durch neuere überschrieben werden kann. Gängige Strategien sind *Least Recently Used (LRU)*, bei der die Elemente ersetzt werden, die am längsten nicht mehr verwendet wurden, und*Least Frequently Used (LFU)*, bei der die Elemente ersetzt werden, die am seltensten genutzt wurden. Weitere, komplexere Strategien finden sich in [Mar01], diese sollen hier aber nicht weiter betrachtet werden.

Auch im Datenbank-Bereich findet dieses Prinzip Anwendung, so beschreibt [CR94] die Integration von Query Optimizer und Cache für bessere Leistungsfähigkeit, [CDI99] befasst sich speziell mit dem Bereich datenbankgetriebener, dynamischer Webseiten. In letzterem Bereich findet auch die nachfolgend beschriebene Technik der Ausmaterialisierung Anwendung.

Bekannte freie Datenbankprodukte wie MySQL oder PostgreSQL verlassen sich vornehmlich auf die Zwischenspeicher-Mechanismen des Betriebssystems, implementieren jedoch auch rudimentäre eigene Verfahren. Im Falle von MySQL gibt es einen Anfrage-Zwischenspeicher, der jedoch bei jeder Änderung komplett invalidiert wird[8]. Für PostgreSQL existiert das Konzept der gemeinsamen Pufferspeicher(*shared buffers*[9]), das eher dem klassischen Zwischenspeicher verwandt ist und sich zwischen dem Zwischenspeicher des Betriebssystems und dem Datenbankverwalter (*database backend*) integriert.

Neben diesen direkt in das DBMS integrierten Zwischenspeichern existieren auch anwendungsspezifische Zwischenspeicher, beispielsweise durch gesonderte Datenbanktabellen, in denen Zwischenergebnisse gespeichert werden ([Sch05]). Sie sind in der Regel weitgehend DMBS-unabhängig und stark auf die Anwendung zugeschnitten.

Die Verwendung eines Zwischenspeichers impliziert stets einen höheren Speicherverbrauch, zudem sind spezielle Logiken für die Invalidierung veralteter Einträge und (bei beschränktem Speicherplatz) die Verdrängung notwendig, die bei der Berechnung des zeitlichen Aufwandes bedacht werden müssen. Die Aktualität der Ergebnisse wird durch die Invalidierungslogik sichergestellt.

Ausmaterialisierung/ *Webcache* Ein Sonderfall eines Zwischenspeichers ist der sog. *Webcache*, ein teilweise dediziertes System, welches ausmaterialisierte, statische Versionen dynamisch generierter Inhalte eines Webservers vorhält, um diesen zu entlasten. Dies kann das Antwortverhalten des Gesamtsystems siginifikant positiv beinflussen ([IC97]). Solange sich die zugrunde liegende dynamische Seite nicht verändert, kann ohne zusätzliche Rechenzeit die statische Version ausgeliefert werden. Effektive Methoden, zu bestimmen, ob Änderungen stattgefunden haben, sind Gegenstand der Forschung ([IC98], [LR01]).

Hier gilt jedoch die Regel, dass der Zwischenspeicher von geringerer Größe ist nicht mehr zwingend. Vielmehr wird hier deutlich ein Kompromiss in Sachen Aktualität und Speicherplatz in Kauf genommen, um die Antwortzeit zu verbessern respektive die durch zahlreiche Benutzer entstehende Last überhaupt

[8] siehe `http://dev.mysql.com/doc/refman/5.1/de/query-cache.html`
[9] siehe `http://momjian.us/main/writings/pgsql/hw_performance/node3.html`

zu bewältigen.

2.2 Semantic Web

Während das *World Wide Web (WWW)* der aktuellen Generation sich am Menschen orientiert und seine Informationen nur visuell präsentiert, ist für die Zukunft absehbar, dass das *Semantic Web* ([BLHL01]), eine Weiterentwicklung anstrebt, um diese Informationen auch für Maschinen interpretiebar und formaler Logik zugänglich zu machen.

Tim Berners-Lee sah diese Möglichkeit bereits beim Entwurf des WWW vor. Die wesentlichen Entwicklungen fanden jedoch erst in den letzten Jahren statt ([BL03]). Die Gartner Group erwähnt das Semantic Web erstmal 2002, misst ihm aber noch keine große Marktrelevanz bei. Im Jahr 2004 betrachtet Berners-Lee „Phase eins“ des Semantic Web als abgeschlossen ([Rev04]) und im Jahr 2006 attestiert auch Gartner dem Einsatz des Semantic Web in Unternehmen eine Marktreife in 2 bis 5 Jahren ([Com06]).

Das *Semantic Web* setzt sich dabei aus einer Reihe von Standards zusammen, die gemeinsam das Ziel der maschinen-lesbaren Information realisieren sollen. Die beiden Wichtigsten sind im Falle dieser Arbeit *RDF* ([W3C06]), welches die Speicherung der Information spezifiziert und *SPARQL* ([P05]), die Anfragesprache, welche das Abrufen von Informationen aus RDF-Datenspeichern ermöglicht.

2.2.1 Resource Description Framework (RDF)

Das *Resource Description Framework (RDF)* ist einer der Grundpfeiler des *Semantic Web*. Es handelt sich hierbei um einen W3C-Standard, der bereits seit 1998 ([LS98]) diskutiert wird und seitdem mehrfach überarbeitet wurde.

Den aktuellen Stand dokumentiert [W3C06], die nachfolgende Beschreibung von RDF basiert überwiegend auf [KC04] und [Bec04]. Der Abschnitt über die *Turtle - Terse RDF Triple Language (Turtle)* verwendet außerdem Teile von [BL00a] und [Bec06b]. Mitunter wurde außerdem auf [EE04] zurückgegriffen.

Konzeptionell hat RDF eine Syntax, die auf einem graphenbasierten Datenmodell und eine Semantik, welche eine klar definierte Anwendung formaler Logik erlaubt.

Motivation Die Gründe für die Entwicklung von RDF waren vielfältig, auch wenn die meisten davon dem Umfeld des *World Wide Web* entstammen. Zentrale Anliegen waren:

- Strukturierte Metadaten zu ermöglichen, welche Ressourcen und die Systeme, die sie verwenden, beschreiben (z.B. Funktionalitätsbeschreibungen, Datenschutzrichtlinien etc.)
- Die Beschreibung einschränkender Datenmodelle für Applikationen, die dies verlangen, zu ermöglichen(z.B. Terminplanung, organisatorische Prozesse, *Annotation von Web-Ressourcen*)
- Die Aggregation von Wissen aus verschiedenen Anwendungen zu nutzen, um neues Wissen zu erhalten

- Die automatisierte Verarbeitung von Information im *World Wide Web* durch Software Agenten zu ermöglichen. Für diese stellt RDF eine weltweit einheitliche Kommunikationsbasis dar

Dabei soll RDF die Informationsrepräsentation so wenig wie möglich einschränken. Der Einsatz in isolierten Anwendungen zur besseren Strukturierung und Verständlichkeit gespeicherter Informationen ist möglich, jedoch profitieren vor allem solche Anwendungen von RDF, die Informationen mit anderen im Internet austauschen.

Datenmodell Das RDF-Modell ist ein formal definiertes, graphisches Datenmodell, bestehend aus benannten Knoten und (gerichteten) Kanten. Damit ist es unabhängig von einer konkreten serialisierten Syntax[10] (eine standardisierte, empfohlene Syntax wird jedoch durch RDF/XML ([Bec04]) definiert.

Alle Dinge, die durch RDF-Ausdrücke beschrieben werden können, werden als *Ressourcen* aufgefasst. Es können etwa einzelne Seiten im *WWW*, Sammlungen solcher Seiten, aber auch Bücher oder andere dingliche Gegenstände aus der realen Welt *Ressourcen* sein. Voraussetzung ist lediglich, dass sie über einen eindeutigen Universellen Ressourcen Identifizierer *URI* eindeutig festlegbar sind. Graphisch erfolgt ihre Repräsentation über Ellipsen (siehe Graphik 2).

Ressourcen werden durch ihre Eigenschaften charakterisiert (beispielsweise Ersteller, Titel oder Datum der letzten Änderungen). Die Bedeutung jeder Eigenschaft wird dabei durch die zugelassenen Werte, die Typen von Ressourcen, die durch sie beschrieben werden können und ihre Beziehung zu anderen Eingeschaften bestimmt. Ihre graphische Repräsentation ist die benannte, gerichtete Kante - ihr Wert, formal ein unicode-codiertes, möglicherweise typsiertes *Literal*, wird durch ein Rechteck symbolisiert (siehe 2).

Will man strukturierte Werte verwenden, so steht an der Stelle des Wertes eine (leere) Ressource (Ellipse), der wiederum die einzelnen Werte der Struktur zugeordnet sind[11]. Bei typisierten Literalen ist es zudem möglich, die in XML-Schema ([BM01]) definierten Datentypen zu verwenden.

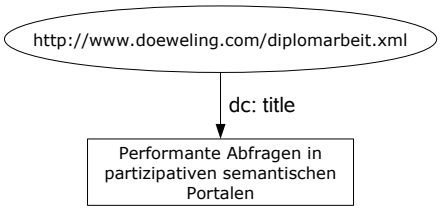

Abbildung 2: Beispiel für eine einfache RDF-Aussage

Die RDF-Spezifikation schlägt vor, dass man die Aussagen als Sätze der Form „Subjekt, Prädikat, Objekt“ interpretiert, wobei die Ressource dem Subjekt, die Eigenschaft dem Prädikt und der Wert dem Objekt entspricht. Daran orientiert ist auch die *Turtle*-Darstellung (siehe Abschnitt 2.2.1).

[10] Der hier verwendete Modellbegriff unterscheidet sich von demjenigen der mathematischen Logik. Details zu letzterem finden sich in [Hay04]

[11] für formale Betrachtung wird der entsprechenden Ressource ein eindeutiger *URI* zugeordnet

XML basierte Syntax Auch wenn das RDF-Datenmodell (s.o.) keine konkrete serialisierte Syntax vorgibt, existiert mit RDF/XML eine Empfehlung, wie RDF-Aussagen in XML-strukturierten Dokumenten gespeichert werden können. Diese ist im Kontext dieser Arbeit jedoch nicht weiter relevant. Es soll lediglich der Begriff des „qualifizierten Bezeichners“ (*QName*) definiert werden (siehe auch [BHLT06]).

Qualifizierte Bezeichner (*QNames*) Alle qualifizierten Bezeichner besitzen einen Namensraum, der durch eine URI-Referenz festgelegt wird; es ist zudem möglich, ein Präfix für qualifizierte Bezeichner zu definieren oder sie als Standard-Namensraum zu definieren. Die URI-Referenz bestimmt sich schließlich durch das Anhängen des lokalen Teils des qualifizierten Bezeichners an den Namensraum-Teil. URI-Referenzen, die in obiger Terminologie Subjekte oder Objekte identifizieren, können als XML Attributwerte gespeichert werden. RDF-Literale, die nur als Objektwerte auftreten, werden entweder zu einem XML Textelement oder zu einem XML Attributwert. Diese Konzeption erlaubt es auch, die in [BM01] definierten Datentypen zu verwenden.

Praktische werden qualifizierte Bezeichner immer dann verwendet, wenn abkürzend im Dokumentenkopf ein Präfix definiert wird, mit dem ein bestimmter Namensraum referenziert wird. Von dieser Möglichkeit wird in den meisten Beispielen dieser Arbeit Gebrauch gemacht.

Turtle - Terse RDF Triple Language Bei *Turtle* handelt es sich um eine alternative Möglichkeit, Aussagen in RDF zu machen. Im Gegensatz zu *RDF/XML* ist sie für Menschen einfacher verständlich und erlaubt es, sich einen schnellen Überblick über die Funktionsweise von RDF zu machen. Sie ist jedoch im weiteren Verlauf nur für den nachfolgenden Abschnitt 2.2.2 relevant. Daher wird hier nur ein kurzer Überblick gegeben, der zu diesem Zweck ausreicht.

Turtle orientiert sich stark Auffassung von RDF-Aussagen als Tripel (Subjekt,Prädikat,Objekt). Zur Vereinfachung der Schreibweise können außerdem Namensräume mittels „*@prefix*“ definiert werden. Beispielsweise definiert folgende Zeile „*dc*“ als Abkürzung für die *Dublin Core*-Definition (`http://purl.org/dc/elements/1.1/>`):

```
@prefix dc: <http://purl.org/dc/elements/1.1/>
```

Aussagen über das die Anweisungen enthaltende Dokument selbst können über „<>“ gemacht werden. So kann man mittels

```
@prefix dc: <http://purl.org/dc/elements/1.1/>
<> dc:title "Ein einfaches Beispiel in Turtle"
```

den Titel des entsprechenden Dokumentes als „Ein einfaches Beispiel in Turtle“ definieren. Typisierte Literale werden dabei durch *namensraum:typ*^^*literal* gekennzeichnet.

Wird bei diesen Aussagen Bezug auf Elemente aus dem Dokument genommen, so verwendet man statt des Namensraum-Präfixes einfach das Doppelkreuz (#). Eine Aussage wie

```
@prefix dc: <http://purl.org/dc/elements/1.1/>
<> <#title> "Ein einfaches Beispiel in Turtle"
```

hätte aber nur dann eine klar definierte semantische Bedeutung, wenn diese im selben Dokument festgelegt wird. Oftmals definiert man mittels

```
@prefix : <#>
```

den leeren Namensraum als den Standard-Namensraum des aktuellen Dokuments.

In *Turtle* ist es ebenfalls möglich, Ressourcen als Instanzen einer Klasse zu definieren. Dies geschieht mit obiger Schreibweise etwa durch:

```
@prefix      : <#>
@prefix   rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix  rdfs: <http://www.w3.org/2000/01/rdf-schema#>

:Person a rdfs:Class
```

Dies kann analog auch für andere Elemente von RDFS geschehen.

Formale Semantik und logische Inferenz RDF verfügt über eine klar definierte, formale Semantik , welche eine verlässliche Basis für logische Schlussfolgerungen über die Bedeutung eines RDF-Ausdrucks bietet. Diese ist durch die Ontologiesprache *RDF Schema*, kurz *RDFS* definiert ([Hay04]). Insbesondere gibt es eine eindeutig definierte Idee der logischen Schlussfolgerung, welche von nachfolgend mit Inferenz (*inference*) bezeichnet wird.

Konzeption von RDF-Schema (RDFS) Während XML Schema lediglich die syntaktische Struktur wohlgeformter XML Instanzen vorgibt, können mit RDFS Begriffe semantisch zueinander in Beziehung gesetzt werden. Dabei existieren verschiedene Arten der Beziehung zwischen Begriffen, im wesentlichen sind die Beziehung „Klasse $\leftrightarrow$ Unterklasse (*Class* $\leftrightarrow$ *subClassOf*)“ und „Eigenschaften $\leftrightarrow$ Untereigenschaften (*Property* $\leftrightarrow$ *subPropertyOf*)“.

Analog zur objektorientierten Modellierung sind Unterklassen (Untereigenschaften) spezielle Untermengen ihrer Oberklasse (Obereigenschaft). Sie beschreiben zudem im zugehörigen Instanzen-Dokument verwendete Begriffe (Eigenschaften von Begriffen).

Wichtige Klassen Der RDFS-Standard kennt die drei zentralen Klassen Ressource (*rdfs:resource*), Eigenschaften (*rdf:Property*) und Klasse (*rdfs:Class*). Dabei kennzeichnet der Namensraum *rdf*, dass sich um einen Begriff aus dem RDF-Modell handelt, der in RDFS Verwendung findet und der Namensraum *rdfs*, dass er dem RDFS-Modell entstammt. Die Zuordnung ist der Entwicklungshistorie von RDF geschuldet.

Die Menge aller Dinge, die in RDF beschrieben werden können, ist die der Ressourcen. Es handelt sich also sämtlich um Instanzen der Klasse *rdfs:Resource* betrachtet. Eigenschaften (*rdf:Property*) sind eine Teilmenge dieser Menge. Das Klassenkonstrukt (*rdfs:Class*) erlauben dabei namensgemäß die Zusammenfassung verschiedener Instanzen zu einer Klasse.

Jede Klasse oder Eigenschaft, die in RDF gebildet werden kann, ist Instanz der Klasse *rdfs:Class* respektive *rdf:Property*. Die in RDFS als Klassen definierten Ressourcen können in RDF-Dokumenten als Typ-Elemente bzw. -Attribute, die als Element definierten Ressourcen als Eigenschafts-Elemente bzw. -Attribute verwendet werden.

Wichtige Eigenschaften Von den im RDFS-Standard definierten Eigenschaften wiederum drei von zentraler Bedeutung:

- Die erste, erlaubt es den Typ (*rdf:type*), also die Klassen, deren Instanz die Ressource ist, zu beschreiben. Dies impliziert, dass alle Eigenschaften der Klasse auch Eigenschaften der Instanz sind[12]. Wird kein Typ angegeben, so wird die allgemeinste Klasse *rdf:resource* angenommen.
- Über die Eigenschaft *subClassOf* können neue Klassen in die bestehende Hierarchie integriert werden. Die Beziehung ist transitiv. Es ist weiterhin möglich, dass eine Klasse Unterklasse ihrer selbst oder einer ihrer Unterklassen ist. Auf diesem Wege kann Äquivalenz ausgedrückt werden. Jede Unterklasse muss außerdem Instanz von *rdfs:Class* sein.
- Analog zu *subClassOf* wirkt *subPropertyOf* im Bereich der Eigenschaften. Jedoch existiert hier kein oberstes Element der Hierarchie.

Ontology Web Language (OWL) Bei der *Ontology Web Language* handelt es sich um eine Erweiterung des RDFS-Standards, welche diesen eine Reihe von Ausdrücken erweitert. Zudem ist es erstmals Möglich, Ontologien zu versionieren und Kompatibilität (oder Inkompatibilitä) zwischen verschiedenen Version zu spezifizieren.

Es existieren verschiedene Varianten von OWL, namentlich *OWL Lite*, *OWL DL* und *OWL Full*. Dabei ist insbesondere *OWL DL* von Interesse, da es gerade die maximale Ausdrucksmächtigkeit bietet, bei der Vollständigkeit und Entscheidbarkeit gegeben sind.

Die RDF-Informationen der *Semantic Wikipedia* werden in OWL gespeichert, nutzen jedoch keine Ausdrücke, die über die Mächtigkeit von RDFS hinausgehen.

2.2.2 SPARQL

Dieser Abschnitt gibt einen groben Überblick, über die im April 2006 vom W3C in den Status der „*Candidate Recommendation*“ erhobene RDF-Anfragesprache SPARQL ([P05]). SPARQL steht dabei für *SPARQL Protocol and RDF Query Language*. Ebenso wie ihr Vorgänger RQDL (*RDF Query Language*) dient sie zur Suche von Informationen in RDF-Daten. Bereits zum jetzigen Zeitpunkt unterstützen die meisten verfügbaren RDF-DBMSe (z.B. [HS05], [BO04] oder [App]) SPARQL-Anfragen; SPARQL kann also mit Recht als zukünftiger Standard im Bereich der Anfragen an RDF-Datenspeicher gelten.

Grundlage für diesen Abschnitt sind überwiegend [P05] und [Doc06]. Stellenweise wurde auf [Bec06a] und [McC05] zurückgegriffen.

Konzeption Wie bereits in Abschnitt 2.2.1 erläutert, hat RDF ein graphisches Datenmodell. An dessen Aussage-Tripeln, d.h. nicht an einer konkreten serialisierten Darstellung wie RDF/XML, orientiert sich auch SPARQL. Auch wenn RDF/XML die vom W3C empfohle serialisierte Darstellung ist, soll hier analog zu [Doc06] die Turtle-Serialisierung (siehe Abschnitt 2.2.1) verwendet werden.

[12] zudem werden Kardinalitätseinschränkungen ebenfalls vererbt - diese werden jedoch hier nicht beschrieben

Zeitliche Komplexität Allgemein bedeutet eine Suche in SPARQL stets das Auffinden eines bis auf Isomorphie eindeutig bestimmten Teilgraphen (*Paarung* oder *Matching*. Für dieses graphentheoretische Problem ((Exaktes) Bestimmten eines isomorphen Teilgraphen) konnte in [GJ79] gezeigt werden, dass es im allgemeinen Fall NP-vollständig ist. Für den Spezialfall planerer Graphen (Ein *planarer* Graph ist ein Graph, der sich in der Ebene darstellen lässt, ohne dass sich seine Kanten schneiden) liegt das Problem in P, der Aufwand beträgt dabei $O(|V| \log |V|)$, wobei $|V|$ die Anzahl der Knoten in beiden Graphen ist.

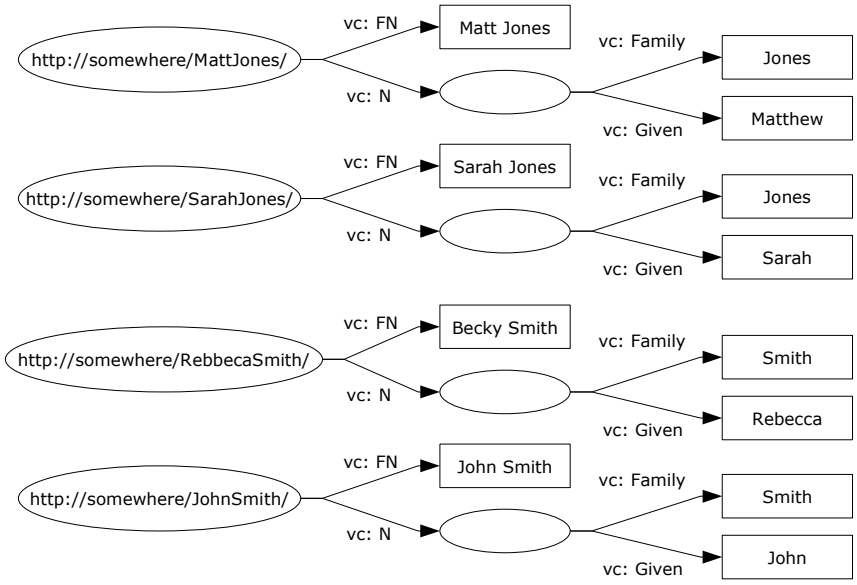

Abbildung 3: Beispieldaten für SPARQL Anfragen

Einfache Suchanfragen Um die Funktionsweise einfacher SPARQL-Anfragen zu illustrieren, wird an dieser Stelle auch auf die Beispieldaten aus [Doc06] zurückgegriffen. Diese sind in Listing 1 aufgeführt, den Ausgangsgraph stellt Abbildung 3 auf Seite 14 dar. Dabei handelt es sich um fiktive Personendaten im *vCard*-Format, welches in [IET98] definiert ist. Die Transformation in RDF/XML beschreibt [Ian01]. Nachfolgend sei stets angenommen, dass die abkürzenden Namensräume *vCard*, *rdf*, *rdf* und „“ analog zu den Beispieldaten definiert sind (die SPARQL-Syntax verwendet lediglich statt „*@prefix*“ das Schlüsselwort „*PREFIX*“) und somit ein qualifizierter Bezeichner (siehe 2.2.1) an Stelle eines vollständigen URIs verwendet werden kann.

Es können nun einige einfache Beispiele für eine Suchanfrage in SPARQL formuliert werden. Dabei werden die gesuchten Variablen durch Terme der Form „?x“ repräsentiert, die Lösungen werden in Tabellenform dargestellt. Weiterhin sei nachfolgend mit dem Begriff „Bindung (*binding*)“ ein Paar „(Variable,RDF-Term)“ bezeichnet. Für eine Lösung sind jedoch nicht zwangsläufig alle Variablen gebunden (s.u.).

Als Minimalbeispiel konstruiert man nachfolgende SPARQL-Anfrage. Die

Listing 1: RDF Beispieldaten für SPARQL-Anfragen

```
@prefix vCard : <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix rdf   : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    .
@prefix info  : <http://somewhere/peopleInfo#>
@prefix       : <#> .

<http://somewhere/MattJones/>
  vCard:FN  "Matt Jones" ;
  vCard:N   [ vCard:Family
                          "Jones" ;
              vCard:Given
                          "Matthew"
            ] .

<http://somewhere/RebeccaSmith/>
  info:age  "23"^^xsd:integer ;
  vCard:FN  "Becky Smith" ;
  vCard:N   [ vCard:Family
                          "Smith" ;
              vCard:Given
                          "Rebecca"
            ] .

<http://somewhere/JohnSmith/>
  info:age  "25"^^xsd:integer ;
  vCard:FN  "John Smith" ;
  vCard:N   [ vCard:Family
                          "Smith" ;
              vCard:Given
                          "John"
            ] .

<http://somewhere/SarahJones/>
  vCard:FN  "Sarah Jones" ;
  vCard:N   [ vCard:Family
                          "Jones" ;
              vCard:Given
                          "Sarah"
            ] .
```

Lösungen der Anfragen sollen hier in Tabellenform dargestellt werden:

```
SELECT ?x
WHERE { ?x  vCard:FN  "John Smith" }
```

x
<http://somewhere/JohnSmith/>

Offensichtlich wurde hier ein *Matching* des Tripel-Muster in der *WHERE*-Klausel im RDF-Graphen gesucht. Prädikat und Objekt sind dabei fest, das Subjekt „*?x*“ kennzeichnet die Variable x, kann also an einen Wert (oder mehrere Werte, siehe nachfolgendes Beispiel) gebunden werden. Bei dem Wert in „<>“ handelt es sich um einen URI[13], bei jenem in „“ um ein einfaches Literal.

Folgendes Beispiel illustriert eine Anfrage, welche mehrere Lösungen im Graphen hat:

```
SELECT ?x, ?fname
WHERE { ?x  vCard:FN  ?fname}
```

x	name
<http://somewhere/RebeccaSmith/>	„Becky Smith“
<http://somewhere/SarahJones/>	„Sarah Jones“
<http://somewhere/JohnSmith/>	„John Smith“
<http://somewhere/MattJones/>	„Matt Jones“

Komplexere Suchmuster Es ist außerdem möglich, über weitere *Matching*-Bedingungen zusätzliche Werte zu einem Resultat abzurufen (wie dies auch bei SQL über den *JOIN*-Operator möglich ist). Die einzelnen Bedingungen werden dabei durch einen „.“ am Ende der Zeile voneinander getrennt, das (anonyme) Subjekt der Anfrage-Bedingung muss identisch sein:

```
SELECT ?givenName
WHERE
{
  ?y vcard:Family "Smith" .
  ?y vcard:Given  ?givenName .
}
```

givenName
„John“
„Rebecca“

Leere Knoten Fügt man in der Anfrage „*?y*“ den Suchparametern hinzu, so hat man einen Fall, in dem leere Knoten auftauchen (s.o.):

y	givenName
_ : *b*0	„John“
_ : *b*1	„Rebecca“

Dabei sollen die Bezeichnungen „_ : *b*0“ respektive „_ : *b*1“ hier symbolisieren, dass es sich um zwei unterschiedliche leere Knoten handelt.

[13] genaugenommen um einen *IRI (Internationalisierter Ressourcen Identifikator)*, eine im Zeichensatz erweiterte Version eines URIs

Wert-Einschränkungen SPARQL erlaubt es, die Werte im Resultat einer Anfrage sowohl durch einfache Testausdrücke als auch durch reguläre Ausdrücke mittels des *FILTER*-Ausdrucks einzuschränken. So würde etwa der Ausdruck

```
FILTER (?age >= 24)
```

nur diejenigen Bindungen für Variablen in der Abfrage liefern, für die der zugehörige Wert „Alter“ über 23 liegt. Ebenso liefert

```
FILTER regex(?g, "r", "i")
```

nur diejenigen Resultate, auf welche der reguläre Ausdruck „*r*“ zutrifft[14]. Die Syntax regulärer Ausdrücke ist dabei die durch [MMW06] definierte.

Optionale Informationen Über die Kennzeichnung von Blöcken als *OPTIONAL* ist es in SPARQL möglich, bestimmte Informationen nur abzurufen, wenn sie verfügbar sind. So liefert etwa die Anfrage

```
SELECT ?name ?age
WHERE
{
    ?person vcard:FN  ?name .
    OPTIONAL { ?person info:age ?age }
}
```

alle definierten Personen, die einen Namen besitzen (in diesem Fall also alle) und optional die Information über ihr Alter:

name	age
„Becky Smith“	23
„Sarah Jones“	
„John Smith“	25
„Matt Jones“	

Ohne das Schlüsselwort „*OPTIONAL*“ wären nur diejenigen Ressourcen (Personen) als Resultat geliefert worden, für die auch ein Alter definiert ist. Es ist zudem möglich, in optionalen Blöcken Wertbeschränkungen zu definieren. In diesem Fall wird die optionale Information nur angezeigt, wenn der entsprechende „*FILTER*“-Ausdruck zutrifft (es fallen aber durch die Wertbeschränkung keine Resultate weg. Ist dies gewünscht, muss die Wertbeschränkung außerhalb des optionalen Blocks geschehen).

Alternativen in Suchmustern Hat man in einem Dokument mehrere, semantisch äquivalente Möglichkeiten eine Information zu beschreiben, beispielsweise sowohl durch den „formatierten Namen (*formatted name*)“ *vCard:FN* aus dem *vCard*-Vokabular als auch durch den „Namen (*name*)“ foaf:name aus dem FOAF-Vokabular ([BM06]), so kann man sich des „*UNION*“-Ausdrucks bedienen, um zu spezifizieren, dass die Information durch die Verwendung einer der beiden Alternativen niedergelegt wurde. Folgendes Beispieldokument illustriert, wie solche Definitionen aussehen könnten:

[14] Der dritte Parameter steuert das Verhalten bei der Auswertung der regulären Ausdrücke, ein „*i*“ bedeutet beispielsweise eine Auswertung unabhängig von Groß- und Kleinschreibung

```
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#> .

_:a foaf:name   "Matt Jones" .

_:b foaf:name   "Sarah Jones" .

_:c vcard:FN    "Becky Smith" .

_:d vcard:FN    "John Smith" .
```

Mittels des Alternativen-Ausdrucks „*UNION*“ können nun die beiden alternativen Definitionen in einer Anfrage kombiniert erhalten werden (der Bezeichner „[]“ markiert dabei einen leeren Knoten):

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name
WHERE
{
   { [] foaf:name ?name } UNION { [] vCard:FN ?name }
}
```

givenName
„Matt Jones“
„Sarah Jones“
„Becky Smith“
„John Smith“

2.3 Portale

Es ist zunächst zu klären, was unter einem *Portal* in der Informatik zu verstehen ist. Wikipedia ([Wik06g]) liefert unter Rückgriff auf einer entsprechenden Informationsbroschüre des Fraunhofer Institut für Arbeitswirtschaft und Organisation ([KGHV04]):

> „Ein Portal ist [...] eine Applikation, die [...] einen zentralen Zugriff auf personalisierte Inhalte sowie bedarfsgerecht auf Prozesse bereitstellt. Charakterisierend für Portale ist die Verknüpfung und der Datenaustausch zwischen heterogenen Anwendungen über eine Portalplattform. Eine manuelle Anmeldung an den in das Portal integrierten Anwendungen ist durch Single-Sign-On nicht mehr notwendig, es gibt einen zentralen Zugriff über eine homogene Benutzungsoberfläche. Portale bieten die Möglichkeit, Prozesse und Zusammenarbeit innerhalb heterogener Gruppen zu unterstützen.“

Es stehen also bei einem Portal das Bereitstellen von applikationsübergreifenden Leistungen und somit der Integrationsaspekt und nicht die technische Implementierung im Vordergrund. Ein Portal kann - muss aber nicht - auf Webtechnologie basieren. Beispiele für Portalsysteme sind im kommerziellen Bereich

etwa *Websphere*[15] von IBM oder *mySAP*[16] von SAP. Im Bereich freier Software existieren sowohl auf den Unternehmensbereich zugeschnittene Lösungen (wie etwa *Apache Jetspeed*[17]) als auch solche, die sich eher für Privatpersonen und Kleinunternehmen eigenen (wie etwa *Joomla*[18]).

2.4 Partizipatives Portal

Unter einem partizipativen Portal versteht man ein Portal, welches seine Inhalte ganz oder teilweise durch seine Nutzer erhält. In der Regel kann jeder Benutzer, mitunter erst nach einer Registrierung, Inhalte auf das Portal spielen und diesen Informationen hinzufügen. Diese Informationen sind wiederum allen Benutzern zugänglich. Je nach Portal können auch bereits vorhandene Inhalte verändert werden.

2.4.1 Wikis

Einen wichtige Klasse der partizipativen Portale stellen die sog. *Wikis* dar. Der Begriff geht auf eine Entwicklung von Ward Cunningham zurück ([Wik06k],[LC01]). Seitdem wird der Begriff *Wiki* in der Informatik prototypisch für eine Software verwendet, welche es ihren Benutzern (insbesondere anonymen) ermöglicht, Seiten anzulegen, zu editieren oder zu löschen[19].

Das System stellt also eine Plattform dar, die Nutzer „auffordert“, ihr Wissen einzubringen und gemeinsam an der Schaffung einer Wissensbasis zu partizipieren. Dies impliziert insbesondere, dass als Qualitätskontrolle nicht mehr vornehmlich auf redaktionelle Kräfte vertraut wird, sondern vielmehr auf eine „soziale“ Kontrolle durch die Nutzerbasis.

Dieses Prinzip wurde unter anderem von der freien Enzyklopädie Wikipedia[20] aufgegriffen, die nachfolgend beschrieben wird.

2.4.2 Wikipedia

Nach ihrem Selbstverständnis ([Wik06i]) ist *Wikipedia* eine

> [...] von freiwilligen Autoren verfasste, mehrsprachige, freie Online-Enzyklopädie [...] [mit dem Ziel] [...] die Gesamtheit des Wissens unserer Zeit in lexikalischer Form anzubieten.[...]

Der Begriff „Wikipedia“ selbst ist dabei eine Wortkreuzung

> [...] aus „Encyclopedia“ (englisch für Enzyklopädie) und „Wiki“ [...]

Wie bereits eingangs in Abschnitt 2.4.1 dargelegt, handelt es sich bei einem *Wiki* um eine Software, welche es ihren Benutzern (insbesondere anonymen) ermöglicht, Artikel anzulegen, zu editieren oder zu löschen. Die Prinzipien der Wikipedia sehen derzeit (für die deutsche Wikipedia) keine Einschränkungen dieser Möglichkeiten vor. Es gibt jedoch sowohl die Möglichkeit, dass Administratoren

[15] Websphere, `http://www-306.ibm.com/software/de/websphere/`
[16] mySAP, `http://www.sap.com/germany/solutions/business-suite/index.epx`
[17] Jetspeed, `http://portals.apache.org/jetspeed-1/`
[18] Joomla, `http://www.joomla.org`
[19] Die Möglichkeiten können von Administratoren eingeschränkt werden
[20] Wikipedia, `http://www.wikipedia.org`

durch Vandalismus gefährdete Seiten sperren als auch Artikel gänzlich entfernen. Weiterhin wird derzeit die Einführung „stabiler Artikelversionen“ diskutiert (siehe `http://de.wikipedia.org/wiki/Wikipedia:Stabile_Version`). Die englische Version sieht außerdem vor, dass nur noch registrierte Benutzer neue Artikel anlegen können.

Zum Editieren der Seiten wird die Mediawiki eigene Syntax verwendet[21]; diese orientiert sich entfernt an der Syntax von HTML 4[22]. Inhaltlich hat

> Bestand [...], was von der Gemeinschaft akzeptiert wird. [...] [Die] GNU-Lizenz für freie Dokumentation [erlaubt dabei] [...] die Inhalte unentgeltlich - auch kommerziell - zu nutzen, zu verändern und zu verbreiten.

Wikipedia selbst hat sich jedoch verpflichtet, niemals Werbung in sein Angebot zu integrieren.

Geschichte Die nachfolgende Darstellung der geschichtlichen Entwicklung von Wikipedia ist wiederum überwiegend [Wik06i] entnommen. Informationen zu Nupedia stammen aus [Fro06]:

Der ersten Wurzeln der *Wikipedia* datieren auf März 2000, als der Internet-Unternehmer Jimmy Wales seinen Anlauf zu einer Internet-Enzyklopädie startete. Er engagierte den Philosophiedozenten Larry Sanger und rief mit ihm als Chefredakteur das Projekt *Nupedia* ins Leben.

Der 7 stufige Redaktionsprozess und die Tatsache, dass bestimmte Rollen im Begutachtungsprozess sogar nur für promovierte Nutzer vergeben wurden, setzte jedoch die Hemmschwelle für Autoren so hoch, dass das Projekt nur sehr langsam wuchs. Daher regte Sanger die Integration eines *Wikis* innerhalb des Nupedia-Projekts an. Dies gilt als die Geburtsstunde der Wikipedia.

Diese sollte ursprünglich als Plattform zur gemeinsamen Erstellung von Artikeln dienen, die später den Redaktionsprozess der Nupedia durchlaufen. Vor allem aufgrund seiner Offenheit trat die ursprüngliche Idee schnell in den Hintergrund.

Im Laufe der weiteren Entwicklung markieren die Einrichtung der ersten nicht-englischsprachigen Versionen im März 2001, die Entlassung von Larry Sanger als Chefredakteuer im Februar 2002 sowie vor allem die Gründung der Non-Profit-Organisation „Wikimedia Foundation“ und gleichzeitige Übereignung der Server, auf denen Wikipedia betrieben wurde an diese.

Relevanz Auch wenn die Qualität der Artikel in der *Wikpedia* mitunter umstritten ist[23], kann der Erfolg der *Wikipedia* nur schwerlich geleugnet werden. Im Februar 2006 zählte sie 68.000 Nutzern und über eine Million Artikel (englische Sektion)[Wik06d] und ist damit eines der größten, frei im Internet zugänglichen, partizipativen Portale.

Bereits im Oktober 2004 gab es täglich etwa 6 Millionen Zugriffe und etwa 20.000 Bearbeitungen. Mittlerweile hat sich diese Zahl noch deutlich erhöht auf

[21] Mediawiki Syntax, `http://www.mediawiki.org/wiki/Help:Formatting`

[22] HTML 4.01 Standard, `http://www.w3.org/TR/html4/`

[23] So wurde etwa John Seigenthaler kurzzeitig fälschlicherweise laut dem Wikipedia-Artikel (`http://en.wikipedia.org/wiki/John_Seigenthaler_Sr.`) eine direkte Involvierung in das Kennedy-Attentat nachgesagt (siehe [Tod05])

etwa 80 Millionen Zugriffe täglich [Web06]. Detailliertere Analysen scheitern derzeit leider an der Nicht-Verfügbarkeit aktueller Daten.

2.5 Partizipative semantische Portale

Klassische Portale exportieren ihre Information in der Regel nur in für Menschen interpretierbarer Form oder in herstellerspezifischen Schnittstellen. Der Ansatz, die Ideen des *Semantic Web* ([BLHL01]) in Portalen zu nutzen, findet sich bereits 2001 in [MSS+01]. Eine Überblick liefert auch [RSC04].

Unter Rückgriff auf letztere Arbeit soll unter einem *semantischen Portal* nachfolgend ein solches Portal verstanden werden, das seine Informationen (oder Teile davon) in einem mit dem RDF-Standard konformen Format exportiert.

Ein *partizipatives semantisches Portal* sei durch das Vereinen der Eigenschaften partizipativer und semantischer Portale definiert. Ein solches ist die für diese Arbeit relevante *Semantic Wikipedia*.

2.5.1 Semantic MediaWiki

Bei *Semantic MediaWiki* handelt es sich um eine Erweiterung der klassischen *MediaWiki*-Software, auf der u.a. *Wikipedia* basiert (siehe Abschnitt 2.4.2). *Semantic MediaWiki* wird über den Mechanismus der Parser-Erweiterung in *Mediawiki* integriert und hat daher Zugriff auf dieselben Informationen und Funktionen wie *MediaWiki*.

Die Funktionalität von *Semantic MediaWiki* wird im nachfolgenden Abschnitt beschrieben.

2.5.2 Semantic Wikipedia

Bei *Semantic Wikipedia* ([VKV+06a]) handelt es sich um ein Projekt, das die partizpative Natur eines viel genutzten partizipativen Portals wie *Wikipedia* mit der maschinen-interpretierbaren Informations-Klassifizierung des *Semantic Web* verbindet.

Dazu wird mit *Semantic MediaWiki* eine Erweiterung entwickelt, welche die Syntax von *MediaWiki* nur um wenige Ausdrücke erweitert, um eine leichte Erlernbarkeit durch die Benutzer zu gewährleisten. Es existiert bereits eine Reihe von Systemen, die *Semantic MediaWiki* einsetzen, unter anderem das Informationsportal Ontoworld[24], welches Informationen zu Entwicklungen und Ereignissen aus der Welt des *Semantic Web* bietet.

Funktionalität Die erweiterte Funktionalität des *Semantic MediaWiki* gründet im Wesentlichen auf zwei Ergänzungen der ursprünglichen *MediaWiki*-Syntax:

- *Typisierten Links* und
- *Attributwerten*

Diese Möglichkeiten sollen nachfolgend unter dem Begriff *(semantische) Annotationen* zusammengefasst werden.

[24] Ontoworld.org, `http://www.ontoworld.org`

Auf dieser Basis wurden ein zusammenfassender Informationskasten (*Factbox*), eine einfache semantische Suche[25], der Export von Seiten im RDF-Format[26] sowie eine Konversions-Routine für einige gängige physikalische Einheiten implementiert. Eine wesentlich komplexere Funktionalität existiert mit der Möglichkeit, innerhalb von *Wiki*-Seiten Anfragen an die semantische Wissensbasis zu stellen.

Technisch betrachtet ist *Semantic MediaWiki* eine Erweiterung zu *MediaWiki*, die in PHP implementiert ist und die semantische Wissensbasis in einer MySQL-Tabelle speichert. Mittelfristig ist jedoch geplant, die Speicherung ein geeignetes RDF-Datenbank-Managementsystem zu verlagern ([VKV+06a]). Weitere Details zur Architektur finden sich im Abschnitt 3.4.2.

Nachfolgend sollen die wichtigsten Funktionalitäten des *Semantic MediaWiki* jeweils kurz beschrieben werden (der Text ist im Wesentlichen eine Übersetzung und Zusammenfassung von [Ont06]. Einzelne Elemente sind [VKV+06a] entnommen. Der Abschnitt „Integrierte Anfragen (*Inline Queries*)“ stützt sich zudem auf [VKV+06b]). Keine Beachtung finden dabei die einfache semantische Suche, semantische Vorlagen (*Semantic templates*) sowie eine Reihe von Spezial-Seiten, welche dem Benutzer die Verwendung des *Semantic MediaWiki* erleichtern, jedoch für das Verständnis und für diese Arbeit nicht weiter relevant sind.

Annotationen Annotationen sind spezielle Textauszeichnungs-Elemente, die es Benutzern, die eine *Wiki*-Seite bearbeiten, erlauben, Teile des Seiteninhalts explizit zu klassifizieren, so dass die korrespondierende Information der semantischen Wissenbasis hinzugefügt werden kann. Insbesondere bieten semantische Annotationen daher die Möglichkeit, die entsprechenden Informationen über die Suchmöglichkeiten des *Semantic MediaWiki* gezielt abzurufen.

Die Annotationen in *Semantic MediaWiki* können als logische Fortsetzung des existierenden Kategoriensystems von *MediaWiki* betrachtet werden. Kategorien sind ein Mittel, Artikel nach gewissen Kriterien zu klassifizieren - so kann etwa durch Hinzufügen von [[Category:Stadt]] die entsprechende Seite als Beschreibung einer Stadt markiert werden - und somit die Menge der Informationen zu ordnen. Mit Hilfe dieser Information kann etwa eine geordnete Liste aller Städte in einem *Wiki* erzeugt werden.

Semantic MediaWiki nutzt diese Kategorie-Informationen, bietet jedoch darüber hinaus weitere Möglichkeiten, ein *Wiki* zu strukturieren:

- Relationen können als „Kategorien für Verweise“ interpretiert werden. Sie erlauben, die Bedeutung eines Verweises zwischen zwei Artikeln zu benennen. So kann etwa der Verweis vom Artikel Berlin auf den Artikel Deutschland durch die Beziehung, die Hauptstadt eines Landes zu sein, beschrieben werden.

- Attribute erlauben dem Benutzer, Datenwerte für charakteristische Eigenschaften zu spezifizieren. Beispielsweise kann auf die Weise Berlin mit einer *Bevölkerungzahl* von $3,396,990$ assoziiert werden.

Dabei kann für jede Relationen und jedes Attribut jeweils ein Artikel angelegt werden, mit dem ihre bzw. seine Verwendung textuell beschrieben wird.

[25] Einfache semantische Suche, *Special:SearchTriple*
[26] RDF Export einer Seite, *Special:ExportRDF*

Diese beschreibenden Artikel werden durch spezielle Namensräume von den restlichen Artikeln abgegrenzt („*Relation:*“ für Relationen, „*Attribute:*“ für Attribute).

Datentypen bei Attributen Damit Attributwerte korrekt interpretiert werden können, muss außerdem ein Datentyp spezifiziert werden. Im Falle obigen Beispiels ist dies der Datentyp Ganzzahl. Neben dem Datentyp Ganzzahl (*Type:integer*) definiert Semantic MediaWiki eine Reihe weiterer Datentypen[27]. Der Präfix „*Type:*“ ist dabei wiederum ein eigener Namensraum, der durch *Semantic MediaWiki* definiert wird und es erlaubt, beschreibende Artikel für Typen von anderen Artikeln abzugrenzen.

Wenn also ausgedrückt werden soll, dass das Attribut „Bevölkerungszahl“ eine Ganzzahl ist, so ist dem Artikel zum Attribut Bevölkerungszahl[28] die Information „*[[has type::Type:integer]]*“ hinzuzufügen. Dies impliziert außerdem, dass die spezielle Relation „*has type*“ für die Beziehung zwischen dem Attribut Bevölkerungszahl und dem Typ Ganzzahl gilt. Die Relation „*has type*“ hat anders als gewöhnliche, benutzerdefinierte Relationen eine vordefinierte Bedeutung.

Informationskasten (*Factbox*) Der *Informationskasten* verwertet semantische Annotationen auf elementarer Ebene. Dies bedeutet üblicherweise die Darstellung aller Annotationen, die in einem Artikel vorhanden sind, an seinem Ende (die Darstellung ist abhängig von der Konfiguration).

Dies dient hauptsächlich dazu, dem Benutzer eine Rückmeldung darüber zu geben, welche seiner Eingabe durch *Semantic MediaWiki* korrekt ausgewertet werden konnten. Eine mögliche Fehlerquelle ist beispielsweise die Zuweisung von Werten inkorrekten Datentyps zu einem Attribut, so etwa wenn beliebiger Text einem Attribut zugewiesen wird, das Ganzzahlen erwartet.

Der *Informationskasten* bietet zudem Verweise auf die benutzten Attribute und Relationen, so dass der Benutzer schnell weitere Informationen über deren Benutzung erhalten kann. Relationen und Attribute werden dabei in zwei getrennten Abschnitten angezeigt, um auf die verschiedenartige Benutzung hinzuweisen. Bei Annotationen, die bereits ins System integrierte Bedeutungen haben, zeigt der *Informationskasten* diese im zusätzlichen Abschnitt „Besondere Eigenschaften (*Special properties*)“ an (z.B. für die Relation „*has type*“).

Zusätzlich bietet der *Informationskasten* die Möglichkeit einer einfachen Suche. Annotationen liefern[29] Verweise auf ähnliche Artikel[30]. In einigen Fällen, wie etwa bei Attributen vom Typ „geographische Koordinate (*Type:Geographic coordinate*)“ werden auch Verweise auf externe Dienste[31] angezeigt. Für Attribute die Maßeinheiten unterstützen, werden umgerechnete Werte in anderen allgemein üblichen Maßeinheiten angezeigt.

Schließlich liefert der *Informationskasten* auch noch einen Verweis, der ihre Inhalte im maschinenlesbaren RDF-Format ausgibt und einen weiteren, der auf

[27] Integrierte Datentypen, siehe `http://wiki.ontoworld.org/index.php/Help:Annotation#Special_types`

[28] *Attribute:Bev%C3%B6lkerungszahl*

[29] bis auf Ausnahmen wie den bereits diskutierten speziellen Relationen

[30] in dem Sinne, dass diese dieselbe Relation beinhalten

[31] hier sind dies Online-Kartendienste, die Satellitenbilder oder Karten der entsprechenden Gegend anzeigen

eine Hilfe zum Editieren im Semantic MediaWiki anbietet. Einen Gesamteindruck bietet Abbildung 4.

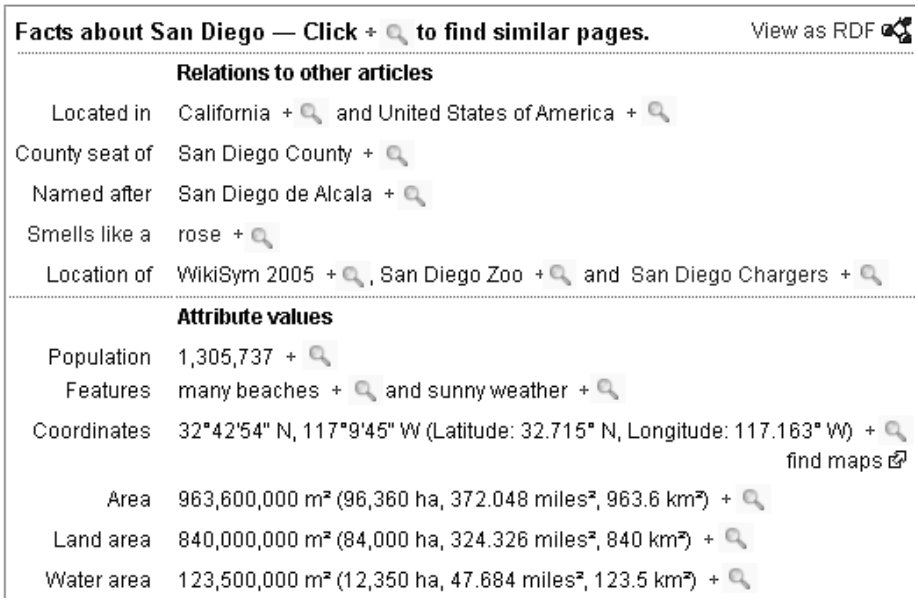

Abbildung 4: Semantic Mediawiki Informationskasten

Integrierte Anfragen (*Inline Queries*) / SMW-QL Die für diese Arbeit maßgebliche Funktionalität des *Semantic MediaWiki* ist die Integration einer Anfragesprache, die es ermöglicht, auf das durch Annotationen formalisierte Wissen innerhalb des Portals zuzugreifen. Diese soll nachfolgend mit *SMW-QL* abgekürzt werden.

Ihre Syntax ist an jene der Annotationen angelehnt, um eine hohe Zugänglichkeit für Benutzer, die bereits mit dem Bearbeiten von semantisch annotierten *Wiki*-Seiten vertraut sind, zu gewährleisten (genauere Untersuchungen der Syntax, insbesondere im Vergleich zu SPARQL finden sich in Abschnitt 4.1).

Unabhängig davon erschließt sich die Nützlichkeit dieser Funktionalität auch für Benutzer, die keine Veränderungen vornehmen wollen - die integrierten Anfragen liefern dynamisch aktualisierte Resultate auf der sie einschließenden Seite. Auf diese Weise können wenige Editoren viele Leser mit aktuellen Informationen versorgen. So ist es beispielsweise möglich, im Satz

> Der oberste Befehlshaber ist der Präsident der Vereinigten Staaten.
> Dies ist derzeit [...].

durch eine integrierte Anfrage stets den aktuell korrekten Wert anzuzeigen. Im Sinne besserer Wartbarkeit muss die Änderungen also stets nur noch an einer Stelle erfolgen.

Die Implikationen dieser Möglichkeit auf die Architektur und das Leistungsverhalten des *Semantic MediaWiki* werden in Abschnitt 3.4 diskutiert. Die entstehende Leistungsproblematik bildet den Kernpunkt dieser Arbeit.

3 Analyse des Problems

Dieses Kapitel untersucht im ersten Teil das Szenario partizipativer Portale, teilweise unter Bezugnahme auf statistische Daten über die freie Online-Enzyklopädie Wikipedia. Der zweite Abschnitt stellt eine Analyse der Architektur der Wikipedia dar, welche insbesondere untersucht, mit welchen Maßnahmen den auftretenden Problemen begegnet wurde. Der dritte Abschnitt betrachtet das erweiterte Szenario des Projektes *Semantic Wikipedia* und zeigt potentielle Flaschenhälse der veränderten Architektur auf.

Der letzte Abschnitt widmet sich schließlich der Beschreibung möglicher Optimierungen der Beantwortung von Anfragen über Datenspeichern, um diesen Flaschenhälsen zu begegnen. Die Ideen sind dabei überwiegend dem Bereich der klassischen, relationalen Datenbank-Managementsysteme *((R)DBMSe)* entlehnt. Die Beurteilung der Anwendbarkeit im konkreten Szenario findet im nächsten Kapitel statt.

3.1 Partizipative Portale

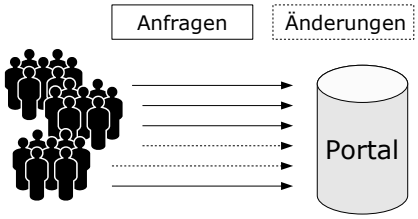

Abbildung 5: Anfragen und Aktualisierungen eines partizipativen Portals

Als Szenario sei ein partizipatives Portal angenommen, semantische Erweiterungen sollen zunächst bei der Betrachtung ausgeblendet werden. Es sei weiter angenommen, dass innerhalb eines Zeitintervalls t eine Menge von Benutzern auf das Portal zugreifen und dabei durch eine Reihe von Anfragen an den zugrunde liegenden Datenspeicher durch Abruf von Seiten des Portals auslösen. Die Menge der Anfragen werde mit Q (*Queries*) bezeichnet, ihre Kardinalität werde mit $q = |Q|$ bezeichnet. Je nach Häufigkeit der Seitenabrufe, werden die Anfragen auch mehrfach ausgeführt.

Es wird weiterhin davon ausgegangen, dass eine Teilmenge der o.g. Benutzer, auch Seiten bearbeitet und dadurch Informationen im zugrundenliegenden Datenspeicher ändert (siehe Abbildung 5). Die Menge der Änderungen sei nachfolgend mit C (*Changes*) bezeichnet, ihre Kardinalität mit $c = |C|$. Außerdem gelte $C \cap Q = \emptyset$, d.h. Anfragen und Änderungen bezeichnen vollständig separate Vorgänge.

Vereinfachend werden für die nachfolgenden Betrachtungen folgende Annahmen gemacht:

- Jede Seite referenziert unterschiedliche Informationen, insbesondere führen unterschiedliche Seiten zu unterschiedlichen Anfragen.

- Pro Änderungsvorgang wird immer nur eine Seite aktualisiert. Ändert ein Benutzer mehrere Seiten, so löst er damit auch mehrere Änderungen im Datenspeicher aus.

3.1.1 Verhältnis Anfragen-Änderungen

In diesem Szenario können nun im wesentlichen drei Fälle für das Verhältnis von Anfragen zu Änderungen unterschieden werden:

1. $q \gg c$
2. $q \ll c$
3. $q \simeq c$

Unter Rückgriff auf die Zugriffsstatistiken der Wikipedia ([Wik06d]) stellt man fest, dass trotz einer relativ hohen Zahl von Änderungen die Zahl der Zugriffe diese deutlich übersteigt (im Oktober 2004 um den Faktor 4000 (6 Millionen Zugriffe bei etwa 20.000 Bearbeitungen)).

Die Tendenz der Statistik lässt außerdem vermuten, dass sich dieses Verhältnis mit steigender Nutzerzahl noch zugunsten der Anfragen verbessert. Es liegt also der erste Fall vor. Es ist zudem davon auszugehen, dass sich in anderen partizipativen Portalen mit ausreichend hoher Nutzerzahl aufgrund ähnlicher Strukturen auch ähnliche Verhältnisse ergeben.

3.1.2 Formale Beschreibung des Szenarios

In [LR01] wird festgestellt, dass auf stark frequentierten, dynamisch generierten Seiten - also auch auf den hier diskutierten partizipativen Portalen - oftmals eine relativ kleine Teilmenge von Seitenabrufen für den größten Teil der Last im System verantwortlich ist. Dies motiviert eine formale Beschreibung des Szenarios, auf welche später bei Leistungsbetrachtungen (siehe Abschnitt 7.2) zurückgegriffen werden kann.

Abzählen von Seiten, Anfragen und Änderungen Es bezeichne zunächst P die Seiten (*Pages*) des Portals, $p = |P|$ ihre Anzahl. Für das Intervall t sei p als konstant angenommen. Aufgrund der Voraussetzung, dass jeder Seitenabruf zu einer anderen Anfrage führt, gilt $p = q$. Dann ist es möglich, eine injektive Funktion $i_P : P \longmapsto \mathbb{N}_q$ zu definieren, welche die Seiten von P mit natürlichen Zahlen abzählt:

Nimmt man an, dass I die systemabhängige Indexmenge bezeichnet, welche es erlaubt, jeder Seite eindeutig ein Element $i \in I$ zuzuordnen (etwa indem man Seiten über Ihre Titeln identifiziert (und alle Titel paarweise verschieden sind) oder die Schlüssel des zugrunde liegenden DBMS zu diesem Zweck verwendet), so kann diese Menge linear durchlaufen und jedem Element eine natürliche Zahl zugeordnet werden.

Da nach Voraussetzung die Menge P genau q Elemente hat, kann sie auch als $P = \{p_1, \ldots, p_q\}$ geschrieben werden.

Mit Hilfe der Funktion $r : P \longmapsto Q$, welcher jeder Seite p_i die Anfrage $x \in Q$ zuordnet, welche durch ihren Abruf (*request*) entsteht, können auch die Anfragen abgezählt werden. Man verwendet die Funktion $i_Q : Q \longmapsto \mathbb{N}_q$ und

setzt $i_Q(x) = i$, wenn $x = r(p_i)$. Somit kann auch Q durch $Q = \{q_1, \ldots, q_q\}$ beschrieben werden.

Definiert man außerdem die Funktion $e : P \longmapsto C$, welcher jeder Seite p_i die Änderung $y \in C$ zuordnet, die entsteht, wenn die systemabhängigen Bearbeitungsroutinen (*edit*) verwendet werden, so erhält man analog über die Funktion $i_C : C \longmapsto \mathbb{N}_q$ mit $i_C(y) = i$, wenn $y = e(p_i)$ die Darstellung $C = \{c_1, \ldots, c_q\}$ für C

Häufigkeit von Anfragen und Änderungen Es sei nun die Funktion $\#_P : P \longmapsto \mathbb{N}^0$, welche zu jeder Seite die Anzahl ihrer Abrufe angibt. Es gilt also $\#_P(p_i) = n_i$ mit $n_i \in \mathbb{N}^0$. Nach Voraussetzung kann dann eine entsprechende Funktion $\#_Q : Q \longmapsto \mathbb{N}^0$ durch $\#_Q(q_i) = \#_Q(r(p_i)) = \#_P(p_i) = n_i$ definiert werden. Analog bezeichne $\#_C : C \longmapsto \mathbb{N}^0$ mit $\#_C(c_i) = \#_C(e(p_i)) = m_i$ die Anzahl der Änderungen an einer Seite.

Nachfolgend soll abkürzend die Funktion zum Zählen der Häufigkeit nur noch als $\#$ bezeichnet werden, solange aus dem Kontext erschlossen werden kann, ob es sich um Seiten, Anfragen oder Änderungen handelt.

Fazit Zusammenfassend lässt sich, unter Einbeziehung der Beobachtung „$q \gg c$“ aus Abschnitt 3.1.1 sagen:

1. Die Mengen der Seiten, Anfragen und Änderungen lassen sich mit den natürlichen Zahlen bis q abzählen, es gilt:
   - $P = \{p_1, \ldots, p_q\}$
   - $Q = \{q_1, \ldots, q_q\}$
   - $C = \{c_1, \ldots, c_q\}$
2. Die Anzahl der Seitenabrufe und Anfragen ist gleich:
$$\forall i \in \mathbb{N}_q : \quad \#p_i = \#q_i$$
3. Die Gesamtanzahl der Anfragen ist deutlich größer als die der Änderungen
$$\sum_{i=1}^{q} \#q_i \gg \sum_{i=1}^{q} \#c_i$$

3.2 Szenario: Wikipedia

Im Februar 2006 zählte Wikipedia 68.000 Nutzern und über einer Million Artikel in der englischen Sektion ([Wik06d]) und ist damit die größte frei im Internet verfügbare Enzyklopädie. Bereits im Oktober 2004 gab es täglich etwa 6 Millionen Zugriffe und etwa 20.000 Bearbeitungen. Mittlerweile hat sich diese Zahl noch deutlich erhöht auf etwa 80 Millionen Zugriffe täglich [Web06]. Detailliertere Analysen scheitern derzeit leider an der Nicht-Verfügbarkeit aktueller Daten.

Die stetig steigende Zahl von Zugriffen und Änderungen an Seiteninhalten führten dabei zur evolutionären Entwicklung einer komplexen, mehrschichtigen Architektur, welche auch aktuell noch im Wachsen begriffen ist. Vorgreifend auf die Analyse des eigentlichen Anwendungsfalls *Semantic Wikipedia* in Abschnitt

3.4 wird nachfolgend ein Überblick über den aktuellen Stand dieser Architektur gegeben und die einzelnen ergriffenen Maßnahmen zur Verbesserung des System-Leistungsverhaltens werden erläutert.

3.2.1 Überblick

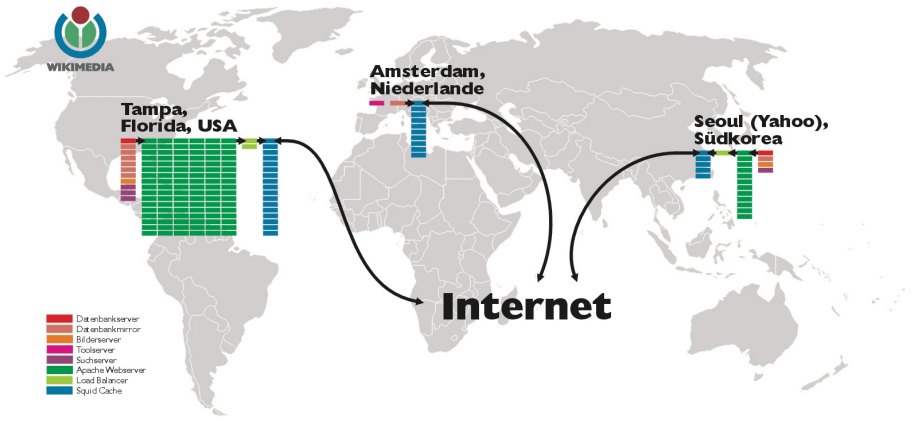

Abbildung 6: Wikipedia Server (Stand: März 2006), Quelle: Wikimedia

Abbildung 6 bietet einen Überblick über die Serverzahl der Wikipedia. Zwischen diesen Systemen wird die Last dynamisch verteilt[32], für die Datenbank-Server werden Replikationstechniken angewandt ([Wik06f]).

Abbildung 7 schlüsselt die Verwendung der Server näher auf. Dabei wird bereits deutlich, dass neben der genannten Lastverteilung auch Zwischenspeicher (*Caches*) Verwendung finden.

3.2.2 Konzeption der Zwischenspeicher-Hierarchie

Die zur Leistungssteigerung verwendeten Zwischenspeicher sind dabei teilweise in die *MediaWiki*-Software, auf der Wikipedia basiert, integriert, teilweise werden sie durch zusätzliche Software-Lösungen implementiert([Wik05], [Krs06]). Es existieren insgesamt vier verschiedene Zwischenspeicher, welche nachfolgend im Detail beschrieben werden sollen.

Zwischenspeicher „*squid*“ (*webcache*) Bei *Squid*[33] handelt es sich um einen sogenannten *Webcache* (siehe Abschnitt 2.1.4). Er hält unveränderte Seiten für eine konfigurierbare Zeit[34] vor und fängt auf diese Weise im Mittel 78% der Last auf Wikipedia ab (vor allem Zugriffe von nicht registrierten oder nicht angemeldeten Benutzern). Die Wikipedia hat zudem eine verteilte Architektur für die eingesetzen *Squid*-Systeme konzipiert. Die Invalidierung der Inhalte des Zwischenspeichers geschieht über das speziell zu diesem Zweck entwickelte *Multicast HTCP purging* Protokoll[Wik06b].

[32] für die Webserver im Round-Robin Verfahren
[33] Squid, `http://www.squid-cache.org/`
[34] Die Zeit in Sekunden wird durch den *MediaWiki*-Parameter $wgSquidMaxage gesteuert

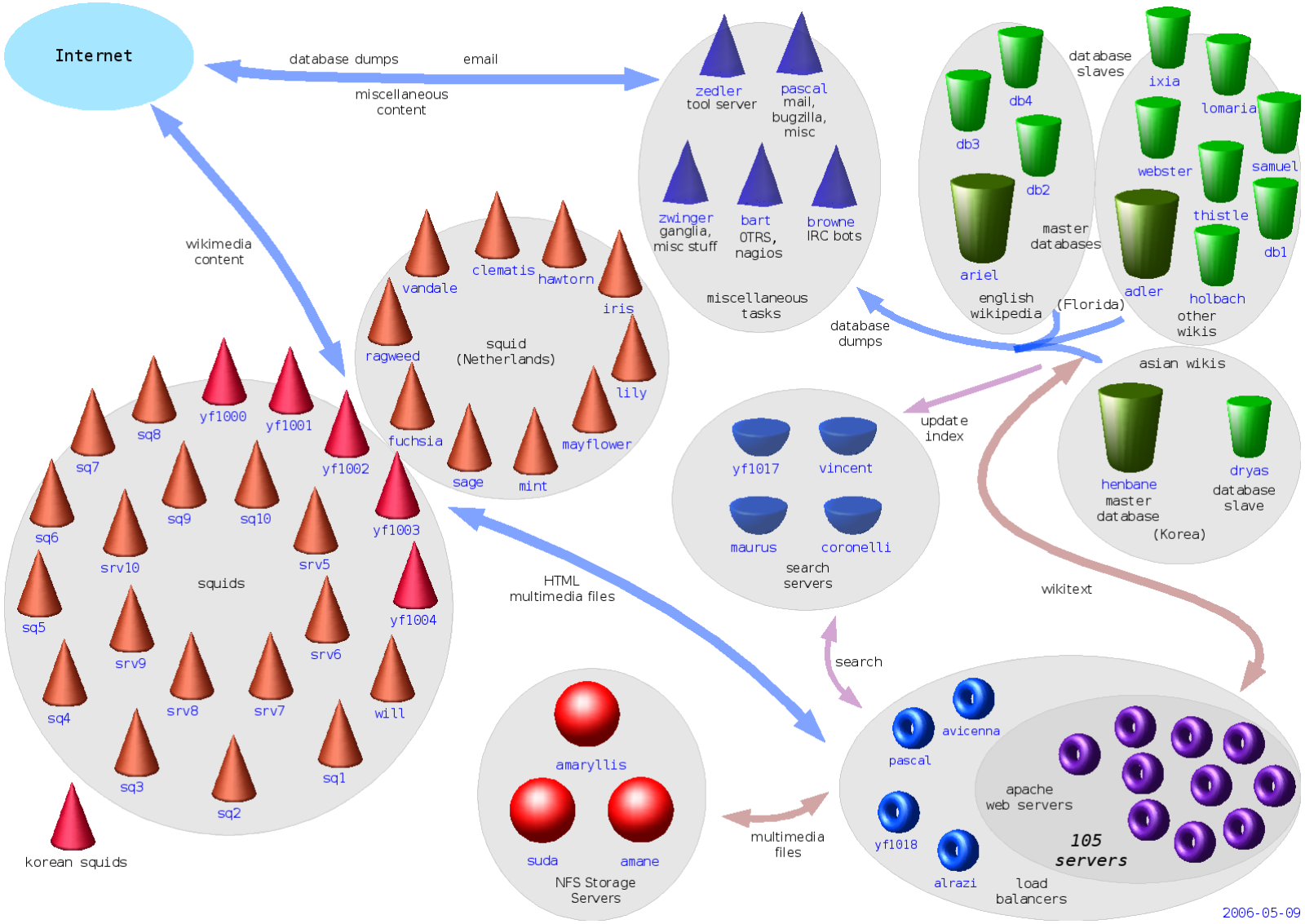

Abbildung 7: Aufgabenverteilung der Wikipediaserver, Quelle: Wikimedia

Frühere Versionen von der von Wikipedia eingesetzten Mediawiki Software setzten hier auf Punkt-zu-Punkt TCP Verbindungen (*Unicast*) zwischen allen Apache Webservern und allen Squid Systemen. Diese Lösung skalierte jedoch mit steigender Benutzerzahl sehr schlecht. Laut Angabe von [Wik06b] ist die neue Methode ist im Vergleich zu den Punkt-zu-Punkt Verbindungen etwa um den Faktor 8000 schneller.

Prüfung des Parsers bei Änderungen Bei jedem durch Benutzer eingeleiteten Änderungsvorgang prüft der Parser zunächst, ob tatsächlich Änderungen an der Seite vorgenommen wurden, indem er den aktuellen Zustand mit dem letzten vergleicht, der beim Speichern der betroffenen Seite abgelegt wurde. Wurden keine Änderungen vorgenommen (oder alle Änderungen vor dem Speichern wieder vollständig rückgängig gemacht), so wird der Parser die Auszeichnungselemente (*markup*) der Seite nicht noch einmal analysieren, sondern auf das Resultat des letzten Durchlaufs für diese Seite zurückgreifen.

Dies ist insbesondere für Erweiterungen wie Semantic Mediawiki hinderlich, weil Änderungen, die sich nicht auf die *MediaWiki*-internen Datenbank-Tabellen auswirken, somit bis zur Invalidierung des Zwischenspeichers nicht sichtbar sind. Es ist jedoch möglich, diesen Zwischenspeicher komplett zu deaktivieren ([Wik06a]) (in der Konfigurationdatei von *MediaWiki*) oder eine Löschung des Zwischenspeichers für eine einzelne Seite zu erzwingen ([Wik06j])[35]. Letzteres bewirkt in Folge auch eine kaskadierende Invalidierung der weiteren Zwischenspeicher.

Verteilter binärer Objektspeicher *memcached* Bei *Memcached*[36] handelt es sich um einen Objektspeicher, der universell eingesetzt werden kann, um (meist binäre) Objekte in einem verteilten, schnellen Zwischenspeicher abzulegen. Dazu wird der Hauptspeicher der Systeme, auf denen der *Memcached*-Dienst läuft, eingesetzt.

Eine mögliche Anwendung dieses Prinzips ist das Zwischenspeichern von Datenbank-Anfragen. Wikipedia setzt *Memcached* zusammen mit dem von diesem abgeleiteten *Tugela Cache* ein ([Wik06e]). Bei diesem handelt es sich um einen Zwischenspeicher, der aus *Memcached* durch die Kombination mit einem Berkley Datenbank System (*Berkley DataBase*)[37] als Hintergrundspeicher einsetzt. Dieses ermöglicht es, die Zwischenspeicherung der Objekte teilweise aus dem Hauptspeicher in das entsprechende DBMS auf Festplattenspeicher zu verlagern.

Wikipedia verwendet dieses System, um die vom Parser analysierten und in HTML übersetzen Seiten (inklusive eventueller Bildinhalte) zwischenzuspeichern. Dies vermeidet weitere Fälle, in denen der Parser alle Übersetzungsschritte durchlaufen muss. Dies Gesamt-Trefferrate für die Zwischenspeicher verbessert sich dadurch um ca. 7%.

Zwischenspeicher für den PHP-Interpreter Bei PHP handelt es sich um eine sog. Interpreter- oder Skriptsprache, d.h. bei jedem Aufruf eines Skripts mit

[35] das Setzen des Parameters „*action=purge*“ in der Artikel-URL bewirkt den Aufruf der entsprechenden Funktion „*purge()*“

[36] Memcached, `http://www.danga.com/memcached`

[37] Berkley Database, `http://en.wikipedia.org/wiki/Berkeley_DB`

PHP-Anweisungen muss diese durch den Interpreter in ausführbaren Bytecode übersetzt werden. Um zu verhindern, dass dieser Vorgang für dasselbe Skript mehrfach ausgeführt werden muss, ohne dass sich an diesem etwas geändert hat, existieren verschiedene Zwischenspeicherlösungen, welche die Ausführungszeit eines Skripts in diesen Fällen verbessern.

Wikipedia setzte dabei zunächst auf *Turck MM Cache*[38] ([Wik06c]), dessen Weiterentwicklung jedoch eingestellt wurde (das Projekt wird inzwischen unter dem Namen *eAccelator*[39] weitergeführt). Derzeit setzt Wikipedia auf den u.a. von George Schlossnagle entwickelten *Alternative PHP Cache (APC)*[40].

Es existieren seitens Wikipedia keine genauen Daten über die erreichte Leistungssteigerung, [Kir05] weist jedoch für ein Szenario einer Nachrichtenseite mit hoher Last Leistungssteigerungen zwischen 10% und 500% im Falle des Einsatz des *APC* gegenüber einer identischen Lastsituation ohne einen Interpreter-Zwischenspeicher aus, ohne dass es zu einer signifikante Erhöhung des Speicherbedarfs kommt.

3.3 Partizipative semantische Portale

Ergänzt man das partizipative Portal im Szenario (siehe Abschnitt 3.1) um die Fähigkeit, semantische Information zu speichern, so müssen die Betrachtungen um semantische Anfragen und semantische Änderungen erweitert werden. Sprachlich präziser müsste von einer Anfrage an den bzw. einer Änderung im Datenspeicher für semantische Informationen gesprochen werden, der Einfachheit halber soll jedoch die gewählte Formulierung beibehalten werden.

Dazu sei S die Menge der durch Seitenabruf ausgelösten semantischen Anfragen (*Semantic queries*), D die Menge der semantischen Änderungen (*semantic Data changes*).

Abzählen semantischer Anfragen und Änderungen Wie im Abschnitt 3.1.2 bezeichne $P = \{p_1, \ldots, p_q\}$ die Menge der Seiten. Es sei weiter die Funktion $r^* : P \longmapsto 2^S$ so definiert, dass sie jeder Seite p_i die Menge der semantischen Anfragen $\{s_{i_1}, \ldots, s_{i_j}\} \subset S$ zuordnet, welche durch ihren Abruf (*request*) entstehen.

Das Abzählen der semantischen Anfragen lässt sich zwar formal auch über Seitenabrufe definieren[41], jedoch kann dies auch einfacher über die Anfragen selbst geschehen, indem beispielsweise die Zeichenrepräsentation der Anfragen als Unterscheidungsmerkmal herangezogen und lexikographisch sortiert wird.

Nachfolgend wird daher die Darstellung $S = \{s_1, .., s_s\}$ mit $s = |S|$ verwendet, ohne die Aufzählung formal näher zu spezifizieren. Eine Möglichkeit, um semantische Anfragen zu indexieren, ist die in 4.4.2 diskutierte Verwendung einer Hashfunktion.

Analog zu der Forderung für partizipative Portale, dass der Abruf unterschiedlicher Seiten auch zu unterschiedlichen Anfragen an den Datenspeicher

[38] Turck MM Cache, `http://turck-mmcache.sourceforge.net/index_old.html`

[39] eAccelator, `http://www.eaccelerator.net/`

[40] Alternative PHP Cache, `http://pecl.php.net/package/APC`

[41] Hierzu muss zunächst der Index durch ein Paar (i, j), wobei i der Seitenindex derjenigen Seite p_i ist, deren Abruf die semantische Anfrage $s_{(i,j)}$ ausgelöst hat und j die semantischen Anfragen innerhalb dieser Seite abzählt. Die Menge dieser Paare muss dann wieder in die natürlichen Zahlen abgebildet werden

führen soll, wird auch für semantische Anfragen die Betrachtung dergestalt eingeschränkt, dass jeweils unterschiedliche Seiten zu unterschiedlichen semantischen Anfragen führen, d.h.

$$\forall i, j \in \mathbb{N}_{\shortparallel} : i \neq j \Longrightarrow r^*(p_i) \cup r^* p_j = \emptyset.$$

Es sei weiter die Funktion $e^* : P \longmapsto 2^D$ so definiert, dass sie jeder Seite p_i die Menge der möglichen semantischen Änderungen $\{d_{i_1}, \ldots, d_{i_j}\} \subset D$ zuordnet, welche durch Bearbeitung (*edit*) dieser Seite entstehen können (die semantische Änderung d_i sagt dabei lediglich aus, dass die betreffende Information geändert wird, nicht von welchem Typ (Einfügen, Aktualisieren, Löschen) die Änderung ist).

Auf eine formale Herleitung der Aufzählbarkeit der Änderungen wird wiederum verzichtet - sie kann praktisch wieder durch Verwendurch der Zeichenrepräsentation der entsprechenden semantischen Information geschehen, die von der Änderungen betroffen wären. Nachfolgend kann daher $D = \{d_1, .., d_d\}$ mit $d = |D|$ geschrieben werden.

Für spätere Betrachtungen wird davon ausgegangen, dass jeweils nur eine einzelne semantische Information mit jeder Bearbeitung verändert wird. Mehrere Änderungen können durch wiederholtes Ausführen der für eine einzelne Änderung notwendigen Schritte beschrieben werden.

Häufigkeit von Anfragen und Änderungen Es sei analog zu Abschnitt 3.1.2 die Funktion $\#_S : S \longmapsto \mathbb{N}^0$ für semantische Anfrage derart definiert, dass $\#_S(s_j) = \#_P(p(i)) = n_i$, falls $s_j \in r^* p(i)$, d.h. wenn die semantische Anfrage s_j durch Aufruf der Seite p_i entsteht.

Weiter bezeichne $\#_D : D \longmapsto \mathbb{N}^0$ durch $\#_D(d_i) = \#_D(e^*(p_i) \cup \{d_i\}) = l_i$ die Anzahl der Änderungen an einer Seite, bei der genau die mögliche semantische Änderung d_i durchgeführt wird.

Wiederum wird abkürzend nur noch die Funktion # verwendet, solange aus dem Kontext erschlossen werden kann, ob es sich um Seiten, Anfragen oder Änderungen handelt und ob die semantische oder die konventionelle (nichtsemantische) Variante gemeint ist.

Fazit Zusammenfassend lässt sich für die semantischen Anfragen und Änderungen sagen:

1. Semantische Anfragen und Änderungen sind abzählbar, jedoch sind ohne weitere Annahmen keine direkten Vergleiche mit den konventionellen Anfragen und Änderungen möglich:
   - $S = \{s_1, .., s_s\}$
   - $D = \{d_1, .., d_d\}$
2. Geht man davon aus, dass das Verhältnis der Anzahl der semantischer Änderungen zur Anzahl der ausgelösten semantischen Anfragen im Wesentlich identisch mit dem konventionellen Fall ist, so ist die Gesamtanzahl der semantisch Anfragen deutlich größer als die der semantischen Änderungen:

$$\sum_{i=1}^{s} \#s_i \gg \sum_{i=1}^{d} \#d_i$$

3.4 Szenario: Semantic Wikipedia

Wie bereits in Abschnitt 2.5.2 erläutert, handelt es sich bei *Semantic MediaWiki* um eine Erweiterung von *MediaWiki*. Nachdem im vorigen Abschnitt die Architektur der *Wikipedia* näher betrachtet wurde, soll nun eine Analyse stattfinden, mit welchen Elementen *Semantic MediaWiki* die Architektur der *Wikipedia* zu jener der *Semantic Wikipedia* ergänzt und in welchem Maße neue, potentielle Leistungsengpässe entstehen.

3.4.1 Architektur

Im Kern fügt *Semantic MediaWiki* der ursprünglichen *MediaWiki*-Software als entscheidende Änderung die Funktionalität der *semantischen Annotation* hinzu (siehe Abschnitt 2.5.2). Zur Speicherung dieser Daten wird derzeit noch eine Reihe von zusätzlichen Tabellen in der MySQL-Datenbank eingesetzt. Mittelfristig ist aber zur Realisierung des Projektes „*Semantic Wikipedia*“ die Speicherung der semantischen Daten in einem RDF-DBMS geplant ([VKV+06a]).

Abbildung 8 illustriert diese erweiterte Architektur der *Semantic Wikipedia*, der Einfachheit halber wird die Betrachtung von Mechanismen der dynamischen Lastverteilung und Datenbankreplikation ausgeblendet und für die einzelnen Bausteine der Architektur jeweils angenommen, dass sie nur einfach vorhanden sind. Pfeile symbolisieren dabei einen Informationsfluss in beide Richtungen (im Falle des gestrichelten Pfeils zum RDF-DBMS sind es semantische Informationen), wobei das *Semantic MediaWiki*-Element stets das steuernde ist. Die enge Verknüpfung zwischen diesem und dem PHP-(Interpreter-)Zwischenspeicher symbolisiert dabei, dass diese beiden Elemente *zwingend* auf dem selben System installiert sein müssen, um überhaupt eine Nutzbarkeit des Zwischenspeichers zu ermöglichen.

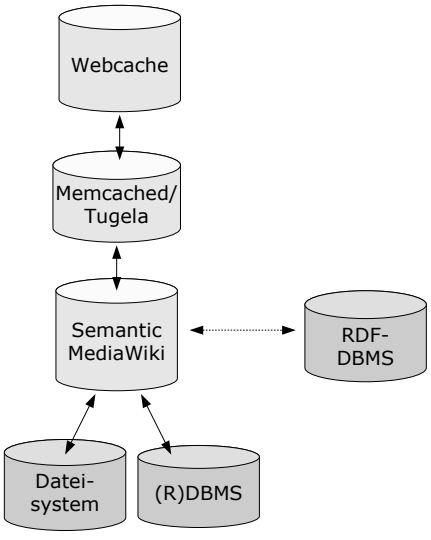

Abbildung 8: Schematische Darstellung der *Semantic Wikipedia*-Architektur

In dieser schematisierten Darstellung wird deutlich, dass im Vergleich zum

vorherigen Szenario der Wikipedia (Abschnitt 3.2) einzig das RDF-DBMS als neues Element der System-Architektur hinzugekommen ist. Daher soll nachfolgend anhand der Analyse der Prozesse für die Fälle des Einfügens, Löschens und Suchens semantischer Informationen die Rolle des RDF-DBMS näher betrachtet werden. Der Vorgang der Aktualisierung von semantischer Informationen wird nicht näher betrachtet, da er über eine Kombination von Löschen und Einfügen ersetzt werden kann.

3.4.2 Prozess-Beschreibung

Bei den folgenden Beschreibungen werden jene Schritte, die analog auch beim alleinigen Einsatz von *MediaWiki* vorkommen, verkürzt dargestellt, da ihr Aufwand sich bei späteren Laufzeit-Betrachtungen herauskürzt. Auch verläuft die tatsächliche Abarbeitung derzeit stärker integriert als dargestellt; zur besseren Herausarbeitung der neuen Strukturen wird jedoch in den folgenden Abbildungen zwischen den bereits in *MediaWiki* vorhandenen Elementen und jenen von *Semantic MediaWiki* unterschieden.

Einfügen Wie in Abschnitt 2.5.2 dargelegt, existieren prinzipiell zwei Arten von Annotationen, mit denen semantische Informationen über die auf einer Seite beschriebenen Entität zu de vorhandenen textuellen hinzugefügt werden können:

1. Typisierte Verweise
2. Attribute

Während typisierte Verweise direkt in den Datenspeicher übertragen werden können, muss bei Attributen noch geprüft werden, ob diese den gewünschten Typ haben. Bei komplexeren Typen wie Geokoordinaten ist es außerdem notwendig, die vom Benutzer eingegebenen Zeichen mit einer entsprechenden Konvertierungsroutine in das Zielformat zu übertragen (für gebräuchliche Datentypen wie Ganzzahlen oder Gleitkommazahlen übernimmt dies in der Regel bereits die Programmiersprache oder das DBMS). Der entsprechende Prozessschritt entfällt offensichtlich für typisierte Verweise. Die schematische Darstellung eines Einfüge-Vorgangs findet sich in Abbildung 9.

Löschen Für das Löschen ist zu beachten, dass es aufgrund der Konzeption der Benutzereingaben (die Eingabe und Veränderung sowie die semantische Annotation von Information soll für den Benutzer so einfach wie möglich sein; dies impliziert unter anderem, dass das Löschen von Annotationen nicht explizit kenntlich gemacht werden muss) nur implizit über das Fehlen einer Annotation festgestellt werden kann. Es muss also entweder der letzte Stand festgehalten worden sein, beispielsweise stets nach dem Speichern einer Seite, um einen Vergleich anstellen zu können, oder es müssen bei jedem Speichervorgang sämtliche semantischen Informationen zunächst gelöscht und die noch vorhandenen neu eingefügt werden[42]. Die schematische Darstellung eines Lösch-Vorgangs findet sich in Abbildung 10.

[42] Dies ist in den Abbildungen nicht explizit aufgeführt. Derzeit wird letztere Methode verwendet, der Umstieg auf erstere ist jedoch in Planung

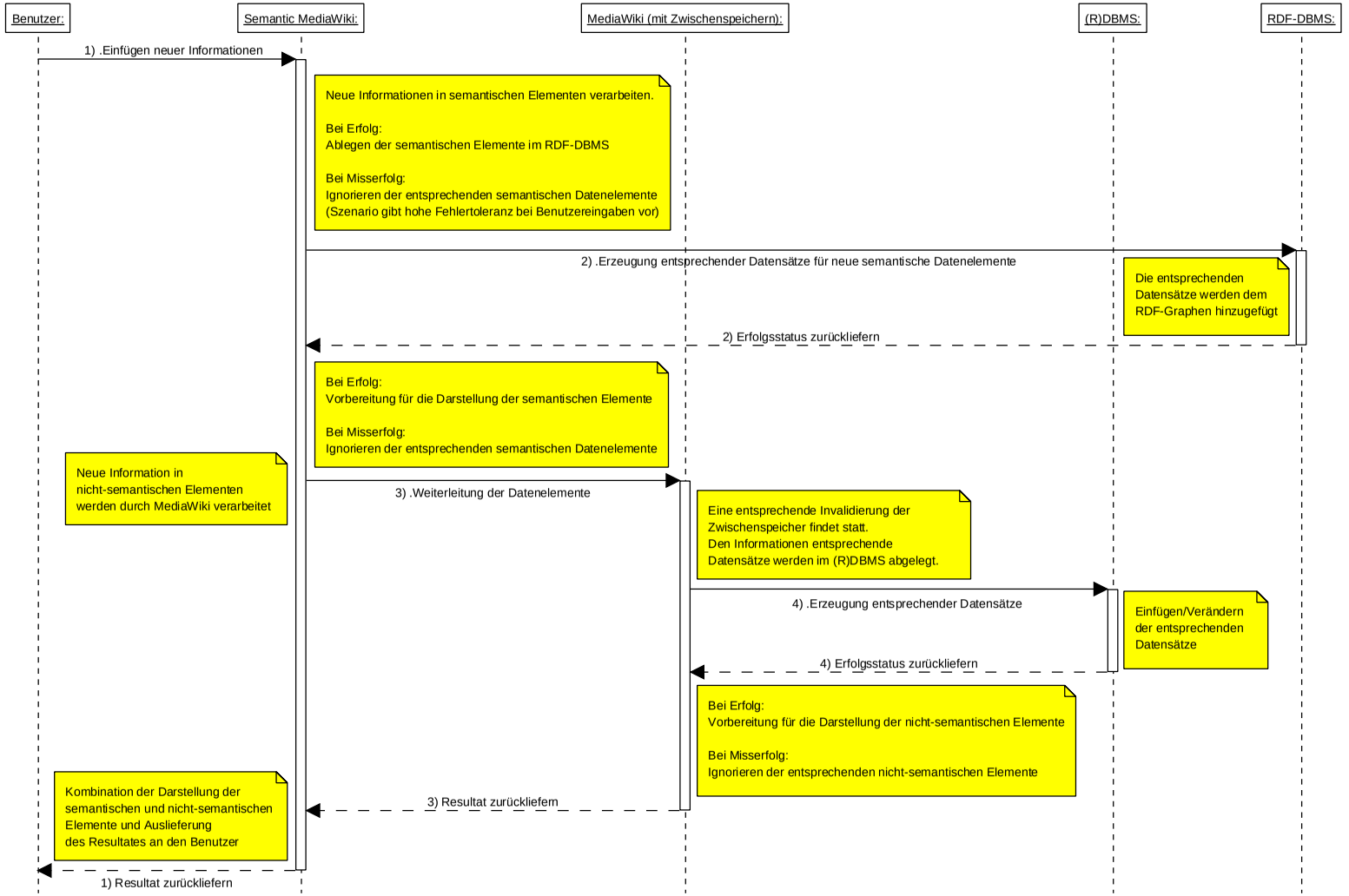

Abbildung 9: Schematische Darstellung der Vorgänge beim Einfügen neuer semantischer Information

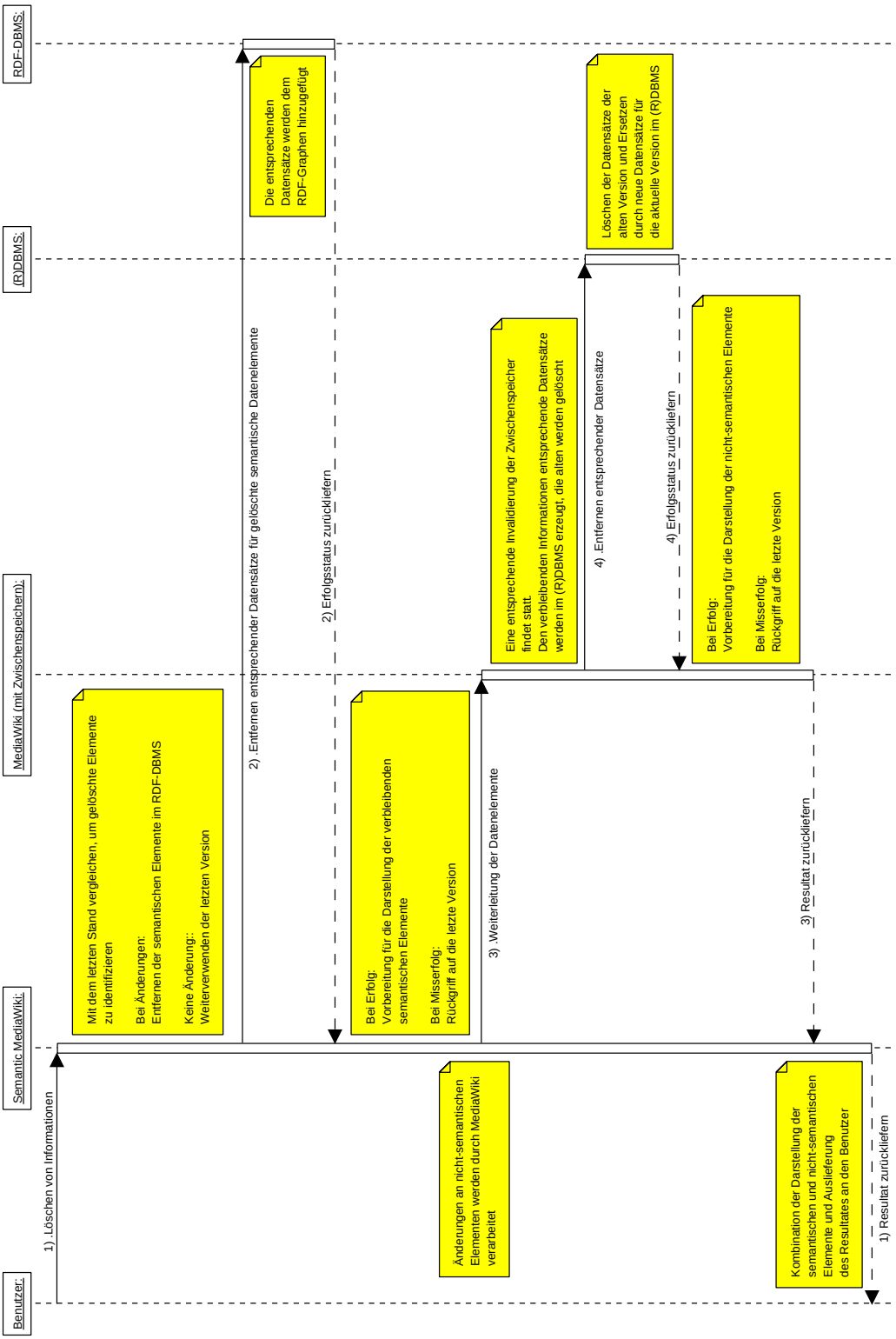

Abbildung 10: Schematische Darstellung der Vorgänge beim Löschen semantischer Information

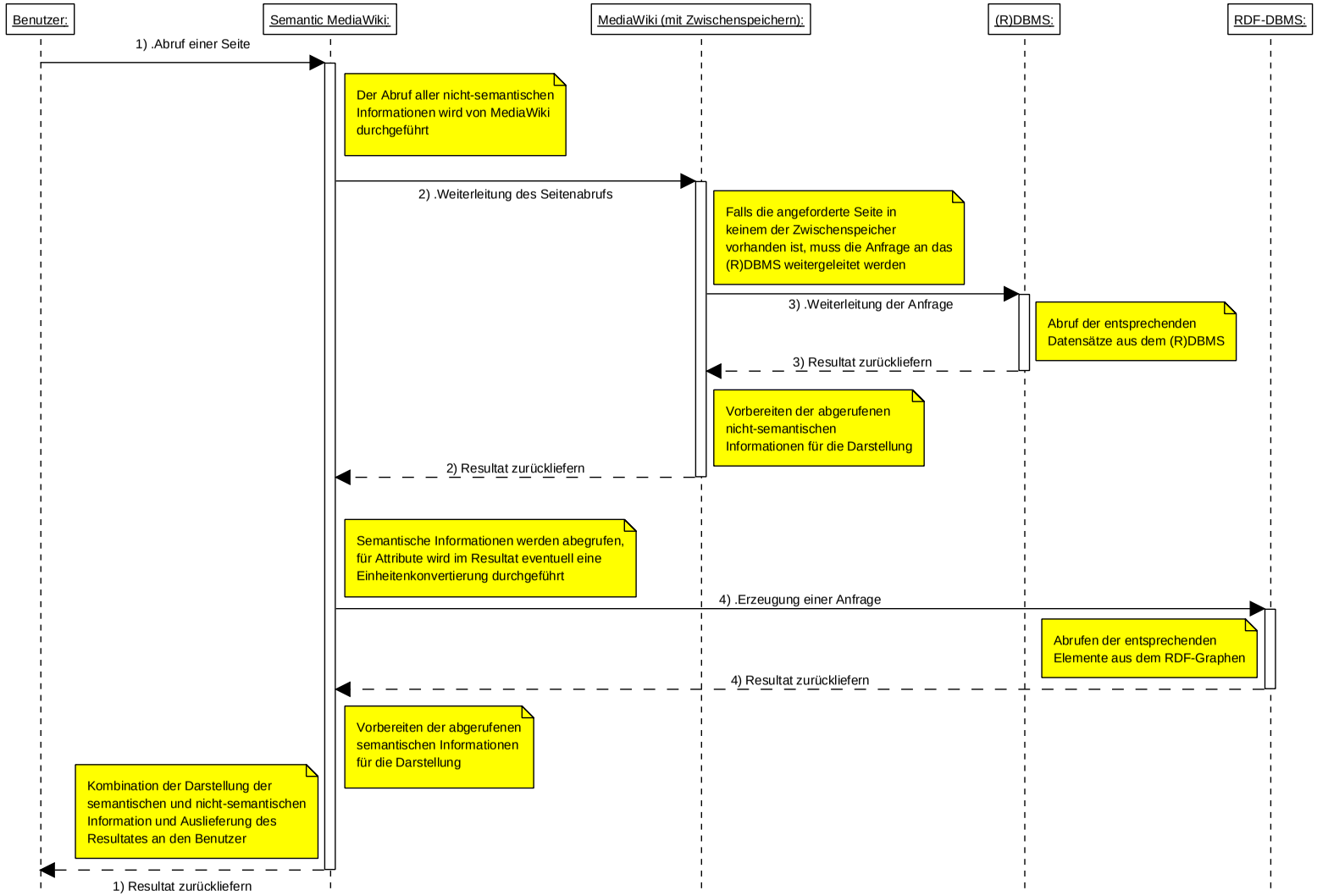

Abbildung 11: Schematische Darstellung der Vorgänge beim Abruf einer Seite mit integrierter Suchanfrage

Abrufen/Suchen Im Falle eines Suchvorgangs soll die Betrachtung hier auf eine Seite mit einer integrierten Suchanfrage in SMW-QL (siehe Abschnitt 2.5.2) reduziert werden. Prinzipiell verlaufen Anfragen aus der einfachen semantischen Suche analog, gehen jedoch von einer Spezial-Seite[43] aus und benötigen einige Prozessschritte weniger als die integrierten Suchanfragen. Die schematische Darstellung einer solchen in eine Seite integrierten Suchanfrage findet sich in Abbildung 11.

3.4.3 Implikationen für das Leistungsverhalten

Bereits in 3.4.1 wurde festgestellt, dass der Einsatz eines RDF-DBMS den maßgebliche Unterschied zwischen den Architekturen der *Wikipedia* und der des Projektes *Semantic Wikipedia* darstellt. Die Konkretisierung der Prozesse, insbesondere desjenigen, der durch die Verwendung von integrierten Suchanfragen entsteht, zeigt zudem auf, dass das RDF-DBMS bei jedem Vorgang benötigt und belastet wird.

Zusätzliche Last Zieht man die Ergebnisse von Abschnitt 3.1.2 heran, so kann man die entstehende Last einfach konkretisieren. Denn mit denselben Bezeichnungen lässt sich der zusätzliche Aufwand beschreiben durch die Summe der Ausführungszeiten der semantischen Anfragen t_{s_i} und der Ausführungszeiten der semantischen Änderungen t_{d_j}, jeweils gewichtet durch die entsprechende Auftrittshäufigkeit.

Der zeitliche Aufwand für die Bearbeitung der konventionellen Anfragen respektive Änderungen hebt sich weg[44]:

$$T = \sum_{i=1}^{s} \#(s_i) \cdot t_{s_i} + \sum_{j=1}^{d} \#(d_i) \cdot t_{d_i}$$

Es fällt auf, dass die zusätzliche Last proportional zur Anzahl der Seitenabrufe steigt.

Konflikte mit der bestehenden Architektur Betrachtet man die Architektur der *Wikipedia* (siehe Abschnitt 3.2), so fällt auf, dass zunächst auch die Resultate der Anfragen in der resultierenden HTML-Seite gespeichert werden und somit von den bestehenden Mechanismen profitieren. Um jedoch die Aktualität der Anfrage-Resultate zu garantieren, muss der Zwischenspeicher bei jeder Änderungen semantischer Daten invalidiert werden.

Dies wirft ein weiteres Problem auf, welches die bereits bestehende Architektur in ihrer Wirkung einschränkt. Da, ohne eine entsprechende Entscheidungslogik, jede Änderung semantischer Information potentiell jede Anfrage betreffen kann, müssten auch bei jeder Änderung alle Zwischenspeicher für Seiten, die Anfragen enthalten, oder wenn eine entsprechende Information nicht zur Verfügung steht, sogar aller Seiten invalidiert werden.

[43] Einfache semantische Suche, `http://ontoworld.org/wiki/Special:SearchTriple`

[44] formal erfolgt die Berechnung des Mehraufwand durch $T = T_{\text{neu}} - T_{\text{alt}}$ und Einsetzen der entsprechenden Werte

3.5 Anforderung an eine Lösung zur Anfrageoptimierung

Das Ziel nachfolgender Untersuchungen ist also, eine entsprechende Lösung zu entwerfen, die den Mehraufwand zur Beantwortung semantischer Anfragen reduziert und gleichzeitig größtmögliche Aktualität der Resultate gewährleistet, ohne die bisherigen leistungssteigernden Maßnahmen signifikant zu beeinträchtigen. Dies soll in den nachfolgenden drei Kriterien festgehalten werden, die eine solche Lösung erfüllen muss.

K1 Verbesserung der mittleren Antwortzeit bei der Anfragebeantwortung

K2 Garantie aktueller Anfrageergebnisse (korrekte Invalidierung bei Änderungen)

K3 Integration in die bestehende Architektur; keine (oder möglichst geringe) negative Beeinflussung der bestehenden Architektur

4 Lösungsentwurf

Dieses Kapitel diskutiert einen Entwurf zur Lösung des Problemes, die Beantwortung der im vorigen Kapitel diskutierten Anfragen performant zu gestalten. Erklärtes Ziel des *Semantic MediaWiki* Projektes ist es, die Speicherung der semantischen Daten in einem geeigneten RDF-DBMS durchzuführen.

In der ersten Hälfte wird zunächst durch einen Vergleich der in *Semantic Mediawiki* integrierten Anfrage-Sprache (*SMW-QL* [VKV+06b]) mit SPARQL ([P05]) gerechtfertigt, für weitere Betrachtungen ausschließlich SPARQL-Anfragen an ein geeignetes RDF-DBMS zu betrachten. Damit wird die Lösung über den konkreten Anwendungsfall hinaus für all jene Szenarien einsetzbar, die ein solches RDF-DBMS einsetzen.

Nachfolgend werden die in Abschnitt 2.1 vorgestellten Optimierungsmethoden im Hinblick auf das Szenario diskutiert und es wird begründet, warum ein Zwischenspeicher (*Cache*) als Lösungsansatz favorisiert wird. Im Anschluss werden die Anforderungen für einen solchen definiert.

In der zweiten Hälfte werden die Entwurfsentscheidungen dokumentiert, die bei der Umsetzung dieser Anforderungen getroffen wurden. Eine Sonderrolle nimmt dabei die intelligente Invalidierung des Zwischenspeichers ein - ihre Umsetzung wird eigens in Kapitel 5 beschrieben.

Abschließend werden die Ergebnisse zusammengefasst und der Stand der tatsächlichen Implementierung beschrieben.

4.1 Vergleich: SMW-QL und SPARQL

Die Anfragensprache SMW-QL wurde zur Umsetzung integrierter Anfragen innerhalb einer *Semantic MediaWiki*-Installation entworfen. Im Hinblick auf das Projekt *Semantic Wikipedia* war dabei besonderes die einfache Zugänglichkeit für Benutzer ausschlaggebendes Entwurfskriterium ([VKV+06b]).

In der derzeit aktuellen Version des *Semantic MediaWiki* wird die Speicherung semantischer Informationen noch in der Tabelle eines RDBMS vorgenommen, so dass auch die integrierte Suche auf diese zugreift. Mittelfristig ist jedoch die Verwendung eines RDF-DBMS zu diesem Zweck vorgesehen. Mit SPARQL (siehe Abschnitt 2.2.2) existiert eine vom W3C standardisierte Anfragesprache für diese Datenbanksysteme; dies motiviert die Untersuchung, ob eine vollständige Umsetzung der in SMW-QL formulierbaren Anfragen in SPARQL möglich ist. Dazu sollen die verfügbaren Ausdrücke von SMW-QL schrittweise analysiert und beispielhaft in SPARQL transformiert werden. Die möglichen Typen von Ausdrücken sind dabei [Ont06] entnommen, die korrespondierenden SPARQL-Anfragen wurden nach [P05] und Abschnitt 2.2.2 konstruiert (ausgeblendet werden Aspekte, die nur die Darstellung innerhalb der resultierenden HTML-Seite bestimmen, beispielsweise, welche Elemente als Verweise dargestellt werden).

4.1.1 Einfache Anfragen

Eine einfache Anfrage, welche nach allen Ressourcen (Seiten) fragt, die Schauspieler beschreiben, die in Boston geboren *und* zwischen 6 und 7 Fuß groß sind[45],

[45] Intern wird eine Konvertierung ins metrische System vorgenommen, so dass z.B. auch Schauspieler mit einer Größe von 1,95 Metern in die Suche einbezogen werden. Dies soll aber hier nicht näher betrachtet werden

lautet in SMW-QL:

```
<ask>
  [[Category:Actor]]
  [[born in::Boston]]
  [[height:=>6 ft]]
  [[height:=<7 ft]]
</ask>
```

Dabei werden alle durch die Anfrage abgerufenen Informationen angezeigt (also auch die Größe). Ein entsprechendes SPARQL-Äquivalent lässt sich einfach konstruieren:

```
SELECT
  *
WHERE
{
  ?x  :Category  :Actor .
  ?x  :bornIn    :Boston .
  ?x  :height    ?h .
}
FILTER
(
  ?h  > 6 && ?h < 7
)
```

Es können also bei Anfragen in denen Bedingungen in konjunktiver Form gegeben sind, alle Relationen $1:1$ in SPARQL-Bedingungen umgesetzt werden, für Attributwerte muss neben der Aufführung in den Bedingungen[46] zusätzlich eine Wertbeschränkung über *FILTER* durchgeführt werden.

4.1.2 Modifikatoren für Suchbedingungen und Anzeige

SMW-QL stellt verschiedene Ausdrücke zur Verfügung, um sowohl die Suchbedingungen als auch die angezeigten Informationen zu beeinflussen.

Auswahl aller Ressourcen mit einer Relation Die Verwendung von „+“ bietet die Möglichkeit, alle Ressourcen zu selektieren, für welche die gegebene Relation definiert ist. Die SMW-QL-Anfrage

```
<ask>
  [[born in::+]]
</ask>
```

enspricht demnach der SPARQL-Anfrage:

```
SELECT
  *
WHERE
{
  ?x  :bornIn    ?y
}
```

Die Übersetzung erfolgt also schlicht durch Hinzufügen einer weiteren ungebunden Variable ohne weitere Beschränkungen.

[46] mit einem Platzhalter für den eigentlichen Wert

Disjunktionen Disjunktionen werden mittels „//“ formuliert und erlauben Anfragen, die (mindestens) eine von mehreren möglichen Wertbedingungen erfüllen. Dies lässt sich sowohl auf Kategorien als auch auf Relationen anwenden (die Unterstützung für Attribute ist derzeit nicht implementiert, dort können Wertbeschränkungen nur konjunktiv verknüpft verwendet). In SPARQL überträgt sich die disjunktive Verknüpfung für Relationen in den Wertbeschränkungs-Abschnitt, für Kategorien kann sie durch „*UNION*“ ersetzt werden. So wird

```
<ask>
  [[Category:Musical actor||Theatre actor]]
</ask>
```

zu:

```
{
  SELECT
    *
  WHERE
  {
    ?x  :Category   "Musical actor"
  }
}
UNION
{
  SELECT
    *
  WHERE
  {
    ?x  :Category   "Theatre actor"
  }
}
```

Optionale Informationen Die Verwendung von „*“ dient dazu, die Anzeige einer Information, welche nicht immer vorhanden ist, herbeizuführen. Ihre Übersetzung ist die optionale Klausel in SPARQL. Die Beschreibung ihrer Verwendung findet in Listing 3 zusammen mit derjenigen von verschachtelten Anfragen statt.

4.1.3 Begrenzung der Anzahl der Resultate

Neben den Parametern „*sort*“ und „*link*“, welche allein der Ausgabeformatierung dienen, existiert mit dem Parameter „*limit*“ eine Möglichkeit, die Anzahl der Resultate zu begrenzen. Er ist in SPARQL identisch vorhanden und kann daher nahezu unverändert übernommen werden[47]. Der Parameter wird in Listing 3 verwendet.

4.1.4 Verschachtelte Anfragen

Im Kern handelt es sich bei *Semantic MediaWiki* um eine Erweiterungen des *Mediawiki*-Parsers. Dieser erlegt dem System jedoch Beschränkungen auf, da er nicht mit verschachtelten Auszeichnungselementen umgehen kann. Man muss

[47] SMW-QL verwendet ein Gleichheitszeichen für die Angabe, SPARQL einen gesonderten Block und eine Trennung durch Leerzeichen

man sich daher mit der Verwendung eines weiteren Auszeichnungselementes behelfen: „<q>“ (Die maximale Verschachtelungstiefe ist somit auf 2 festgelegtj).

Das folgende komplexere Beispiel stellt eine Anfrage dar, welche alle Städte in der Europäischen Union und ihre Bevölkerung, sofern diese 300.000 übersteigt, ausgibt. Zudem wird optional der entsprechende Bürgermeister ausgegeben. Die Anzahl der Resultate wird auf 10 begrenzt:

Listing 2: Eine komplexere SMW-QL-Anfrage

```
<ask sort="founded in" limit="10" link="all">
  [[Population::>300000]]
  [[Category:City]]
  [[Mayor::*]]
  [[located in::<q>[[Category:Country]] [[part of::EU]]</q>]]
</ask>
```

Die entsprechende SPARQL-Übersetzung erhält man durch:

Listing 3: Eine komplexere SMW-QL-Anfrage

```
SELECT
  *
WHERE
{
  ?x        :Category    :City .
  ?x        :locatedIn   ?y .
  ?y        :Category    :Country .
  ?y        :partOf      :EU .
}
OPTIONAL
{
  ?x        :population ?pop .
  ?major    :major      ?x
}
FILTER
(
  ?pop > 3000000
)
LIMIT 10
```

Verschachtelte Bedingungen können also durch Hinzufügen einer weiteren Variable mit Bedingungen, die aus der Unter-Anfrage in SMW-QL resultieren, in SPARQL übersetzt werden.

4.1.5 Fazit: SMW-QL in SPARQL transformierbar

Eine Übersetzung der Ausdrücke von SMW-QL in SPARQL ist, wie obige Betrachtungen zeigen, ohne Einschränkung möglich. Daher kann die Betrachtung semantischer Anfragen nachfolgend in SPARQL-Darstellung geschehen. Dies ermöglicht es zudem, die gewonnenen Resultate in den allgemeinen Kontext se-

mantischer partizipativer Portale zu stellen statt sie nur auf den konkreten Anwendungsfall zu beschränken.

4.2 Bewertung der Optimierungsmöglichkeiten

Wie durch den Vergleich in Abschnitt 4.1 deutlich wurde, ist eine Transformation von SMW-QL nach SPARQL ohne Einschränkungen in der Ausdrucksmächtigkeit möglich. Es kann also im folgenden davon ausgegangen werden, dass bei dem verwendeten Datenspeicher um ein RDF-DBMS handelt, welches in der Lage ist, Anfragen in SPARQL zu interpretieren.

Da die effiziente Speicherung von RDF-Tripeln Gegenstand aktueller Forschung ist (siehe u.a. [SK06], [BO04], [App]) und sich noch kein „Standard"-RDF-DBMS herauskristallisiert hat, ist es Ziel des Lösungsentwurfs, möglichst keine Annahmen über die Struktur des Datenspeichers zu machen und von konkreten Produkten zu abstrahieren.

Inbesondere unter diesem Gesichtspunkt sollen nun die in Kapitel 3 vorgestellten Optimierungsmöglichkeiten auf Tauglichkeiten im Kontext dieser Arbeit untersucht werden.

4.2.1 Umschreiben von Anfragen (*Query rewriting*)

Die Möglichkeit, Anfragen durch Transformationen in äquivalente mit günstigerem Ablaufplan zu überführen, erweist sich für das gegebene Szenario als untauglich. Sowohl Ablaufpläne als auch deren Kostenberechnung basieren auf Annahmen, die in der physikalischen Organisation der Daten begründet liegen. Diese sind aber DBMS-spezifisch und somit nicht ohne weiteres abstrahierbar (siehe Abschnitt 2.1.1).

Auch das Prinzip der Minimierung von Zwischenergebnissen, welches im Bereich der RDBMSe zur Bestimmung günstiger Ablaufpläne eingesetzt wird, kann nicht direkt auf ein RDF-DBMS übertragen werden. In letzteren ist eine Suche gleichbedeutend mit dem Finden eines isomorphen Teilgraphen (siehe Abschnitt 2.2.2). Bei diese vom relationalen Fall verschiedenen Herangehensweise, treten Teilergebnisse nicht in analoger Weise auf.

Zwar verwenden mehrere RDF-DBMSe relationale DBMSe im Hintergrund ([App], [HS05], [BO04], [SK06]), jedoch unterscheiden sich die verwendeten Organisationsstrukturen mitunter deutlich. Eine Optimierung auf der DBMS-Ebene wäre wiederum datenspeicherspezifisch und somit an ein bestimmtes Produkt gebunden.

4.2.2 Optimierte Index-Strukturen

Im Falle der Index-Strukturen gelten im Wesentlichen die eben angeführten Argumente, da eine optimierte Index-Struktur immer nur relativ zum jeweils gewählten Datenspeicher existiert. Zwar beschäftigt sich [HD05] mit optimierten Index-Strukturen, jedoch wiederum spezifisch für den dort vorgestellten RDF Datenspeicher.

4.2.3 Zwischenspeicher (*Caches*)

Wie bereits im Abschnitt 2.1.4 diskutiert, stellen Zwischenspeicher eine flexible und etablierte Möglichkeit dar, sowohl die Antwortzeit auf eine Anfrage als auch

die Systemlast signifikant zu verringern, wenn es gelingt, die entsprechenden Informationen vorzuhalten.

Speziell in einem Szenario wie dem hiesigen, in dem potentiell viele gleiche Anfragen gestellt werden, ohne das sich die betroffenen Daten häufig ändern, verspricht ein Cache, das Leistungsverhalten des Gesamtsystems positiv zu beeinflussen. Weiterhin kann die Invalidierungslogik des Zwischenspeichers genutzt werden, um eine negative Beeinflussung der bisherigen leistungssteigendern Maßnahmen (siehe Abschnitt 3.4.3) zu verhindern.

4.3 Anforderungen an den Zwischenspeicher

Resultat von Abschnitt 4.2 war, dass die Implementierung eines Zwischenspeichers die einzig gangbare Optimierungsmöglichkeit ist, wenn man eine hinreichende Unabhängigkeit vom verwendeten Datenspeicher erreichen will. Es ist nun zunächst zu untersuchen, welche Entwurfsentscheidungen zur Realisierung eines solchen Zwischenspeichers zu treffen sind.

Dem Problem inhärent ist die Frage, wie der Zwischenspeicher in die bestehende Architektur integriert werden kann. Dies soll nachfolgend geschehen.

4.4 Entwurfsentscheidungen

Unter Rückgriff auf die Definition eines Zwischenspeichers in Abschnitt 2.1.4 stellt man fest, dass weitere Entwurfsentscheidungen getroffen werden müssen, um eine entsprechende RDF-ZSE zu realisieren.

- Wie integriert sich der Zwischenspeicher in die Architektur?
- Welche Informationen sollen gespeichert werden?
- Wie wird auf Änderungen reagiert?
- Welche Verdrängungsstrategie wird gewählt?

Besonders der letzte Punkt ist hier von Bedeutung. Aufgrund des Szenarios müssen hier sowohl Änderungen am Daten-Schema als auch an den tatsächlichen Instanzen betrachtet werden. Er wird gesondert in Abschnitt 5 betrachtet.

4.4.1 Integration in die SMW-Architektur

In Abschnitt 3.4 wurde bereits festgestellt, dass der zu erwartende Leistungsengpass im Zugriff auf das RDF-DBMS liegt, welches die aus semantischen Annotationen gewonnen Informationen speichert und für Anfragen bereithält. Diese Architektur lässt im Wesentlichen die Platzierung einer entsprechenden Zwischenspeicher-Einheit nur an zwei Orten zu:

1. In einer dem RDF-DBMS vorgelagerten Einheit
2. In direkter Koppelung mit dem RDF-DBMS

Wie jedoch in Abschnitt 4.2 festgehalten, ist es zentrales Entwurfskriterium, von konkreten RDF-DBMS zu abstrahieren. Dies legt den Entwurf auf Möglichkeit 1 fest. Die Integration des Zwischenspeichers findet also zwischen dem *Semantic MediaWiki*-Kern und dem RDF-DBMS statt (Abbildung 12 zeigt den

entsprechenden Ausschnitt aus der Architektur, die nicht sichtbaren Elemente entsprechen denen aus Abbildung 8). Dies erlaubt es zusätzlich, die bisherigen, leistungssteigernden Maßnahmen der ursprünglichen Architektur beizubehalten.

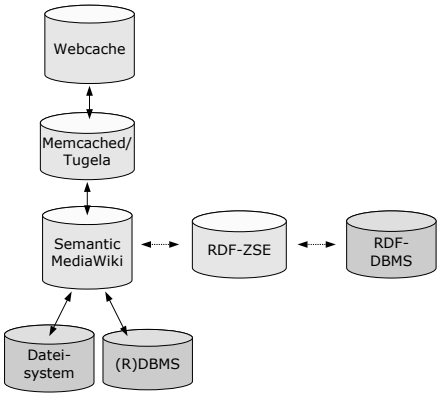

Abbildung 12: Ausschnitt der modifizierten Architektur der *Semantic Wikipedia*

Aufgrund ihrer Positionierung in der Architektur wird nachfolgend dieser, dem RDF-DBMS vorgelagerte, Zwischenspeicher abkürzend mit *RDF-ZSE (RDF-Zwischenspeicher Einheit)* bezeichnet.

4.4.2 Zu speichernde Informationen

Als wichtigste Frage nach der Verortung der RDF-ZSE in der Architektur stellt sich jene nach den Informationen, die überhaupt in den Zwischenspeicher aufgenommen werden sollen und auf welcher Informationsebene (Anfragen, Anfragenteile, RDF-Tripel, IRIn und Literale) diese liegen. Dabei ist es hilfreich, die im Falle von RDF möglichen Typen von Information zu betrachten. Dies sind:

- Konkret definierte Information
- Über logische Inferenz gewonnene Information

Im Falle der reinen Beantwortung von Anfragen ist die Unterscheidung der Informationstypen noch ohne Bedeutung, sie wird erst bei der Betrachtung von Aktualisierungen wichtig.

Anfragebeantwortung Unter Rückgriff auf Abschnitt 3.3 und 4.2.3 stellt man weiter fest, dass es das Ziel sein muss, möglichst für jede Anfrage die Lösung der entsprechenden SPARQL-Anfrage (aufgrund des konkretisierten Szenarios können semantische Anfragen mit SPARQL-Anfragen gleichgesetzt werden) in der RDF-ZSE vorzuhalten. Insbesondere für Anfragen mit hoher Auftrittshäufigkeit $\#(s_i)$ ist dies wünschenswert.

Um die Lösung für Aufrufe derselben Anfragen auffindbar zu machen, müssen diese in eine eindeutig definierte, algorithmisch gut handhabbare Form transformiert werden. Dies kann durch eine effizient berechenbare, injektive Abbildung geschehen, die jedem Element einen Zahlenwert zuordnet (beispielsweise

eine *Hash*-Funktion (Kollisionen können durch die Textrepräsentation der Anfrage aufgelöst werden).

Die Entscheidung, ob sich die Lösung einer Anfrage im Zwischenspeicher befindet, kann somit effizient ausgeführt werden. In der Regel wird die Zeit zum Abruf in $O(\log n)$ liegen.

Eine mögliche Verbesserung der Trefferrate durch Anfragezerlegung respektive der Kombination vorhandener Ergebnisse wird in Abschnitt 8.1 diskutiert. Für den hiesigen Entwurf der Lösung wird angenommen, dass das Zwischenspeichern der Lösungen auf der Ebene von SPARQL-Anfragen und ihren Lösungen geschieht.

4.4.3 Behandlung von Änderungen

Das hier betrachtete Szenario (siehe Abschitt 3.1) gibt außerdem vor, dass es Änderungen an den semantischen Informationen gibt. Dies impliziert die Frage nach einer Methode, wie für diejenigen Anfragen, welche von den Änderungen betroffen sind, die Einträge im Zwischenspeicher geeignet invalidiert werden können. Dieselbe Information kann dann auch dazu genutzt werden, die Zwischenspeicher der konventionellen *Wikipedia*-Architektur zu invalidieren.

Während es für die Identifikation des zu einer Anfrage gehörenden Resultates notwendig war, ist es im Falle von Änderungen nötig, alle potentiell betroffenen Anfragen (und Seiten) auszumachen. Daher ist es nötig, eine umgekehrte Abbildung zu definieren, welche ausgehend von einem RDF-Tripel, d.h. der Repräsentation einer RDF-Aussage, wie es bei der Speicherung semantischer Informationen und als elementarer Teil einer SPARQL-Anfrage auftritt, diejenigen Anfragen liefert, die von einer Änderung betroffen sind. Speichert man außerdem noch die Seiten-Identifikatoren, so können sämtliche Zwischenspeicher korrekt invalidiert werden.

Um jedoch alle Anfragen zu erhalten, deren Einträge invalidiert werden müssen, ist es notwendig, auch diejenigen RDF-Tripel zu betrachten, die erst über Inferenz entstehen. Das dabei entstehende Problem, wie viel der Zwischenspeicher über die Ontologiesprache wissen muss, wird in Abschnitt 5 diskutiert. Nimmt man für den Moment an, man hätte diese Tripel bereits gegeben, dann kann bei der Änderung eines RDF-Tripels auf korrespondierende Einträge in der Liste der umgekehrten Abhängigkeiten geprüft werden. Bei entsprechender Organisation ist die Invalidierung ebenfalls effizient durchgeführt werden.

Zugriff auf Schema-Abhängigkeiten Da diese Erzeugung erst durch logische Inferenz enstehender Informationen für jedes geänderte RDF-Tripel zu prüfen ist, sie aber einen Zugriff auf das RDF-DBMS erfordert, ist es wichtig, auch die Abhängigkeiten zwischen Schema-Elementen zwischenzuspeichern, um die Vorteile eines Zwischenspeichers voll ausschöpfen zu können. Detaillierte Überlegungen in dieser Hinsicht werden in Abschnitt 5.4 dargelegt.

4.4.4 Verdrängungsstrategie

Zwischenspeicher basieren üblicherweise auf schnellen, aber begrenzten Speichereinheiten[48]. Daher muss eine Strategie entwickelt werden, welche Elemente aus

[48] Ausnahme sind die in Abschnitt 2.1.4 beschriebenen *Webcaches*, welche über vergleichsweise große Speicher verfügen

dem Zwischenspeicher entfernt werden sollen, um Platz für neue zu schaffen, wenn die Speichergrenze erreicht ist.

Strategie *LFU (least frequently used)* Auf Basis der Überlegungen in Abschnitt 3.1.2 kann man die Wichtigkeit einer Seite des Portals für den Benutzer und damit einer (semantischen) Anfrage für den Benutzer durch die Anzahl ihrer Aufrufe im Zeitintervall t, d.h. die Größe der Äquivalenzklasse von Anfragen, die von der Seite mittels der Funktion g aufgespannt wird, definieren. Dies legt zunächst die Strategie nahe, das am wenigsten verwendete Anfrage (mitsamt ihrer Lösung) aus dem Zwischenspeicher zu entfernen (*LFU (least frequently used)*). Dazu wäre es nötig, die Häufigkeit der Verwendung in der Liste der im Zwischenspeicher verfügbaren Anfragen zu protokollieren.

Diese Lösung hat jedoch den Nachteil, dass Anfragen, deren Nutzung innerhalb eines Intervalls nur langsam ansteigt, welche aber später intensiv verwendet werden, sich eher gegenseitig verdrängen. Eine Anfrage, die innerhalb eines Intervalls sehr häufig, dann aber längere Zeit nicht mehr aufgerufen wird, bleibt mit hoher Wahrscheinlichkeit im Zwischenspeicher, obwohl ihr Nutzen nur noch beschränkt ist.

Strategie *LRU (least recently used)* Eine hohe Anzahl von Aufrufen bedeutet jedoch gleichzeitig eine hohe Wahrscheinlichkeit, dass der letzte Ausführungszeitpunkt einer Anfrage erst kurz zurückliegt. Dies wiederum legt es nahe, im Zweifelsfall die Anfrage (mitsamt ihrer Lösung) aus dem Zwischenspeicher zu entfernen, deren letzter Ausführungszeitpunkt am längsten zurückliegt (*LRU (least recently used)*). Um die Strategie zu verwenden, ist ein Mitprotokollieren des Zeitpunktes der letzten Ausführung in der Liste der Anfragen nötig.

Diese Strategie bietet zudem den Vorteil, dass die im Falle *LFU* auftreten, hier nicht zum Tragen kommen. Sie ist daher für das gegebene Szenario zu bevorzugen.

4.5 Zusammenfassung der Ergebnisse

An dieser Stelle soll eine kurze Zusammenfassung der Resultate und Beobachtungen dieses Kapitels gegeben werden. Eine genauere Darstellung der Vorgänge im Zwischenspeicher findet in Kapitel 6 statt, dort wird auch ein Algorithmus in Pseudo-Code als Resultat der Überlegungen formuliert. Eine Leistungsanalyse erfolgt in Kapitel 7 auf theoretischer Ebene.

4.5.1 Resultate

In Abschnitt 4.1 wurde nachgewiesen, dass jede SMW-QL-Anfrage in eine korrespondierende SPARQL-Anfrage übersetzt werden kann. Auf diesem Wege wurde die Möglichkeit geschaffen, eine Lösung zu entwerfen, welche über das konkrete Szenario hinaus auch in anderen partizipativen semantischen Portalen eingesetzt werden kann (Voraussetzung ist lediglich, dass das zugrundeliegende RDF-DBMS in der Lage ist, SPARQL-Anfragen zu beantworten).

Abschnitt 4.2 untersuchte die verschiedenen Möglichkeiten zur Optimierung, die in Abschnitt 2.1 vorgestellt wurden, im Hinblick auf das konkrete Szenario (siehe Abschnitt 3.1. Fazit der Untersuchung war, dass hier ein Zwischenspeicher am besten zur Verbesserung der Leistung geeignet ist. In Abschnitt 4.3

wurden die Anforderungen an diesen Zwischenspeicher formuliert, in Abschnitt 4.4 die Entwurfsentscheidungen, die zur Umsetzung dieser Anforderungen getroffen wurden, dokumentiert.

Es wurde dabei die Entscheidung für eine dem RDF-DBMS vorgelagerte Zwischenspeicher-Einheit getroffen, welche auf der Ebene vollständiger Anfragen arbeitet. Als Verdrängungsstrategie wurde *Least Recently Used (LRU)* gewählt; die effiziente Auffindung vorhandener Anfrageresultate kann mittels Hashtabellen realisiert werden. Zur Invalidierung von Ergebnissen bei Änderungen werden zusätzlich zu den Anfragen auch umgekehrte Abhängigkeiten gespeichert, welche zu elementaren Tripel-Suchmustern die Einträge der potentiell betroffenen Anfragen im Zwischenspeicher liefern.

Die spezielle Problematik der korrekten Invalidierung bei der Verwendung logischer Inferenz wurde in Abschnitt 5 diskutiert. Es stellte sich dabei heraus, dass keine vollständige Unabhängigkeit von den verwendeten Ontologiesprachen erreicht werden kann. Es ist jedoch möglich, unter der Verwendung der Unterscheidung zwischen Typ und Instanz sowie des RDF-Eigenschafts- und des RDF-Klassen-Elements einen Algorithmus anzugeben, der die ontologischen Abhängigkeiten aus einem RDF-Graphen extrahiert. Für Ontologiesprachen, welche die Spezifikation negativer Aussagen über die Beziehungen von Klassenelementen erlauben (beispielsweise Disjunktheit), muss der Algorithmus zusätzlich auf eine Liste zugreifen, welche diese Beziehungen bei der Erzeugung der Abhängigkeiten ausschließt.

Weiterhin wurde dabei festgehalten, dass die resultierenden Abhängigkeiten, bei Speicherung in einer Hashtabelle, auch dazu genutzt werden können, um zu entscheiden, ob eine Änderung eine Neuausführung zur Erfassung der ontologischen Abhängigkeiten erforderlich macht.

Zum Abschluss von Abschnitt 5 wurde ein Verfahren skizziert, durch das mit Hilfe von SPARQL-Anfragen die Abhängigkeiten im konkreten Fall aus dem RDF-DBMS extrahiert werden können.

5 Intelligente Invalidierung des Zwischenspeichers

Der Abschnitt 4.4.2 führt aus, dass die Kenntnis jener RDF-Tripel, die erst durch logische Inferenz entstehen, für eine korrekte Invalidierung aller von einer Änderungen betroffenen Zwischenspeicher-Einträge notwendig ist. Das für logische Inferenz notwendige Wissen über die Beziehungen von Klassen und Eigenschaften wird im Falle von RDF in der *Ontologie* spezifiziert, welche die eigentliche Semantik des Dokumentes definiert. Diese wird auch als *Schema* bezeichnet. Zur Beschreibung eines solchen Schemas ist eine *Ontologiesprache* notwendig. Im Falle von RDF wurden die maßgeblichen Ontologiesprachen durch RDF-Schema (*RDFS*, siehe [Hay04]) und die *Ontology Web Language* (*OWL*, siehe [PSHH04]) definiert.

Keine Unterstützung logischer Inferenz durch das RDF-DBMS Unterstützt das RDF-DBMS keine Inferenz[49], sind neben den Überlegungen aus Abschnitt 4.4.2 keine weiteren nötig - es existieren keine RDF-Tripel, die erst durch logische Inferenz entstehen. In diesem Fall ist auch das Zwischenspeicher der Abhängigkeiten zwischen Schema-Elementen hinfällig; finden Änderungen am Schema statt, so werden nur deren direkt Auswirkungen auf Einträge im Zwischenspeicher untersucht (also beispielsweise das Löschen einer Klasse, für welche es Instanzen gibt, die Teil eines zwischengespeicherten Anfrageresultats sind).

Unterstützung logischer Inferenz durch das RDF-DBMS Nimmt man jedoch ein RDF-DBMS an, welches Unterstützung für logische Inferenz implementiert, so stellt sich die Frage, ob und in welchem Maße Wissen über die Ontologiesprache in die Logik des Zwischenspeichers integriert werden muss respektive wie stark von den konkreten Ontologiesprachen abstrahiert werden kann.

Dabei können im wesentlichen zwei Fälle unterschieden werden:

1. Die RDF-ZSE kennt das Vokabular und die Semantik der verwendeten Ontologiesprachen
2. Die RDF-ZSE besitzt keine Kenntnisse über die verwendeten Ontologiesprachen

Es sollen nun Vor- und Nachteile der beiden Möglichkeiten im Einzelnen betrachtet werden.

5.1 Ontologiesprache vollständig bekannt

Für den Fall, dass Vokabular und Semantik dem Zwischenspeicher für eine Reihe von Ontologiesprachen vollständig bekannt sein soll, muss für jede Ontologiesprache eine interne Nachbildung der Semantik erfolgen. Die entstehenden Algorithmen und Datenstrukturen müssen es erlauben, eine Rekonstruktion aller Beziehungen und Abhängigkeiten durchzuführen.

[49] Die Umsetzung von Algorithmen zur Unterstützung logischer Inferenz befindet sich in den meisten verfügbaren RDF-DBMSen in einem frühen Stadium oder ist noch nicht vorhanden

Vorteile Für dem Zwischenspeicher bekannte Ontologiesprachen können die Abhängigkeiten intern rekonstruiert und für die korrekte Invalidierung verwendet werden.

Nachteile Es geht Flexibilität geht verloren, da nur RDF-DBMSe mit Ontologiesprachen angebunden werden können, welche von der RDF-ZSE unterstützt werden. Diese Unterstützung bedeutet im Wesentlichen die Nachbildung der Funktionalität, die bereits im RDF-DBMS vorhanden ist. Unterstützt ein RDF-DMBS mehrere Ontologiesprachen, muss zudem zur Laufzeit bekannt sein, welche für die Spezifikation der Semantik im aktuellen Dokument eingesetzt wird.

5.2 Ontologiesprache vollständig unbekannt

Die Betrachtungen des ersten Falls zeigt, dass bei der vordefinierten Integration von Wissen über bestimme Ontologiesprachen die Flexibilität verloren geht, mit unbekannten Ontologiesprachen umzugehen.

Naive Lösung (keine weiteren Annahmen) Im anderen Fall, nämlich jenem, in dem die RDF-ZSE kein Wissen über die Ontologiesprache besitzen soll, ist zunächst eine naive Lösung denkbar, welche alle Abhängigkeiten direkt aus dem RDF-Graphen entnimmt. Diese Lösung hält einer kritischen Beurteilung jedoch nicht stand:

Vorteile Es handelt sich theoretisch um die flexibelste Lösung, da keinerlei Annahmen über die verwendete Ontologiesprache gemacht werden.

Nachteile Ohne jegliche zusätzliche Annahmen kann die RDF-ZSE nicht entscheiden, welche Abhängigkeiten zwischen Elementen des RDF-Graphen tatsächlich relevant sind, um über die Notwendig von Invalidierungen entscheiden zu können. Daher muss zur Bestimmung der Abhängigkeiten der komplette RDF-Graph in der RDF-ZSE repliziert werden. Dieser müsste zudem bei jeder Änderung neu als dem RDF-DBMS geladen werden, da Änderungen am Schema nicht von Änderungen an Instanzen unterschieden werden können (es kann also das Zwischenspeichern entfallen). Die Lösung ist daher aufgrund des zusätzlich anfallenden Speicherplatz- und Zeitbedarfes praktisch nicht einsatzfähig.

5.3 Minimale zusätzliche Annahmen: Typ, Instanz und das Eigenschaftselement

Da der naive Ansatz praktisch nicht einsetzbar ist, stellt sich daher die Frage, welche zusätzlichen Annahmen (oder welches zusätzliche Wissen) nötig sind, um eine korrekte Invalidierung im Falle der Verwendung logischer Inferenz zu gewährleisten. Wie die folgenden Überlegungen zeigen, ist eine hinreichend effiziente Implementierung ohne Einschränkung der Flexibilität, d.h. ohne weiteres Wissen über konkrete Ontologiesprachen möglich, wenn zwischen Typen (*rdf:type*) und Instanzen unterschieden wird sowie das Eigenschafts-Element (*rdf:Property*) und das Eigenschafts-Element (*rdf:Property*) explizit ausgenutzt wird.

Negative Implikationen für das Leistungsverhalten entstehen allerdings dann, wenn die Ontologiesprache auch negative Aussagen über die Beziehung zweier Klassen zu machen. So erlaubt OWL beispielsweise mit „*owl:disjointWith*“ auszudrücken, dass die Menge der Instanzen zweier Klassen stets disjunkt ist. Diese Problematik wird am Ende dieses Abschnitts diskutiert.

5.3.1 Motivation der Entscheidung

Anhand des Beispiels in Abbildung 13 soll nachfolgend verdeutlicht werden, wie die Entscheidung, lediglich zwischen Typen und Instanzen zu unterscheiden, motiviert ist, und warum sie ausreicht, um die Abhängigkeiten der Schema-Elemente zu analysieren, die für eine korrekte Invalidierung aller in der RDF-ZSE vorgehaltenen Anfrageresultate nötig sind. Sie orientiert sich dabei an den Beispieldaten aus dem Abschnitt 2.2.2 (siehe Abbildung 3). Zudem werden einige Elemente aus der *vCard*-Definition in RDF/XML ([Ian01], der *FOAF (Friend of a Friend)*-Definition sowie der RDF-Definition ([W3C06]) verwendet[50]. Die Typ-Beziehung jeder Klasse zum Klassen-Element (*rdfs:Class*), welche in der Regel nur implizit angegeben ist, wurde hier zusätzlich durch gestrichelte Kanten und Knoten mit kursiv gedruckten Beschreibungen eintragen.

5.3.2 Behandlung von Klassen und ihren Instanzen

Dieses Beispiel demonstriert verschiedene Arten semantischer Beziehungen. So stellt „Person“ (*foaf:Person*) eine Klasse dar, deren Instanz „Matt Jones“ (bzw. die IRI „`http://somewhere/MattJones/`“) ist. Über logische Inferenz folgt also automatisch, dass „Matt Jones“ auch Instanz der Klasse „Agent“ (*foaf:Agent*) ist. Das bedeutet aber gerade, dass hier eine Abhängigkeit existiert, die für die RDF-ZSE relevant ist. Wird eine neue Instanz der Klasse „Person“ eingefügt, so müssen nicht nur alle Anfrage-Resultate, die Instanzen von PPerson“ enthalten, invalidiert werden, sondern auch jene, die Instanzen von „Agent“ enthalten.

Es fällt auf, dass solche Abhängigkeiten genau dann vorhanden sind, wenn im Graphen oberhalb (in Richtung des Wurzelelementes) einer Typ-Kante noch weitere Beziehungen bestehen. Wie durch die Kennzeichnung der meist implizierten Instantiierung des Klassen-Elementes deutlich wird, sind dies gerade Abhängigkeiten zwischen den Klassen im Graphen.

5.3.3 Behandlung von Eigenschafts-Elementen

Hingegen fasst die Eigenschaft „*vc:N*“ alle Eigenschaften, welche den Namen betreffen (bis auf die unabhängig zu ändernde Angabe des anzuzeigenden Namen (*vc:FN*)) über die Eigenschaft-Untereigenschaft-Beziehung (*rdfs:subPropertyOf*) zusammen; hier werden allerdings nur der Familienname (*vc:Family*) und der Vorname (*vc:Given*) dargestellt. Alle Änderungen an den Untereigenschaften betreffen dabei automatisch auch die übergeordnete Eigenschaften.

Hier stellt man fest, dass solche Abhängigkeiten genau dann vorhanden sind, wenn im Graphen unterhalb (vom Wurzelelement aus betrachtet) einer Typ-Kante zum Eigenschafts-Element (*rdf:Property*) noch weitere Beziehungen bestehen, d.h. Beziehungen zwischen Instanzen des Eigenschafts-Elements, d.h.

[50] Gegenüber der exakten Definition des *vCard*-Formates wurde eine sinnerhaltende Änderung vorgenommen, um Eigenschaft-Untereigenschaft-Beziehungen (*rdfs:subPropertyOf*) in das Beispiel aufzunehmen

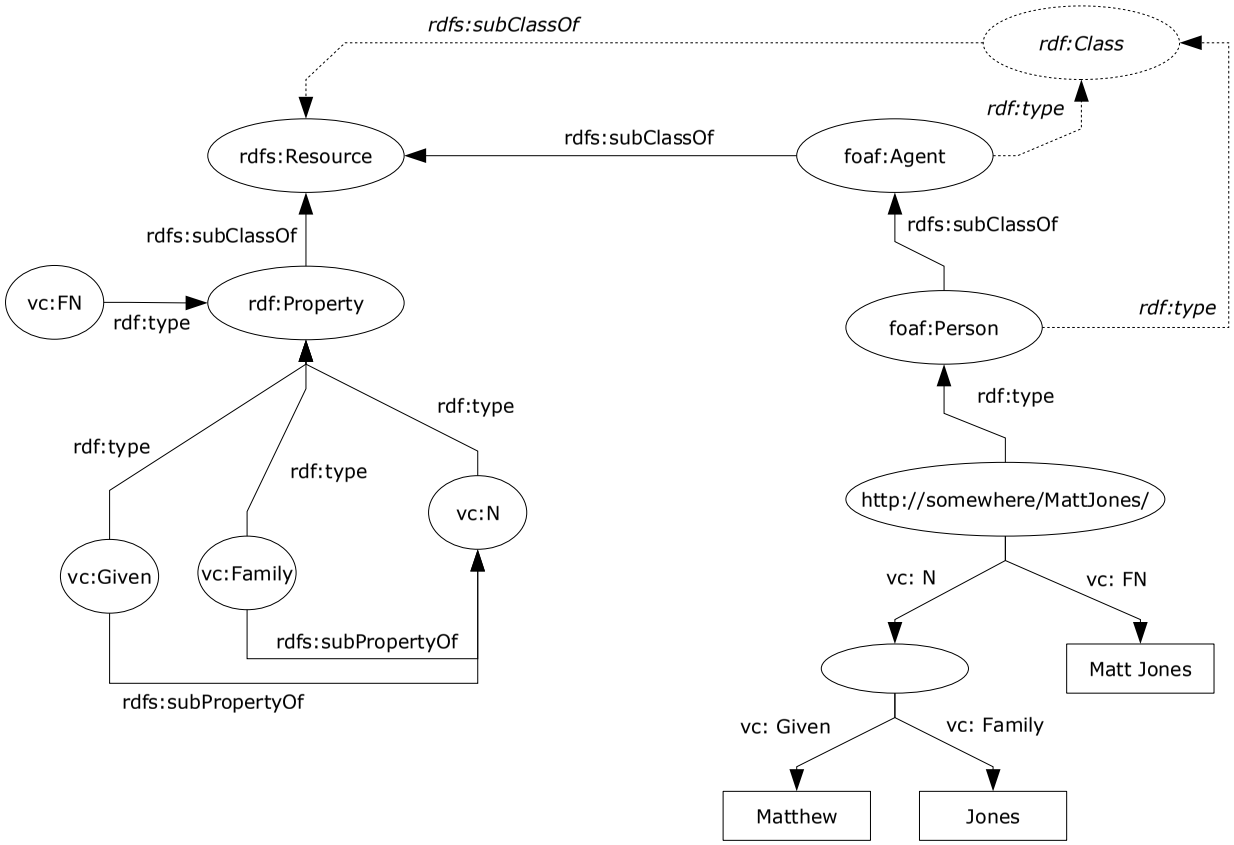

Abbildung 13: Beispiel eines RDF-Graphen mit semantischen Abhängigkeiten

Eigenschaften, bestehen. Im hiesigen Fall sind die verwendeten Eigenschaften direkte Instanzen des Eigenschafts-Elements. Dies ist jedoch nicht immer der Fall. So definiert etwa OWL verschiedene Spezialisierungen des Eigenschafts-Elements (z.B. Funktionale Eigenschaften oder Objekt-Eigenschaften). Unterstützt das RDF-DBMS jedoch Inferenz, was hier gerade Voraussetzung ist, liefert eine Anfrage nach den Instanzen des Eigenschafts-Elements auch die entsprechenden Instanzen seiner Spezialisierungen.

Für die Invalidierung von Resultaten sind Abhängigkeiten zwischen Eigenschaften dann von Belang, wenn in einer Anfrage (u.a.) nach allen Ressourcen gefragt wird, welche eine bestimmte Eigenschaft A aufweisen. Wird dann für eine weitere Instanz der Wert einer Eigenschaft B, von welcher die Eigenschaft A abhängt initial eingefügt, so muss für eine korrekte Antwort auf die Anfrage die entsprechende Instanz in das Resultat aufgenommen werden. Der Eintrag in der RDF-ZSE ist also ungültig geworden.

5.3.4 Probleme im Falle der Spezifikation negativer Beziehungen

Wie bereits in den Anfangsüberlegungen zum minimal notwendigen Wissen über die verwendeten Ontologiesprachen dargelegt, führt es zu Problemen bei der Invalidierung von Einträngen in der RDF-ZSE, wenn die Ontologiesprache es erlaubt, negative Aussagen über die Beziehungen zweier Klassen zu machen. So erlaubt es OWL beispielsweise, zwei Klassen als disjunkt zu spezifizieren. In diesem Fall werden unter Umständen inkorrekterweise Einträge invalidiert, deren Gültigkeit von einer Änderung gar nicht betroffen ist. Zur Illustration dieses Problems wird das vorhergehende Beispiel aus Abbildung 13 um weitere Elemente aus dem FOAF-Vokabular und eine entsprechende Instanz erweitert. Dies wird in Abbildung 14 dargestellt.

Wie bei Betrachtung der Klassen „Person“ (*foaf:Person* und „Dokument“ (*foaf:Document*) erkennbar wird, sind diese über als disjunkt (*owl:disjointWith*) markiert. Es kann also niemals eine Instanz der einen gleichzeitig eine Instanz der anderen Klasse sein (auch nicht über logische Inferenz).

Dies bedeutet aber, dass das Resultat einer Anfrage, welche nach Instanzen der Klasse „Person“ sucht, inkorrekterweise invalidiert wird, wenn eine neue Instanz der Klasse „Dokument“ eingefügt wird, da die RDF-ZSE die Semantik der Disjunkt-Beziehung nicht kennt. Der Effekt tritt umso stärker auf, je komplexer die weiteren Beziehungen der Klassen sind; dadurch kann die Trefferrate in der RDF-ZSE deutlich absinken.

5.3.5 Gegenmaßnahmen

Um diesem Problem entgegenzuwirken, muss zusätzliches Wissen über mögliche Ontologiesprachen in die RDF-ZSE aufgenommen werden. Dies schränkt die Flexibilität offensichtlich ein. Jedoch ist es dabei lediglich vonnöten, die Elemente zu spezifizieren, die bei der Auswertung der Abhängigkeiten ignoriert werden sollen, d.h. jene, die dazu dienen, zwei Elemente als einander ausschließend zu markieren (für OWL sind dies die eingangs erwähnten Eigenschaften *owl:disjointWith* und *owl:complementOf* sowie die Eigenschaft *differentFrom*). Detailliertes Wissen über ihre Semantik muss nicht in die RDF-ZSE integriert werden.

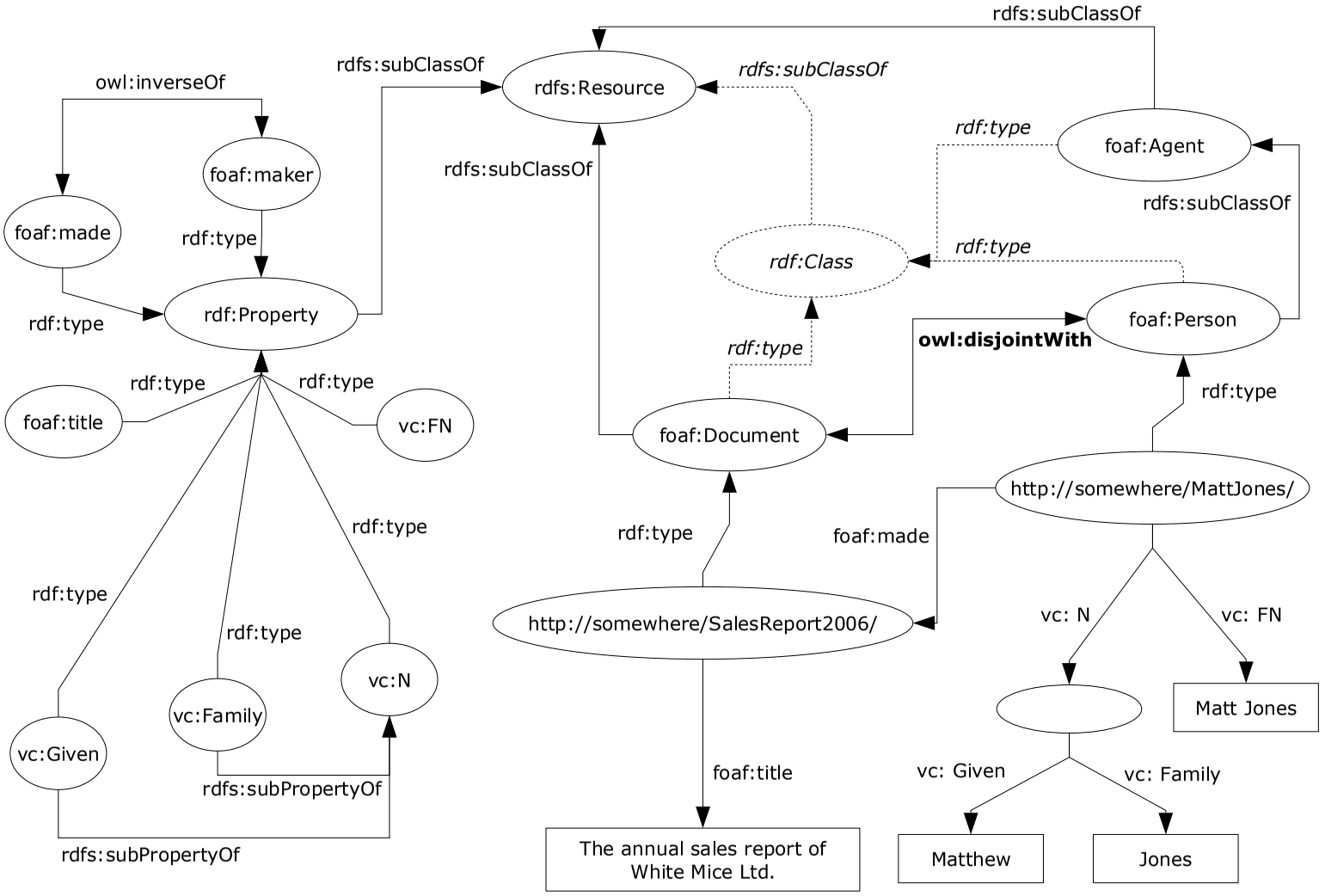

Abbildung 14: Beispiel eines RDF-Graphen, in welchem Informationen als disjunkt spezifiziert werden

5.4 Zwischenspeichern von Schema-Abhängigkeiten

Die Betrachtungen Abschnitts 5.3 und speziell die Beispiele in den Abbildungen 13 und 14 zeigen, dass es sich bei den Abhängigkeiten zwischen Schema-Elementen zunächst einmal um einen Teilgraphen des RDF-Graphen handelt. Für die Verwendung zur Invalidierung in der RDF-ZSE sind eventuell einige Abhängigkeiten zu ignorieren (siehe Abschnitt 5.3.4).

Die nachfolgende Abbildung 15 isoliert die Schema-Abhängigkeiten aus Abbildung 14 zur genaueren Betrachtung. Die Beschriftungen der Kanten wurde dabei entfernt, da die genaue Semantik der Beziehungen zwischen den Schema-Elementen für die RDF-ZSE unerheblich ist. Lediglich die Kante, welche die Disjunkt-Beziehung zwischen der Klasse „Person“ (*foaf:Person* und „Dokument“ (*foaf:Document*) ausdrückt, wird entsprechend Abschnitt 5.3.4 gesondert gekennzeichnet.

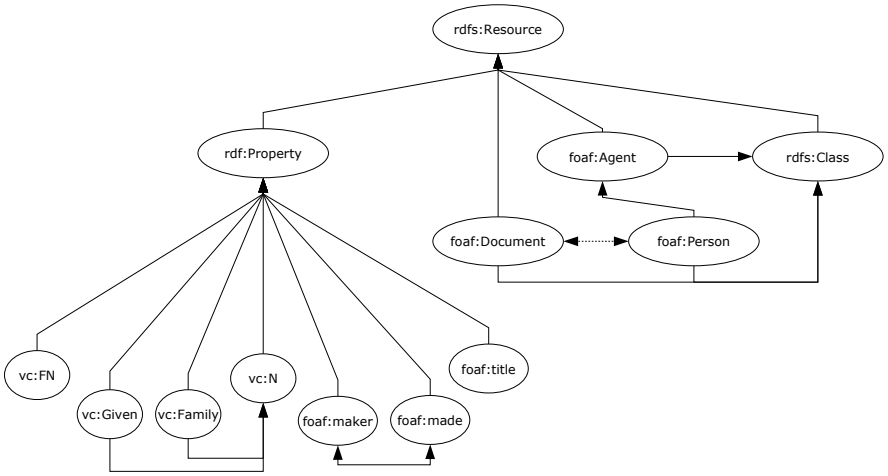

Abbildung 15: Darstellung der Schema-Abhängigkeiten des Beispiels

Wie erkennbar ist, handelt es sich bei dem Teilgraphen, welcher die Schema-Abhängigkeiten darstellt, im Wesentlichen um einen Baum. Störend sind jedoch die bidirektionalen Verbindungen, wie etwa jene zwischen den Klassen „Dokument“ (*foaf:Document*) und „Person“ (*foaf:Person*) oder den Eigenschaften „hat erstellt“ (*foaf:made*) und „Ersteller“ (*foaf:maker*) sowie zweierlei Wege in Richtung der Wurzel wie etwa zwischen „Vorname“ (*foaf:Given*) und „Name“ (*foaf:N*).

5.4.1 Abhängigkeitsbaum

Mittels folgender Transformationsregeln lässt sich dieser Graph vollständig in einen *Abhängigkeitsbaum* überführen, welcher nur noch die Abhängigkeiten enthält, die für eine korrekte Invalidierung von Anfrageresultaten in der RDF-ZSE notwendig sind:

1. Entferne alle direkten Zyklen[51]

[51] Dieser Fall wurde im Beispiel nicht explizit aufgeführt, der Schritt ist allerdings not-

2. Entferne Disjunkt- und Komplement-Beziehungen (siehe Abschnitt 5.3.4)
3. Für jede bidirektionale Kante:
   (a) Dupliziere die Knoten, zwischen den sie besteht
   (b) Entferne die Kante
   (c) Füge zwei unidirektionale (gerichtete) Kanten ein, welche jeweils einen der Ausgangsknoten mit dem entsprechend anderen duplizierten Knoten verbindet
4. Entferne alle Abhängigkeiten, die nicht Kinde-Eemente von Eigenschafts- oder Klassen-Element sind
5. Entferne alle Elemente, die Instanzen des Eigenschafts-Elementes repräsentieren und keine weiteren Abhängigkeiten haben
6. Entferne alle Elemente, die Instanzen des Klassen-Elementes repräsentieren und keine weiteren Abhängigkeiten haben

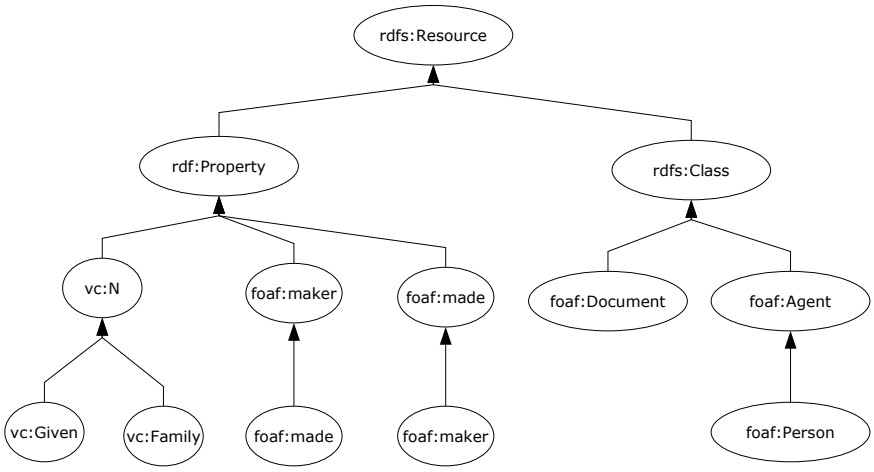

Abbildung 16: Resultat des Algorithmus zur Generierung des Abhängigkeitsbaums

Das Resultat nach Anwendung des Algorithmus im Falle des Beispiel aus Abbildung 15 wird in Abbildung 16 dargestellt. Die Speicherung des resultierenden Abhängigkeitsbaum kann mit gängigen Techniken (z.B. Speicherung eines Baums in einer Halde *(heap)*) erfolgen. Um den Zugriff auf die Abhängigkeiten auf Kosten zu beschleunigen, können die Abhängigkeiten ausmaterialisiert, d.h. mehrfache Abhängigkeiten in direkte überführt, und in einer Tabelle, deren Schlüssel über Hashwerte gebildet wird, gespeichert werden.

Da sich die Resultate des Algorithmus nur dann ändern, wenn Schema-Elemente gelöscht oder neu eingefügt werden, muss zudem effizient geprüft werden können, ob eine Änderung das Schema betrifft. Wird die o.g. Speicherung

wendig, da beispielsweise laut Spezifikation das RDFS-Klassenelement (*rdfs:Class*) mit seiner Typangabe auf sich selbst verweist

in einer Tabelle durchgeführt, so kann diese Prüfung durch den Vergleich mit den Schlüsseleinträgen geschehen. Nur wenn diese positiv ausfällt, d.h. wenn das neue Element eine Klasse ist oder es mit einer anderen als der Typ-Beziehung mit einem der vorhandenen Elemente verknüpft ist, muss der Algorithmus anschließend erneut ausgeführt werden.

5.4.2 Strategie zur Gewinnung der Abhängigkeiten

Zum Abschluss dieses Abschnittes soll nun noch kurz darauf eingegangen werden, wie die Abhängigkeiten zwischen Schema-Elementen konkret gewonnen werden können. Hierzu kann auf die RDF-Anfragesprache SPARQL (siehe Abschnitt 2.2.2, [P05]) zurückgegriffen werden, die auch für die eigentlichen Anfragen von *Semantic MediaWiki* an das RDF-DBMS eingesetzt wird (siehe Abschnitt 4.1).

6 Algorithmische Beschreibung des Zwischenspeichers

Dieses Kapitel konkretisiert die in Kapitel 4 und 5 vorgestellten Überlegungen. Es werden zunächst die Abläufe in der RDF-ZSE während der drei elementaren semantischen Operationen „Suchen“, „Einfügen“ und „Löschen“ untersucht.

Das zusammenfassende Resultat dieser Analyse ist ein Algorithmus in Pseudo-Code, welcher das gestellte Problem der Anfrageoptimierung für die Klasse der RDF-DBMSe, welche RDFS-kompatible Ontologiesprachen einsetzen, löst. Eine Leistungsanalyse dieses Algorithmus findet in Kapitel 7 statt, Verbesserungsmöglichkeiten werden in Kapitel 8 skizziert.

6.1 Analyse elementarer semantischer Operationen

Im folgenden soll anhand von Sequenzdiagrammen erläutert werden, welche Schritte im Einzelnen bei der Bearbeitung einer Such-, einer Einfüge- und einer Lösch-Operation notwendig sind. Dabei wird der Schwerpunkt auf die Vorgänge innerhalb der RDF-ZSE gelegt, die zu diesem Zweck logisch in zwei Teile geteilt wird:

- Das *RDF-ZSE Kernmodul* (nachfolgend kurz: *Kernmodul*), welches die in 4.4 beschriebenen Funktionen des Auffindens und Abrufens gespeicherter Anfrageresultate realisiert und gegebenenfalls nicht vorhandene Resultate in den Zwischenspeicher aufnimmt (und dabei eventuell vorhandene Einträge gemäß der Verdrängungsstrategie entfernt) sowie die Überwachung von Änderungen durchführt und gegebenenfalls in Folge Anfrageresultate im Zwischenspeicher invalidiert. Dieses Modul enthält zu diesem Zweck zwei Tabellen, wobei die eine Anfragen und ihre Resultate (nachfolgend kurz: *Anfragetabelle*) und die andere alle Tripel-Suchmuster enthält, die geprüft werden müssen, um zu entscheiden, ob eine Änderung die Invalidierung von Anfrageresultaten im Zwischenspeicher nötig macht (nachfolgend kurz: *Tripeltabelle*)

- Den *RDF-ZSE Abhängigkeitsgenerator* (nachfolgend kurz: *Generator*), welcher für die Erzeugung und Zwischenspeicherung der Abhängigkeiten im Schema, wie sie in Kapitel 5 beschrieben sind, zuständig ist. Dieser verfügt über eine Tabelle, die zu jedem Schema-Element alle von ihm abhängigen Elemente speichert (nachfolgend kurz: *Abhängigkeitstabelle*) - die Einträge der einzelnen Elemente werden über deren Internationalisierte Ressourcen Identifikatoren (IRIen) abgerufen.

Die Diagramme bietet dabei jeweils einen groben Überblick, Details werden jeweils im Text erläutert. Die übergeordneten Abläufe entsprechen dabei jenen aus den Prozessbeschreiben in Abschnitt 3.4, wobei die RDF-ZSE wie in Abschnitt 4.4.1 erläutert, zwischen *Semantic MediaWiki* und dem RDF-DBMS angesiedelt ist. Zu beachten ist außerdem, dass eine Fehlerbehandlung nur dann stattfindet, wenn externe Module (hier also nur das RDF-DBMS) beteiligt sind. Bei allen Operationen innerhalb der RDF-ZSE wird davon ausgegangen, dass sie erfolgreich sind, um die Übersichtlichkeit der Betrachtungen nicht durch Fehlerbehandlung zu beeinträchtigen.

6.1.1 Such-Operationen

Der grobe Ablauf einer semantischen Such-Operation ist in Abbildung 17 dargestellt. Auf die einzelnen Punkte soll nun im Detail eingegangen werden:

1. Geht eine Anfrage im Kernmodul ein, so wird zunächst ihr Hashwert gebildet.

2. Existiert zu dem Hashwert ein Resultat in der Anfragetabelle, wird dieses zurückgegeben und der Zeitpunkt der letzten Verwendung aktualisiert (existieren mehrere Einträge wegen einer Hashwert-Kollision, so wird die Anfrage selbst als Unterscheidungsmerkmal hinzugezogen).

3. Existiert kein Resultat im Zwischenspeicher, so wird die Anfrage an das RDF-DBMS zur Beantwortung weitergeleitet.

4. Es wird geprüft, ob für das Eintragen eines neuen Resultats ein alter Eintrag verdrängt werden muss.

5. Ist dies der Fall, so wird der älteste Eintrag entfernt, die Abhängigkeiten zu dieser Anfrage werden aus der Tripeltabelle gelöscht und es wird der Seitenindex zur Invalidierung der konventionellen Zwischenspeicher an *MediaWiki* weitergeleitet.

6. Das Resultat wird zusammen mit der Anfrage, ihrem Hashwert, dem aktuellen Zeitpunkt und dem Seitenindex der *Wiki*-Seite, aus der die Anfrage stammt, in die Anfragetabelle eingetragen.

7. Anschließend werden für alle Tripel-Suchmuster, die in der Anfrage enthalten sind, Abhängigkeiten in der Tripeltabelle eintragen (d.h. für jedes Tripelsuchmuster wird dieses selbst, die aktuelle Anfrage und ihr Hashwert als neuer Datensatz eingefügt)

8. Schließlich wird das Resultat an *Semantic MediaWiki* weitergeleitet

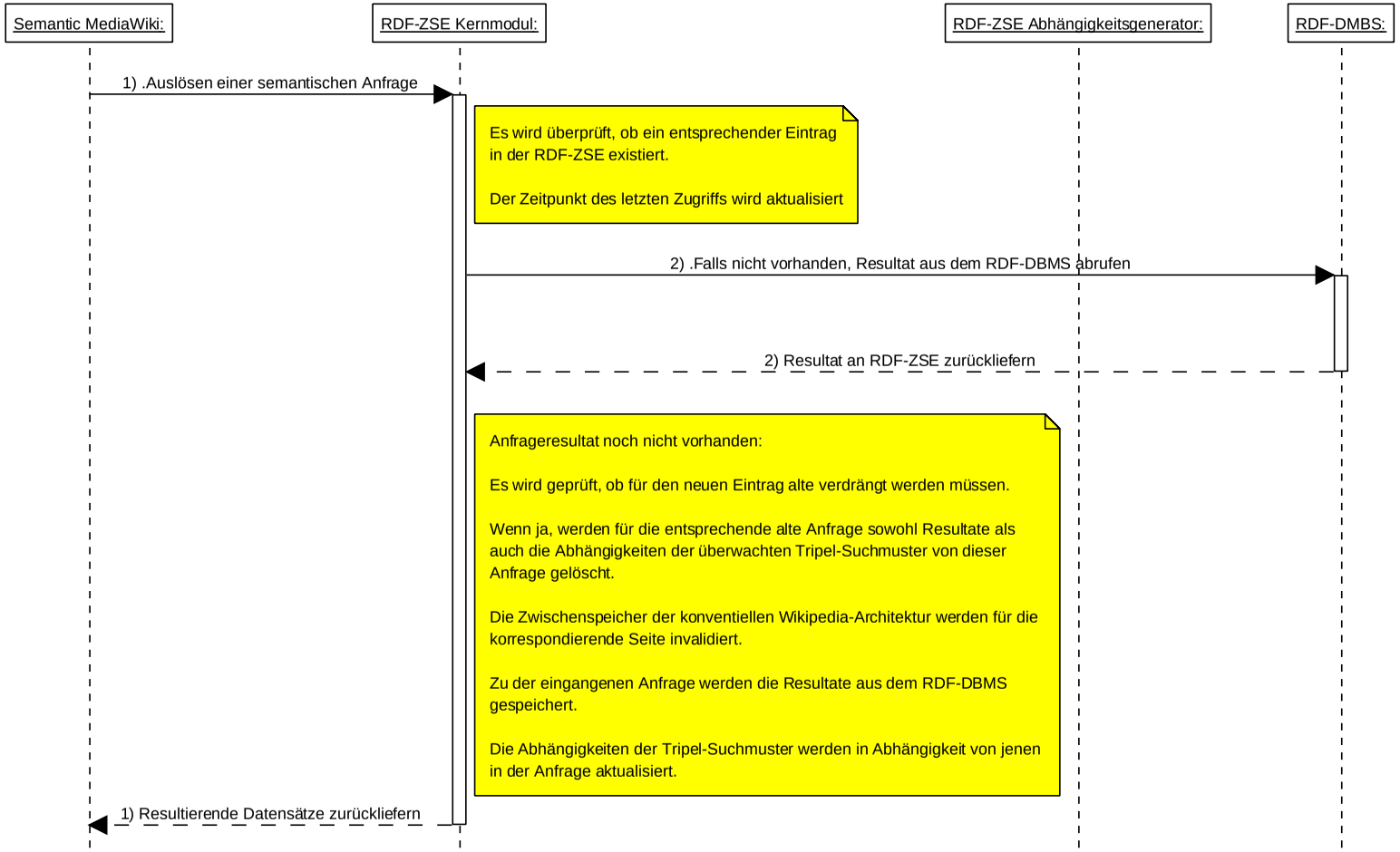

Abbildung 17: Vorgänge in der RDF-ZSE bei einer Such-Operation

6.1.2 Einfüge-Operationen

Der grobe Ablauf einer semantischen Einfüge-Operation ist in Abbildung 18 dargestellt. Auf die einzelnen Punkte soll nun im Detail eingegangen werden:

1. Geht eine Einfüge-Operation im Kernmodul ein, so wird diese zunächst an das RDF-DMBS weitergeleitet.
2. Ist das Einfügen im RDF-DBMS nicht erfolgreich, wird der Fehler an *Semantic MediaWiki* gemeldet (Abbruch der Bearbeitung)
3. Andernfalls wird eine Anfrage an den Generator gesendet, um zu der eingefügten RDF-Aussage alle logisch inferierten Aussagen zu erhalten:
   (a) Der Generator entnimmt der Abhängigkeitstabelle für jeden der drei Teile der RDF-Aussage die entsprechenden Abhängigkeiten
   (b) Mit diesen Abhängigkeiten erzeugt alle RDF-Aussagen, die durch logische Inferenz aus der ursprünglichen entstehen
   (c) Die Menge der RDF-Aussagen (nur die ursprüngliche, falls keine relevanten Abhängigkeiten bestehen) wird zurückgeliefert
4. Nun wird für jedes gespeicherte Tripelsuchmuster und jede erhaltene Aussage geprüft, ob die Aussage durch das Suchmuster erfasst wird.
5. Wenn ja, werden alle Anfragen und ihre Hashwerte, die von diesem Tripelsuchmuster abhängen aus der Tripeltabelle geladen und die entsprechenden Einträge invalidiert:
   (a) Die Abhängigkeiten zu dieser Anfrage werden aus der Tripeltabelle gelöscht.
   (b) Der entsprechende Eintrag wird in der Anfragetabelle entfernt (vorher wird der Seitenindex abgerufen).
   (c) Der Seitenindex wird zur Invalidierung der konventionellen Zwischenspeicher an *MediaWiki* weitergeleitet.
6. Anschließend wird eine Anfrage an den Generator gesendet, um zu entscheiden, ob das neu eingefügte Element das Schema verändert
7. Trifft dies zu, wird im Generator die Erzeugung einer neuen Abhängigkeitstabelle initiiert:
   (a) Der Generator ruft den vollständigen RDF-Graphen aus dem RDF-DBMS ab.
   (b) Er wendet den in Abschnitt 5.4 vorgestellten Algorithmus zu Isolierung der Schemaabhängigkeiten an.
   (c) Für jedes Element des Abhängigkeitsbaums werden anschließend in die Abhängigkeitstabelle die von ihm abhängigen Schema-Elemente eingetragen (anhand dieser Einträge kann später wieder die Prüfung erfolgen, ob ein Element ein Schema-Element ist).
8. Abhängig vom Erfolg der Erzeugung einer neuen Abhängigkeitstabelle wird ein positiver oder negativer Gesamtstatus an *Semantic MediaWiki* zurückgegeben.

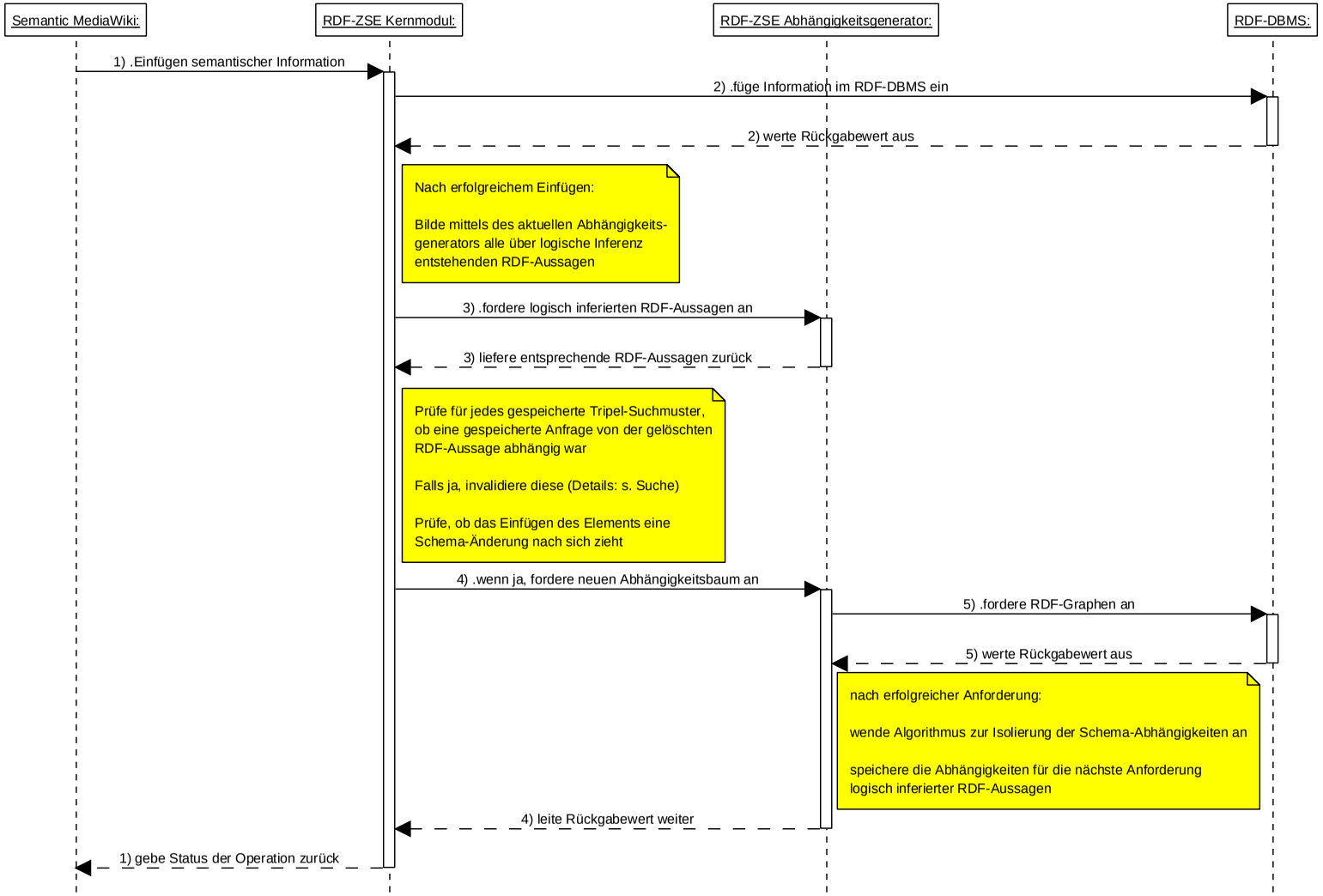

Abbildung 18: Vorgänge in der RDF-ZSE bei einer Einfüge-Operation

6.1.3 Lösch-Operationen

Der Ablauf einer semantischen Lösch-Operation (siehe Abbildung 19) ist im Wesentlichen identisch mit Ablauf einer Einfüge-Operation. Er wird daher hier vor allem der Vollständigkeit halber und um die Teilschritte der Invalidierung und Generierung der Abhängigkeitstabelle gekürzt angegeben:

1. Geht eine Lösch-Operation im Kernmodul ein, so wird diese zunächst an das RDF-DMBS weitergeleitet.
2. Ist das Löschen im RDF-DBMS nicht erfolgreich, wird der Fehler an *Semantic MediaWiki* gemeldet (Abbruch der Bearbeitung)
3. Andernfalls wird eine Anfrage an den Generator gesendet, um zu der gelöschten RDF-Aussage alle logisch inferierten Aussagen zu erhalten.
   ⋮
4. Nun wird für jedes gespeicherte Tripelsuchmuster und jede erhaltene Aussage geprüft, ob die Aussage durch das Suchmuster erfasst wird.
5. Wenn ja, werden alle Anfragen und ihre Hashwerte, die von diesem Tripelsuchmuster abhängen aus der Tripeltabelle geladen und die entsprechenden Einträge invalidiert.
   ⋮
6. Anschließend wird eine Anfrage an den Generator gesendet, um zu entscheiden, ob das gelöscht Element ein Schema-Element war
7. Trifft dies zu, wird im Generator die Erzeugung einer neuen Abhängigkeitstabelle initiiert.
   ⋮
8. Abhängig vom Erfolg der Erzeugung einer neuen Abhängigkeitstabelle wird ein positiver oder negativer Gesamtstatus an *Semantic MediaWiki* zurückgegeben.

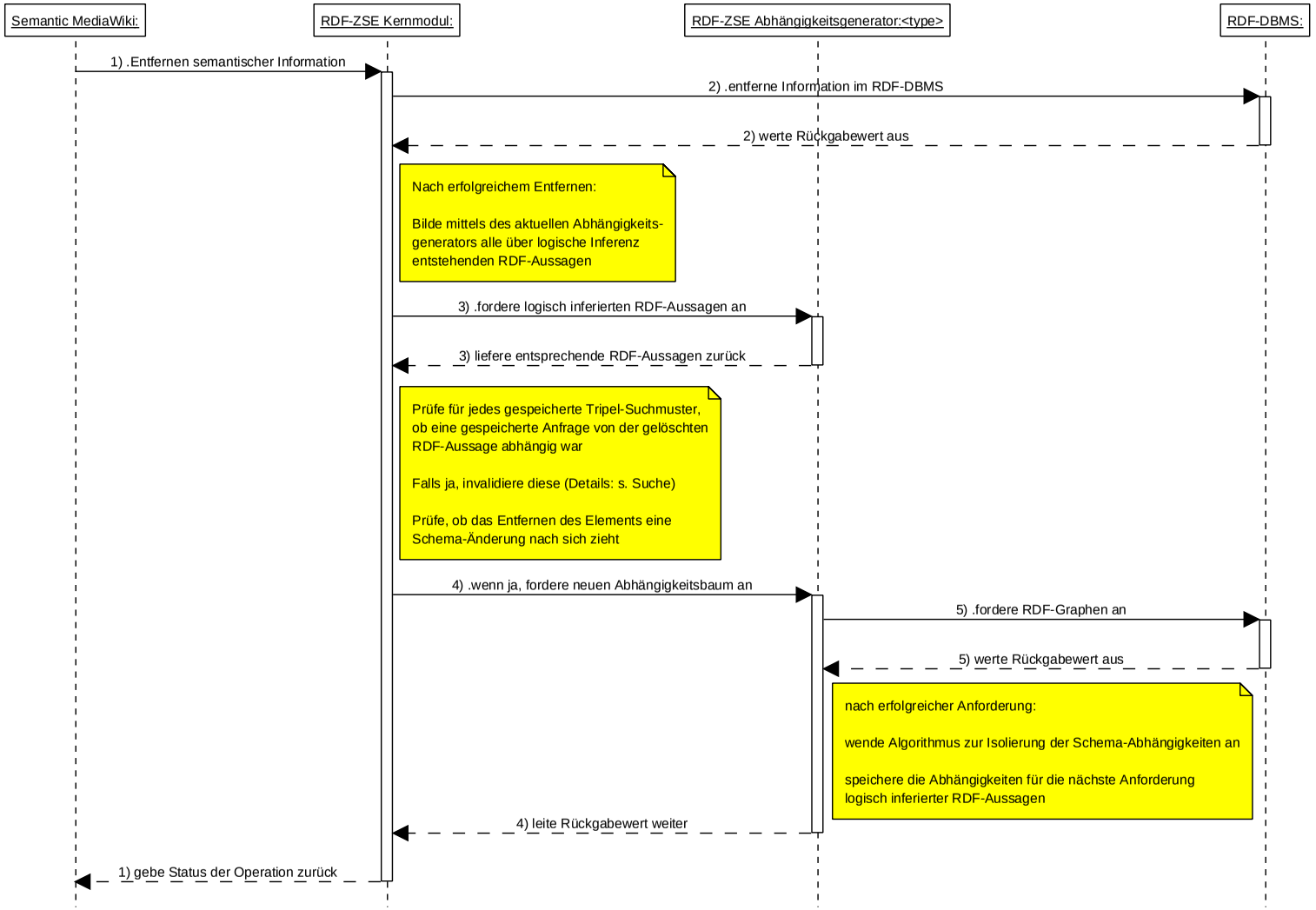

Abbildung 19: Vorgänge in der RDF-ZSE bei einer Lösch-Operation

6.2 Algorithmus

Der folgende Algorithmus in Pseudo-Code fasst die obigen Beschreibungen der verschiedenen Vorgänge in einer geschlossenen Darstellung zusammen. Allgemein werden Funktionen nur dann im Detail ausgeführt, wenn sie für das Verständnis der Abläufe wichtig sind. Insbesondere wird der Algorithmus aus Abschnitt 5.4 nicht in Pseudocode ausgeführt, da dies umfangreiche Einführungen weiterer Elemente nach sich ziehen würde, ohne echten Nutzen für die Darstellung der Zusammenhänge:

```
// Funktionen zur Interaktionen mit anderen Systemelementen

AnfrageResultat SendeAnfrageAnRdfDbms(SPARQLAnfrage q)
{
  AnfrageResultat r;
  r = RdfDbms->Ausfuehren(q);

  return r;
}

boolean SendeBefehlAnRdfDbms(EinfuegeLoeschBefehl c)
{
  boolean erfolgreich;
  erfolgreich = RdfDbms->Ausfuehren(c);

  return erfolgreich;
}

boolean SendeInvalidierungsbefehlAnMediawiki(Seitenindex i)
{
  MediaWiki->InvalidiereZwischenspeicher(i);
}

boolean SendeErfolgAnSemanticMediawiki(boolean erfolg)
{
  SemanticMediaWiki->SetzeStatus(erfolg);
}

boolean SendeResultatAnSemanticMediawiki(Anfrageresultat r)
{
  SemanticMediaWiki->Verarbeite(r);
}


// Funktionen des Generators

IRI[] ErhalteAbhaengigeElemente(IRI e)
{
  if (e in AbhaengigkeitsTabelle)
    return AbhaengigkeitsTabelle[e]
  else
    return array(e);
}
```

```
// Funktionen des Generators (Fortsetzung)

RDFAussage[] ErzeugeInferierteAussagen(RDFAussage a)
{
  RDFAussage[] InferierteRDFAussagen;

  foreach(Subjekt in ErhalteAbhaengigeElemente(a.Subjekt))
    InferierteRDFAussagen.add(Subjekt, a.Praedikat, a.Objekt);

  foreach(Praedikat in ErhalteAbhaengigeElemente(a.Praedikat))
    InferierteRDFAussagen.add(a.Subjekt, Praedikat, a.Objekt);

  foreach(Objekt in ErhalteAbhaengigeElemente(a.Objekt))
    InferierteRDFAussagen.add(a.Subjekt, a.Praedikat, Objekt);

  return InferierteRDFAussagen;
}

boolean PruefeAufSchemaAenderung(RDFAussage a)
{
  if (a.Subjekt in AbhaengigkeitsTabelle || a.Praedikat in
      Abhaengigkeitstabelle || a.Objekt in
      Abhaengigkeitstabelle)
    return true
  else
    return false;
}

boolean ErzeugeNeueAbhaengigkeitsTabelle()
{
  RDFTeilgraph g;
  Anfrageresultat r = SendeAnfrageAnRdfDbms(SPARQLAnfrage q);

  if (r == false)
    return false;

  // SchemaAbhaengigkeiten werden mittels des Algorithmus aus
      Abchnitt \ref{SchemaDependencies} isoliert
  g = r.ErhalteSchemaAbhaengigkeitsbaum();

  Abhaengigkeitstabelle.clear();

  foreach (Element in g)
  {
    foreach(AbhaengigesElement in Element.AbhaengigeElemente)
      Abhaengigkeitstabelle.add(Element,AbhaengigesElement);
  }

  return true;
}
```

```
// Kernmodul: Invalidierungs- und Einfüge-Logik

boolean PruefeAufTreffer(SPARQLAnfrage q)
{
  if ((Hash(q),q) in Anfragetabelle)
    return true
  else
    return false;
}


Anfrageresultat RufeResultatAb(SPARQLAnfrage q)
{
  Anfragetabelle.AktualisiereZugriffszeit(Hash(q),q,Systemzeit
      .Jetzt);
  return Anfragetabelle.RufeResultatAb(Hash(q),q);
}

Tripelsuchmuster[] PruefeTripelsuchmuster(RDFAussage a)
{
  Tripelsuchmuster[] SuchmusterTreffer = array();

  foreach(Suchmuster in Tripeltabelle)
  {
    trifft = Suchmuster.trifft(a);
    if (tritt == true)
      SuchmusterTreffer.add(Suchmuster);
  }

  return SuchmusterTreffer;
}

void InvalidiereEintrag(SPARQLAnfrage q)
{
  Seitenindex i;

  foreach(Muster in q.Tripelsuchmuster)
  {
    Tripeltabelle.remove(Muster, Hash(q), q);
  }

  Anfragetabelle.RufeSeitenindexAb(Hash(q),q);
  SendeInvalidierungsbefehlAnMediawiki(i)

  Anfragetabelle.remove(Hash(q), q);
}
```

```
// Kernmodul: Invalidierungs- und Einfüge-Logik (Fortsetzung)

void FuegeHinzuEintrag(SPARQLAnfrage q, Resultat r,
    Seitenindex i)
{
  while (Speicherverbrauch.Aktuell + r.Groesse >
      Speicherverbrauch.Maximal)
  {
    SPARQLAnfrage a = Anfragetabelle.SucheAeltestenEintrag();
    InvalidiereEintrag(a);
  }

  Anfragetabelle.add(Hash(q), q, r, Systemzeit.Jetzt, i);

  foreach(Muster in q.Tripelsuchmuster)
  {
    Tripeltabelle.add(Muster, Hash(q), q);
  }

}
```

```
// Eigentliche Verarbeitungslogik - angenommen wird,
// dass bei jeder semantischen Operation o die RDF-ZSE
// mit dieser Funktion aufgerufen wird
//
// Es wird außerdem angenommen, dass Semantic Mediawiki
// abhängig von der Operation entweder auf ein Resultat
// oder einen Erfolgsstatus wartet

void main(Operation o)
{
  Anfrageresultat r;
  boolean erfolgreich = true;

  switch(o.Typ)
  {
    case "Suche":
      r = PruefeAufTreffer(o.Anfrage);
      if (r == true)
      {
        return RufeResultatAb(o.Anfrage);
      }
      else
      {
        r = SendeAnfrageAnRdfDbms(o.Anfrage);
        FuegeHinzuEintrag(o.Anfrage, r, o.Seitenindex);

        // die Abarbeitung wurde erfolgreich beendet
        SendeResultatAnSemanticMediawiki(r);
      }
      break;

    case "Einfuegen":
    case "Loeschen":
      erfolgreich = SendeBefehlAnRdfDbms(o.Befehl);

      if (erfolgreich == false)
      {
        // Fehler beim Einfügen oder Löschen im RDF-DBMS
        SendeErfolgAnSemanticMediawiki(erfolgreich);
        break;
      }
      else
      {
        RDFAussage[] Aussagen;

        Aussagen = ErzeugeInferierteAussagen(o.Befehl.
            RDFAussage);

        Tripelsuchmuster[] GesammelteSuchmusterTreffer = array
            ();
```

```
            foreach (PruefAussage in Aussagen)
            {
              Tripelsuchmuster[] SuchmusterTreffer =
                  PruefeTripelsuchmuster(PruefAussage);

              foreach (Treffer in SuchmusterTreffer)
              {
                GesammelteSuchmusterTreffer.add(Treffer);
              }
            }

            foreach (Treffer in GesammelteSuchmusterTreffer)
            {
              SPARQLAnfrage[] InvalidiereAnfragen = array();

              foreach(Anfrage in InvalidiereAnfragen)
              {
                InvalidiereEintrag(Anfrage, Hash(Anfrage));
              }
            }

            boolean schemaElement = PruefeAufSchemaAenderung(o.
                Befehl.RDFAussage);

            if (schemaElement == false)
            {
              // die Abarbeitung wurde erfolgreich beendet
              SendeErfolgAnSemanticMediawiki(erfolgreich);
              break;
            }
            else
            {
              // Gesamtstatus ist abhängig von der Erzeugung der
                  neuen Abhängigkeitstabelle
              erfolgreich = ErzeugeNeueAbhaengigkeitstabelle()
              SendeErfolgAnSemanticMediawiki(erfolgreich);
              break;
            }
          }
        }

        return;
      }
```

7 Bewertung des Entwurfes

Dieses Kapitel untersucht theoretisch die mögliche Leistungssteigerung durch die entworfene RDF-ZSE. Im ersten Teil wird mit der Beschreibung einer Transformation von SPARQL in relationale Algebra eine Möglichkeit skizziert, wie die Resultate des zweiten Teils dieses Kapitels auch auf die derzeit in *Semantic MediaWiki* verwendete Speicherung semantischer Daten in einer MySQL-Datenbank übertragen werden können. Abschließend wird auch der Einfluss des Entwurfs sowie der Alternative, keinen Zwischenspeicher zu verwenden, auf die bestehende Architektur der *Wikipedia* diskutiert.

7.1 Transformation in relationale Algebra

Es soll nun das Resultat von Abschnitt 4.1 weiter abstrahiert werden, um für die nachfolgenden Betrachtungen eine Beschreibung zu ermöglichen, welche auch bei der Verwendung eines RDBMSs weitgehend ihre Gültigkeit behält. Dazu wird die in [Cyg05] beschriebene Transformation von SPARQL in relationale Algebra aufgegriffen und die wichtigsten Transformationen werden zusammengefasst.

7.1.1 Begriffe

Zunächst seien folgende Begriffe definiert:

- Ein *RDF Term* ist entweder ein IRI[52], ein RDF Literal oder ein leeren Knoten. Im hier beschriebenen relationalen Modell entspricht ein RDF Term einem Skalar.
- Eine *Variable* ist eine Anfragevariable wie sie durch die Definition von SPARQL ([P05]) festgelegt wird. Sie wird auch hier mit „*?variablenname*“ beschrieben.
- Ein *RDF Tupel* ist eine partielle Funktion von der Menge der Variablen auf die Menge der RDF Terme. Ein Tupel unterscheidet sich von einem gewöhnlichen Subjekt-Prädikat-Objekt RDF-Tripel dadurch, dass es RDF Terme lediglich gruppiert und keine semantische Bedeutung impliziert. Es ist jedoch eine $1:1$ Abbildung zwischen RDF-Tripeln und dreikomponentigen Tupeln (nachfolgend als *s/p/o-Tupel* bezeichnet). RDF Tupel sind letztlich nur eine andere Bezeichnung für Lösungen auf SPARQL-Anfragen.
- Die Variablen in einem Tupel seien als *Attribute* bezeichnet. Eine Variable wird als *gebunden* bezeichnet, wenn es sich in Definitionsbereich des Tupels befindet, ansonsten als *ungebunden*.

RDF- und Graph-Relationen Eine *RDF Relation* sei eine Menge von RDF Tupeln. Sie kann durch eine Tabelle dargestellt werden, wobei jede Zeile ein RDF Tupel und jede Spalte ein Attribut ist und nach einer Variable benannt wird. Als *Titel* der Relation sei dann die Menge aller Spaltennamen, d.h. Attribute, bezeichnet.

Eine *Graph-Relation* ist nun genau jene RDF Relation, die ein s/p/o-Tupel als Titel hat. Dies entspricht also gerade dem üblichen RDF Tripel.

[52] Internationaler Ressourcen Identifikator

7.1.2 Eine relationale Definition für SPARQL

Anmerkungen zur Definition relational-algebraischer Operatoren Die Operatoren der relationalen Algebra werden nachfolgend in der üblichen Bedeutung und Schreibweise verwendet (siehe etwa [Wik06h]). Analog zu [Cyg05] werden allerdings Umbenennungs- und Projektions-Operator unter dem Symbol des Projektionsoperators (π) kombiniert. Weiterhin seien der *bedingter innere Verbund* und der *bedingte linke äußere Verbund* so definiert, dass der Verbund-Operator angewendet und alle Tupel, die der Bedingung nicht entsprechen, eliminiert werden (die entsprechenden Attribute können dabei beiden Relationen entstammen). Das Symbol dieser Operatoren sei $\bowtie_p$ (es bezeichne p die Bedingung).

Übersetzung von SPARQL-Mustern in relationale Algebra Wie bereits im Abschnitt 2.2.2 erläutert, besteht aus verschiedenen Mustern, die rekursiver miteinander kombiniert werden können. Daher werden nachfolgend die wichtigsten Muster von SPARQL in relationale Algebra übersetzt, so dass wiederum durch ihre Kombination eine komplette Abbildung (bis auf einige in [Cyg05] beschriebene Sonderfälle) von SPARQL in relationale Algebra möglich ist.

Tripel-Suchmuster Einfache Tripel-Suchmuster können nahezu direkt übersetzt werden. Der Selektionsoperator schränkt auf bestimmte Werte ein, der Projektions-/Umbenennungsoperator bildet auf die gewünschten Variablennamen ab. So wird etwa

```
{  ?person  foaf:name ?name }
```

relational-algebraisch zu

$$\pi_{\substack{?person \leftarrow ?subjekt \\ ?name \leftarrow ?object}} (\sigma_{?predicate=foaf:name}(\text{Tripelmenge}))$$

Gruppierte Suchmuster, optionale Blöcke und Werteinschränkungen Gruppierte Suchmuster dienen in SPARQL dazu, aus einfachen Suchmustern komplexere zu kombinieren. Praktisch bedeutet dies vor allem, dass eine Lösung durch die Verwendung derselben Bindung für eine Variable in allen Bedingungen verwendet wird (siehe [PS06]). Außer bei optionalen Blöcken ist die Reihenfolge unerheblich.

Die Übersetzung erfolgt nun folgendermaßen:

1. Bildung des inneren Verbunds für alle Tripel-Suchmuster und Vereinigungen (*UNION*). Die relative Reihenfolge ist wegen der Kommutativität und Assoziativität des Verbund-Operators unerheblich.

2. Bildung des linken äußeren Verbunds aller optionalen Blöcke mit dem Resultat aus (1) in der Reihenfolge, in der sie auftreten (wobei die optionalen Blöcke auf der rechten Seite des Operators stehen, da sie mit niedrigerer Priorität ausgewertet werden müssen).

3. Anwendung aller Wertbeschränkungen auf das Resultat aus (2). Da die Beschränkungen Variablen aus allen Blöcken verwenden können, müssen

sie als letzte angewendet werden. Bei mehreren Filtern innerhalb einer Gruppe ist die relative Reihenfolge unerheblich.

So transformiert man beispielsweise das gruppierte Suchmuster

```
{ p1 . FILTER (p) . OPTIONAL { p2 } . p3 }
```

in den relational-algebraischen Ausdruck:

$$\sigma_p\left((p1 \ltimes p3) :\ltimes p2\right)$$

Dabei sind $p1$, $p2$ und $p3$ beliebige Such-Ausdrücke in SPARQL[53].

Vereinigung Das Vereinigungs-Suchmuster (*UNION*) kann direkt übersetzt werden. Aus

```
{ p1 } UNION { p3 }
```

wird der relational-algebraische Ausdruck:

$$p1 \cup p2$$

Fazit Mit obigen Transformationsregeln können alle hier relevanten SPARQL-Ausdrücke in relationale Algebra übersetzt werden. Das heißt aber auch, dass über diese Transformation der Einsatz der RDF-ZSE auch in einem *Semantic Wikipedia*-Szenario möglich ist, welches die derzeit verwendete MySQL-Datenbank zur Speicherung semantischer Informationen einsetzt.

7.2 Theoretische Leistungsbetrachtungen

Nachfolgend soll das Leistungsverhalten, der in Kapitel 6 beschriebenen Lösung für das Anfrageoptimierungs-Problem der *Semantic Wikipedia* im Vergleich zu einem System ohne eine entsprechende Zwischenspeicher-Einheit analysiert werden. Dazu wird zunächst die Leistung der RDF-ZSE isoliert betrachtet und die zur Bearbeitung der verschiedenen elementaren, semantischen Operationen nötige Zeit unter Verwendung des O-Kalküls berechnet.

Anschließend wird argumentativ die Verbesserung der Laufzeit erläutert. Dabei wird auf die Formalisierung des Szenarios aus Abschnitt 3.3 zurückgegriffen. Abschließend wird auch die Beeinflussung der bestehenden *Wikipedia*-Architektur kurz diskutiert.

7.2.1 Anfrage

Handelt es sich bei der aktuellen Operation um eine semantische Suchanfrage, so können zwei Fälle unterschieden werden: Entweder das gewünschte Resultat befindet sich im Zwischenspeicher oder es muss aus dem Datenspeicher geladen werden. In beiden Fällen muss auch den Verwaltungsoperationen der RDF-ZSE Rechnung getragen werden.

[53] An dieser Stelle gibt es Fälle, für die die Transformation nicht exakt definiert ist; diese sind aber hier nicht relevant

Treffer Im Falle eines Treffers (*hit*) kann die zur Beantwortung nötige Zeit t_{hit} im O-Kalkül beschrieben werden durch:

$$\begin{aligned} t_{\text{hit}} &= t_{\text{PruefeAufTreffer}} + t_{\text{Rest}} \\ &= t_{\text{Hash}} + t_{\text{SucheAnfrage}} + t_{\text{AktualisiereAnfrage}} + O(1) \\ &= O(1) + O(1) + O(\log a) + O(\log a) \\ &= O(\log a) \end{aligned}$$

Dabei ist a die Größe der Anfragetabelle (siehe Abschnitt 6.1). Es wird hierbei die Annahme gemacht, dass aufgrund des Speicherbedarfs eine reine Hashtabelle oder ein Trie zur Speicherung der Ergebnisse nicht in Frage kommen und stattdessen ein RDMBS, welches seine Daten im Speicher einer *Random Access Machine (RAM)* hält und einen Index zur beschleunigten Auffindung der Resultate einsetzt. Außerdem wird angenommen, dass die Zeit zum Übertragen der Ergebnisse zwischen der RDF-ZSE und *Semantic MediaWiki* vernachlässigbar ist.

Kein Treffer Liegt kein Treffer (*miss*) vor, so muss die Anfrage vom RDF-DBMS beantwortet werden. Wie bereits in Abschnitt 2.2.2 erwähnt, läuft der Algorithmus zum Finden isomorpher Teilgraphen in $O\left(|V| \log |V|\right)$, wobei $|V|$ die Summe der Knotenzahl im Such- und Basisgraph ist. Da bei hinreichend großen Datenmengen der Basisgraph deutlich größer ist, kann der Einfluss des Suchgraphen vernachlässigt werden. Sei also v (*vertices*) die Anzahl der Knoten im Basisgraphen.

Weiter wird angenommen, dass nach einer gewissen Zeit im Mittel jeweils ein alter Eintrag zum Einfügen eines neuen Eintrages verdrängt werden muss:

$$\begin{aligned} t_{\text{miss}} &= O(v \log v) + t_{\text{EinfügenEintrag}} + t_{\text{Rest}} \\ &= O(v \log v) + t_{\text{Hash}} + t_{\text{Verdraenge}} \\ &\quad + t_{\text{EinfügeAnfrage}} + t_{\text{AktualisiereMuster}} + O(1) \\ &= O(v \log v) + O(1) + O(\log a) + O(\log m) + O(\log a) + O(x) \\ &\quad + O(\log a) + O(\log m) \\ &= O(v \log v) + O(\log a) + O(\log m) + O(x) \\ &= O(v \log v) + O(\log a) + O(x) \end{aligned}$$

Dabei ist m die Größe der Tripeltabelle (siehe Abschnitt 6.1), x ist die algorithmische Größenordnung, in der die Invalidierung in *MediaWiki* stattfinden kann.

7.2.2 Einfüge-/Lösch-Operationen

Handelt es sich bei der aktuellen Operation um eine semantische Einfüge- oder Lösch-Operation (*ins(ert)/ del(ete)*), so fällt zusätzlich zu der eigentlichen Zeit, die nötig ist, um es im RDF-DMBS einzufügen/zu löschen (es sei hier $O(\log v)$ angenommen), noch der Aufwand für die Verwaltungsoperationen der RDF-ZSE

Rechnung an:

$$
\begin{aligned}
t_{\text{ins/del}} &= t_{\text{RdfDbms}} + t_{\text{Inferiere}} + n_{inf} \cdot t_{\text{PruefeTripel}} \\
&\quad + n_{trip} \cdot t_{\text{InvalidiereEintrag}} + t_{\text{SchemaElement}} \\
&\quad + p_s \cdot t_{\text{neueAbhängigkeiten}} + t_{\text{Rest}} \\
&= O(v \log v) + O(\log g) + O(\log g) \cdot O(\log m) \\
&\quad + O(m \log g) \cdot (O(\log m) + O(\log a) + O(x)) + O(\log g) \\
&\quad + p_s \cdot (O(v) + O(g \log g)) + O(1) \\
&= O(v \log v) + O(m \log g) + O(m^2 \log g) + O(m \log g \log a) \\
&\quad + O(x \cdot m \log g)) + O(\log g) \\
&\quad + p_s \cdot (O(v) + O(g \log g)) \\
&= O(v \log v) + O(a^2 \log g) + O(x \cdot a \log g)) \\
&\quad + p_s \cdot (O(v) + O(g \log g))
\end{aligned}
$$

Die Bezeichnungen für a, m, v und x seien wie im obigen Fall der Suche definiert. Desweiteren sei g die Anzahl der Elemente in der Abhängigkeitstabelle (siehe Abschnitt *AnalyseSemanticBasic*), n_{inf} bezeichne die durchschnittliche Anzahl der durch logische Inferenz (aus der aktuell betrachteten RDF-Aussage) entstehenden RDF-Aussagen, n_{trip} die Anzahl der Tripel, welche auf diese zutreffen und p_s die Wahrscheinlichkeit, dass das Einfügen/Löschen der aktuellen RDF-Aussage eine Schemaänderung nach sich zieht.

n_{inf} hängt dabei von g ab - die Abschätzung, dass $n_{inf} = O(\log g)$ gründet auf der Überlegung, dass im Abhängigkeitsgraphen (siehe Abschnitt 5.4) jedes Element im Mittel mit $\lfloor \frac{1}{2} \cdot \log g \rfloor$ anderen Elementen verbunden ist, solange der Baum nicht zu stark entartet.

n_{trip} hängt ebenfalls von g ab, da es von n_{inf} abhängt. Zudem hängt es von m ab, da die Anzahl der Treffer von Tripelsuchmustern sich mit ihrer Anzahl erhöht. Somit erhält man $n_{trip} = O(m \log g)$

7.2.3 Fazit

Die zur Beantwortung von Anfragen wichtige Zeit t_{hit} liegt in $O(\log a)$. Setzt man eine Trefferrate ähnlich der des Szenarios von [LR01] (auf der analysierten Portalseite für Börsenkurse 60% der Anfragen 0,8% der Seiten betrafen; betrachtet man 11,8% der Seiten, so sind 90% der Anfragen abgedeckt) oder jener der konventionellen *Wikipedia* ([Wik05] spricht von einer Trefferrate von über 85%), so hat man in diesem Fall eine deutliche Senkung der Antwortzeit erreicht, da bei Zugriff auf das RDF-DMBS eine Suche in $O(v \log v)$ durchgeführt werden muss. Diese ist zudem noch signifikant langsamer, wenn das RDF-DBMS seine Daten nicht im Arbeitsspeicher hält, sondern auf Festplatten-Speicher zurückgreifen muss.

Zeitgewinne/-verluste gegenüber einer Version ohne Zwischenspeicher Auch bei höherer Verflechtung von Seiten durch integrierte Anfragen erscheint es plausibel, anzunehmen, dass die Trefferrate über 50% liegt. Das bedeutet aber nichts anderes, als dass die zur Beantwortung einer semantischen

Anfrage nötige Zeit im Mittel auf $O(\log a)$ fällt. Greift man auf die Bezeichnungen von Abschnitt 3.3 zurück, so erhält man ingesamt für den Zeitgewinn

$$\left(\sum_{i=1}^{s} \#s_i\right) \cdot O(v \log v) - \left(\sum_{i=1}^{s} \#s_i\right) \cdot O(\log a)$$

Dem entgegen steht die Erhöhung der Antwortzeit im Nicht-Treffer-Fall um $O(\log a) + O(x)$. Dies fällt jedoch gegenüber $O(v \log v$ nicht ins Gewicht, inbesondere dann nicht, wenn das RDF-DMBS seine Daten nicht im Arbeitsspeicher hält.

Entscheidend ist für das Urteil über das Leistungsverhalten der RDF-ZSE daher vor allem der zusätzliche Aufwand für das Einfügen oder Löschen von semantischen Informationen. Hier beträgt der Mehraufwand pro Operation

$$M = O(a^2 \log g) + O(x \cdot a \log g)) + p_s \cdot (O(v) + O(g \log g))$$

Dabei verdient vor allem der letzte Term von M Beachtung. p_s drückt die Wahrscheinlichkeit aus, dass ein neu eingefügtes/ein gelöschtes Element ein Schemaelement ist. In diesem Fall muss aufwendig $(O(v) + O(g \log g))$ eine neue Abhängigkeitstabelle erzeugt werden. Das Optimierungspotential an dieser Stelle wird im folgenden Kapitel diskutiert.

Geht man weiterhin davon aus, dass die Invalidierungszeit x etwa in der Größenordnung von $\log a$ liegt, d.h. dass die *Wikipedia*-Zwischenspeicher Größenordnung der RDF-ZSE liegen, vereinfacht sich M weiter zu:

$$M = O(a^2 \log g) + p_s \cdot (O(v) + O(g \log g))$$

Im Mittel wird dabei das betreffende Element kein Schemaelement sein. Das heißt, wiederum unter Rückgriff auf die Bezeichnungen aus Abschnitt 3.3, der Gesamtmehraufwand liegt in:

$$\left(\sum_{i=1}^{d} \#d_i\right) \cdot O(a^2 \log g)$$

Unter Rückgriff auf das Szenario, d.h. die Tatsache, dass es wesentlich mehr Anfragen als Änderungen gibt, kann also festgehalten werden, dass der Einsatz der RDF-ZSE im gegebenen Szenario lohnenswert erscheint.

Verhalten bei Integration in die *Wikipedia*-Architektur Bei den bisherigen Leistungsbetrachtungen wurde noch nicht darauf eingegangen, welchen Einfluss der Einsatz der beiden Lösungen (kein semantischer Zwischenspeicher, Einsatz der RDF-ZSE) jeweils auf die Zwischenspeicher der bestehenden *Wikipedia*-Architektur hat. Dazu reicht jedoch bereits eine einfache Überlegung, welche den Einsatz einer Lösung ohne semantischen Zwischenspeicher als Alternative ausscheiden lässt.

Setzt man nämlich voraus, dass integrierte semantischen Anfragen stets aktuelle Resultate liefern sollen, so müssen die entsprechenden Zwischenspeicher der *Wikipedia*-Architektur immer dann invalidiert werden, wenn eine Änderung an den semantischen Daten eine Anfrage auf der betrachteten Seite betrifft. Existiert zu diesem Zweck keine Entscheidungslogik, wie es in einem Szenario

ohne semantischen Zwischenspeicher der Fall ist, so müssen zur Garantie aktueller Resultate sämtliche Zwischenspeicher mindestens für diese Seite deaktiviert werden. Da dies in der aktuellen Version von *MediaWiki* nicht implementiert ist, müssen die Zwischenspeicher sogar komplett deaktiviert werden, was das Leistungsverhalten drastisch verschlechtert.

Eine geringe Beeinträchtigung der Leistungsfähigkeit der bestehenden *Wikipedia*-Architektur tritt jedoch, aufgrund der potentiell höheren Invalidierungsrate der Zwischenspeicher, auch bei Einsatz der RDF-ZSE auf.

8 Fazit und Ausblick

Im Rahmen dieser Arbeit wurde das Szenario partizipativer, semantischer Portale am Beispiel des Projektes *Semantic Wikipedia* untersucht. Die Analyse (Kapitel 3) ging dabei vom allgemeinen Fall eines partizipativen Portals aus und wählte als speziellen Anwendungsfall die Online-Enzyklopädie *Wikipedia*. Es wurde sowohl die Architektur der *Wikipedia* als auch jene der *Semantic Wikipedia* analysiert und ein Vergleich zwischen beiden durchgeführt.

Problemstellung Dies erlaubte es, das Problem der Leistungsverbesserung der *Semantic Wikipedia* auf ein Problem der Anfrageoptimierung zu reduzieren. Als Resultat wurden in Abschnitt 3.5 die Anforderungen an eine Lösung zur Anfrageoptimierung festgehalten:

K1 Verbesserung der mittleren Antwortzeit bei der Anfragebeantwortung

K2 Garantie aktueller Anfrageergebnisse (korrekte Invalidierung bei Änderungen)

K3 Integration in die bestehende Architektur; keine (oder möglichst geringe) negative Beeinflussung der bestehenden Architektur

Lösungsentwurf Existierende Optimierungsmöglichkeiten wurden in Kapitel 4 auf ihren Nutzen im konkreten Szenario geprüft - die Wahl für auf einen Zwischenspeicher. Dabei wurde über eine Transformation der Anfragesprache (SMW-QL) der *Semantic Wikipedia* in SPARQL ([P05]) die Lösung über das konkrete Szenario hinaus so konzipiert, dass sie allgemein als Zwischenspeicher für Anfragen an SPARQL-kompatible RDF-Datenspeicher geeignet ist.

Intelligente Invalidierung Als besonderes Problem stellte sich dabei die korrekte Invalidierung (*K2*) gespeicherter Einträge dar, wenn man annimmt, dass im RDF-Datenspeicher logische Inferenz unterstützt wird. Dieses Problem wurde in Kapitel 5 für die Klasse der RDF-Datenspeicher gelöst, welche RDF-Schema (RDFS) kompatible Ontologiesprachen einsetzen. Dabei werden lediglich das Typ-, das Eigenschafts- und das Klassenelement vorausgesetzt. Für Ontologiesprachen wie OWL, die es erlauben, negative Aussagen über die Beziehung zwischen Elementen des RDF-Graphen zu machen (beispielsweise auszudrücken, dass zwei Klassen disjunkt sind, d.h. keine gemeinsamen Instanzen haben) ist außerdem eine Liste dieser Beziehungen notwendig.

Leistungs- und Integrationsaspekte Die theoretische Leistungsbewertung in Abschnitt 7.2 geben ungefähre Werte für die zu erwartende Leistungssteigerung an. Inbesondere die Beantwortung integrierter Anfragen profitiert deutlich vom Einsatz der konzipierten Zwischenspeicherlösung (*K1*). Zudem wird am Ende von Abschnitt 7.2.3 festgehalten, dass die Invalidierungslogik der konzipierten Zwischenspeichereinheit notwendig ist, um die Funktionsfähigkeit der bestehenden *Wikipedia*-Architektur aufrecht zu erhalten (*K3*), wenn aktuelle Resultate auf integrierte semantische Anfragen geliefert werden sollen (*K2*).

8.1 Verbesserungsmöglichkeiten

Zum jetzigen Zeitpunkt sind Verbesserungen an dem entworfenen Zwischenspeiche vor allem an drei Punkten denkbar.

Anfragezerlegung In [CR94] entwickeln die Autoren ein System zur Optimierung des Leistungsverhaltens Ihres relationalen Datenbanksystems, indem temporäre Teilresultate bei der Beantwortung späterer Anfragen wiederverwendet werden. Daran angelehnt ist es denkbar, die hier entworfene Zwischenspeicher-Einheit dergestalt zu modifizieren, dass in jenem Fall, in dem ein Resultat nicht direkt im Zwischenspeicher vorliegt, die aktuelle Anfrage in Teilanfragen zerlegt wird und geprüft wird, ob sich diese sämtlich oder teilweise im Zwischenspeicher befinden.

Die ausstehenden Teilresultate können aus dem RDF-Datenspeicher gelesen werden. Anschließen werden die Teilresultate im Zwischenspeicher wieder kombiniert, wobei in der Regel Verbund-Operationen (*joins*) durchzuführen sind.

Mit dieser Verbesserung könnte die Trefferrate des Zwischenspeichers gegenüber dem bisherigen Lösungsentwurf noch einmal erhöht werden. Es fällt allerdings zusätzlicher Aufwand an, da Teilresultate im Zwischenspeicher wieder rekombiniert werden müssen. Es ist dabei zu untersuchen, ob im Falle fehlender Teilresultate das Nachladen aus dem RDF-Datenspeicher und anschließendes Kombinieren der Teilresultate tatsächlich günstiger ist als in diesem Fall einen Nicht-Treffer für den Zwischenspeicher anzunehmen und die Anfrage komplett vom RDF-Datenspeicher beantworten zu lassen.

Durchführen von Arbeitsschritten im RDF-DBMS Einen weiteren Ansatzpunkt zur Verbesserung stellt die Zwischenspeicherung der Abhängigkeiten zwischen Schema-Elementen dar. Hier sind zweierlei Punkte wichtig:

- Die Prüfung, ob ein Element ein Schema-Element ist, kann im RDF-Datenspeicher in $O(1)$ durchgeführt werden, da dort die Ontologiesprache bekannt ist
- Die Erzeugung des Abhängigkeitsbaums (siehe Abschnitt 5.4) wäre ebenfalls effizienter im RDF-Datenspeicher möglich. Da die Ontologiesprache dort bekannt ist, kann die Laufzeit in den Bereich von $O(g)$ reduziert werden.

Zudem ist es auf diesem Wege möglich, einen gänzlich von konkreten Ontologiesprachen unabhängigen Zwischenspeicher zu entwerfen. Voraussetzung ist dann allerdings die Unterstützung dieser beiden Funktionalitäten im verwendeten RDF-Datenspeicher.

Inferieren von Tripel-Suchmuster Um die Invalidierung von Elementen zu beschleunigen, ist es denkbar, statt der logisch inferierten RDF-Aussagen, welche aus der Ausgangs-Aussage hervorgehen, zu einer RDF-Aussage alle $8 = 2^3$ möglichen Tripelsuchmuster zu bilden, aus diesen über logische Inferenz alle abgeleiteten Suchmuster zu erhalten und anschließend zu prüfen, ob diese in der Tripeltabelle enthalten sind.

Auf diese Weise könnte der Aufwand für die Prüfung auf betroffene Anfragen deutlich reduziert werden. Zu Untersuchen ist zudem, inwieweit diese Verbesserungsmöglichkeit mit der zweiten kombiniert werden kann, um die Effizienz weiter zu steigern.

Abbildungsverzeichnis

Listings

Literatur

[AI06] Alexa Internet, Inc.: *Alexa Traffic Ranking*, 09 2006. `http://www.alexa.com/site/ds/top_sites?ts_mode=global&lang=none` - letzter Abruf: 20.09.2006.

[App] Appmosphere, Benjamin Nowack: *ARC: appmosphere RDF Classes for PHP Developers.*

[Bar99] Barnick, D. et al: *Q and A: Trends in Internet and Enterprise Portals*, 09 1999. Gartner Group Research Note.

[Bec04] Beckett, Dave: *RDF/XML Syntax Specification (Revised)*, 02 2004. `http://www.w3.org/TR/rdf-syntax-grammar/` - Online-Ressource, letzter Abruf: 25.10.2006.

[Bec06a] Becket, Dave: *SPARQL RDF Query Language Reference V1.8 (A4)*, 02 2006. `http://www.dajobe.org/2005/04-sparql/SPARQLreference-1.8.pdf` - Online-Ressource, letzter Abruf: 26.10.2006.

[Bec06b] Beckett, Dave: *Turtle - Terse RDF Triple Language*, 04 2006. `http://www.dajobe.org/2004/01/turtle/` - Online-Ressource, letzter Abruf: 26.10.2006.

[BHLT06] Bray, Tim, Dave Hollander, Andrew Layman und Richard Tobin: *Namespace in XML 1.0 (Second Edition), Qualified Names*, 08 2006. `http://www.w3.org/TR/xml-names/\#ns-qualnames` - Online-Ressource, letzter Abruf: 26.10.2006.

[BL00a] Berners-Lee, Tim: *Primer - Getting into the semantic web and RDF using N3*, 10 2000. `http://www.w3.org/2000/10/swap/Primer.html` - Online-Ressource, letzter Abruf: 26.10.2006.

[BL00b] Berners-Lee, Tim: *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web.* Collins, 11 2000.

[BL03] Berners-Lee, Tim: *Spinning the Semantic Web*, Kapitel Foreword, Seiten xi – xxiii. MIT Press, 2003.

[BLHL01] Berners-Lee, Tim, James Hendler und Ora Lassila: *The Semantic Web.* Scientific American, 284(5):28–37, 05 2001.

[BM01] Biron, Paul V. und Ashok Malhotra: *XML Schema Part 2: Datatypes*, 02 2001. `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/` - Online-Ressource, letzter Abruf: 25.10.2006.

[BM06] Brickley, Dan und Libby Miller: *FOAF Vocabulary Specification*, 01 2006. `http://xmlns.com/foaf/0.1/` - Online-Ressource, letzter Abruf: 28.10.2006.

[BO04] Bizer, C. und R. Oldakowski: *RAP: RDF API for PHP*, 2004.

[CDI99] Challenger, Jim, Paul Dantzig und Arun Iyengar: *A Scalable System for Consistently Caching Dynamic Web Data.* In: *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, New York, 1999.

[Com98] Computerwoche: *Web-Portale entwickeln sich zunehmend zu Goldgruben*, Jul 1998. Ausgabe 29/1998.

[Com00a] Computerwoche: *Die ganze systemtechnische Intelligenz ins Portal investieren*, Feb 2000.

[Com00b] Computerwoche: *Portale sind eine Reise und kein Ziel*, Nov 2000. Ausgabe 44/2000.

[Com06] Computerwoche: *Gartner beurteilt Reifegrad neuer IT-Techniken*, Aug 2006. Gartner Hype Cycle for Emerging Technologies 2006.

[CR94] Chen, Chung-Min und Nick Roussopoulos: *The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching.* In: *Extending Database Technology*, Seiten 323–336, 1994.

[Cyg05] Cyganiak, Richard: *A relational algebra for SPARQL.* Technischer Bericht, Digital Media Systems Laboratory, HP Laboratories Bristol, 09 2005.

[Doc06] Documentation, ARQ: *ARQ - SPARQL Tutorial*, 08 2006. `http://jena.sourceforge.net/ARQ/Tutorial/` - Online-Ressource, letzter Abruf: 25.10.2006.

[EE04] Eckstein, Rainer und Silke Eckstein: *XML und Datenmodellierung*, Kapitel 6, Seiten 235–283. dpunkt.verlag, 2004.

[Fro06] Frost, Ingo: *Zivilgesellschaftliches Engagement in virtuellen Gemeinschaften? Eine systemwissenschaftliche Analyse des deutschsprachigen Wikipedia-Projektes.* Diplomarbeit, Universität Onsabrück, 2006.

[GJ79] Garey, Michael R. und David S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Company, 1979.

[Hay04] Hayes, Patrick: *RDF Semantics*, 02 2004. `http://www.w3.org/TR/rdf-mt/` - Online-Ressource, letzter Abruf: 25.10.2006.

[HD05] Harth, A. und S. Decker: *Optimized index structures for querying rdf from the web*, 2005.

[HN95] Hald, Anton und Wolf Nevermann: *Datenbank-Engineering für Wirtschaftsinformatiker*, Kapitel 8.4.4, Seiten 196–198. vieweg, 1995.

[HS05] Harris, Stephen und Nigel Shadbolt: *SPARQL Query Processing with Conventional Relational Database Systems.* In: *WISE Workshops*, Seiten 235–244, 2005.

[Ian01] Iannella, Renato: *Representing vCard Objects in RDF/XML*, 02 2001. `http://www.w3.org/TR/vcard-rdf` - Online-Ressource, letzter Abruf: 26.10.2006.

[IC97] Iyengar, Arun und Jim Challenger: *Improving Web Server Performance by Caching Dynamic Data.* In: *USENIX Symposium on Internet Technologies and Systems*, 1997.

[IC98] Iyengar, A. und J. Challenger: *Data Update Propagation: A Method for Determining How Changes to Underlying Data Affect Cached Objects on the Web*, 1998.

[IET98] IETF: *RFC 2426 - vCard MIME Directory Profile*, 09 1998. `http://www.ietf.org/rfc/rfc2426.txt` - Online-Ressource, letzter Abruf: 26.10.2006.

[Ioa96] Ioannidis, Yannis E.: *Query optimization.* ACM Computing Surveys, 28(1):121–123, 1996.

[KC04] Klyne, Graham und Jeremy J. Carroll: *Resource Description Framework (RDF): Concepts and Abstract Syntax*, 02 2004. `http://www.w3.org/TR/rdf-concepts/` - Online-Resource, letzter Abruf: 25.10.2006.

[KGHV04] Kirchhof, Anja, Thorsten Gurzki, Henning Hinderer und Joannis Vlachakis: *Was ist ein Portal? Definition und Einsatz von Unternehmensportalen*, 06 2004.

[Kir05] Kirk, Alexander: *Caching Strategies for Load Reduction on High Traffic Web Applications.* Diplomarbeit, Technische Universität Wien, 05 2005.

[Krs06] Krstic, Ivan: *[Wikitech-l] Some questions about Wikipedia (caching, DB-updates)*, 04 2006. `http://mail.wikipedia.org/pipermail/wikitech-l/2006-April/035094.html` - Online-Ressource, letzter Abruf: 29.10.2006.

[LC01] Leuf, Bo und Ward Cunningham: *The Wiki Way: Collaboration and Sharing on the Internet.* Addison-Wesley Professional, April 2001.

[LR01] Labrinidis, Alexandros und Nick Roussopoulos: *Update Propagation Strategies for Improving the Quality of Data on the Web.* In: *The VLDB Journal*, Seiten 391–400, 2001.

[LS98] Lassila, O. und R. Swick: *Resource Description Framework (RDF) model and syntax specification*, 10 1998.

[Mar01] Markatos, Evangelos P.: *On caching search engine query results.* Computer Communications, 24(2):137–143, 2001.

[McC05] McCarthy, Philip: *Search RDF data with SPARQL*, 05 2005. http://www-128.ibm.com/developerworks/xml/library/j-sparql/ - Online-Ressource, letzter Abruf: 26.10.2006.

[MMW06] Malhotra, Ashok, Jim Melton und Norman Walsh: *XQuery 1.0 and XPath 2.0 Functions and Operators - Regular Expression Syntax*, 06 2006. http://www.w3.org/TR/xpath-functions/\#regex-syntax - Online-Ressource, letzter Abruf: 28.10.2006.

[MSS+01] Maedche, Alexander, Steffen Staab, Nenad Stojanovic, Rudi Studer und York Sure: *SEAL – A Framework for Developing SEmantic Web PortALs*. Lecture Notes in Computer Science, 2097:1–21, 2001.

[Ont06] Ontoworld.org: *Help:Semantics, Semantic MediaWiki Documentation*, 09 2006. http://ontoworld.org/wiki/Help:Semantics - Online-Ressource, letzter Abruf: 25.10.2006.

[O'R05] O'Reilly, Tim: *What Is Web 2.0*, 09 2005. http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html?page=1 - Online-Ressource, letzter Abruf: 25.10.2006.

[P05] Prud'hommeaux, Eric und Andy Seaborne. .: *SPARQL query language for RDF*, July 2005. Technical report, W3C working draft.

[PDTU00] Prof. Dr. Theo Ungerer, Prof. Dr. Winfried Görker und Prof Dr. Detlef Schmid: *Rechnerstrukturen*, Band 1. Universität Karlsruhe, 10 Auflage, 04 2000.

[PS06] Prud'hommeaux, Eric und Andy Seaborne: *SPARQL Query Language for RDF - Group graph patterns*, 10 2006. http://www.w3.org/TR/rdf-sparql-query/\#GroupPatterns - Online-Ressource, letzter Abruf: 01.11.2006.

[PSHH04] Patel-Schneider, Peter F., Patrick Hayes und Ian Horrocks: *OWL Web Ontology Language Semantics and Abstract Syntax*, 02 2004. http://www.w3.org/TR/owl-semantics/ - Online-Ressource, letzter Abruf: 03.11.2006.

[Rev04] Review, Technology: *Das Unvollendete*, 11 2004. Seite 50 ff.

[RSC04] Reynolds, Dave, Paul Shabajee und Steve Cayzer: *Semantic information portals*. In: *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, Seiten 290–291, New York, NY, USA, 2004. ACM Press.

[Sca01] Scarcello, Francesco: *Answering Queries: Tractable Cases and Optimizations*, 04 2001.

[Sch05] Schlossnagle, George: *Professionelle PHP 5 Programmierung*, Kapitel 2, Seiten 253–321. Addison Wesley, 2005.

[SK06] Sintek, Michael und Malte Kiesel: *RDFBroker: A Signature-Based High-Performance RDF Store.* In: *ESWC*, Seiten 363–377, 2006.

[Tod05] Today, USA: *A false Wikipedia 'biography'*, 11 2005. `http://www.usatoday.com/news/opinion/editorials/2005-11-29-wikipedia-edit_x.htm` - Online-Ressource, letzter Abruf: 25.10.2006.

[VKV+06a] Völkel, Max, Markus Krötzsch, Denny Vrandecic, Heiko Haller und Rudi Studer: *Semantic Wikipedia.* In: *Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, May 23-26, 2006*, MAY 2006.

[VKV+06b] Völkel, Max, Markus Krötzsch, Denny Vrandecic, Heiko Haller und Rudi Studer: *Wikipedia and the Semantic Web, Part II.* not yet published, 2006.

[VS06] Völkel, Max und Sebastian Schaffert (Herausgeber): *SemWiki 2006 - First Workshop on Semantic Wikis - From Wiki to Semantics*, 2006. `http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-206/` - Online-Ressource, letzter Abruf: 07.11.2006.

[W3C06] W3C: *Resource Description Framework*, 07 2006. `http://www.w3.org/RDF/` - Online-Resource, letzter Abruf: 25.10.2006.

[Web06] Weber, Leon: *Statistics-graphs of requests to the wikimedia-clusters*, 07 2006. `http://tools.wikimedia.de/~leon/stats/reqstats/reqstats-yearly.png` - Online-Ressource, letzter Abruf: 23.07.2006.

[Wik05] Wikimedia: *Cache strategy*, 03 2005. `http://meta.wikimedia.org/wiki/Cache_strategy` - Online-Ressource, letzter Abruf: 23.07.2006.

[Wik06a] Wikimedia: *Extending wiki markup - Meta (Parser functions)*, 10 2006. `http://meta.wikimedia.org/wiki/Extending_wiki_markup\#Parser_functions` - Online-Ressource, letzter Abruf: 29.10.2006.

[Wik06b] Wikimedia: *Multicast HTCP purging protocol*, 04 2006. `https://wikitech.leuksman.com/view/Multicast_HTCP_purging` - Online-Ressource, letzter Abruf: 23.07.2006.

[Wik06c] Wikimedia: *PHP caching and optimization*, 10 2006. `http://meta.wikimedia.org/wiki/PHP_caching_and_optimization` - Online-Ressource, letzter Abruf: 25.10.2006.

[Wik06d] Wikimedia: *Statistik über Wikipedia (englische Ausgabe)*, 07 2006. `http://stats.wikimedia.org/DE/TablesWikipediaEN.htm` - Online-Ressource, letzter Abruf: 23.07.2006.

[Wik06e] WIKIMEDIA: *Tugela Cache*, 05 2006. http://meta.wikimedia.org/wiki/Tugela_Cache - Online-Ressource, letzter Abruf: 23.07.2006.

[Wik06f] WIKIMEDIA: *Wikimedia servers - Meta*, 10 2006. http://meta.wikimedia.org/wiki/Wikimedia_servers - Online-Ressource, letzter Abruf: 29.10.2006.

[Wik06g] WIKIPEDIA: *Portal (Informatik)*, 10 2006. http://de.wikipedia.org/wiki/Portal_%28Informatik%29 - Online-Ressource, letzter Abruf: 25.10.2006.

[Wik06h] WIKIPEDIA: *Relationale Algebra*, 10 2006. http://de.wikipedia.org/wiki/Relationale_Algebra - Online-Ressource, letzter Abruf: 01.11.2006.

[Wik06i] WIKIPEDIA: *Wikipedia*, 10 2006. http://de.wikipedia.org/wiki/Wikipedia - Online-Ressource, letzter Abruf: 25.10.2006.

[Wik06j] WIKIPEDIA: *Wikipedia:Purge*, 08 2006. http://en.wikipedia.org/wiki/WP:PURGE - Online-Ressource, letzter Abruf: 29.10.2006.

[Wik06k] WIKIPEDIA: *WikiWikiWeb*, 09 2006. http://en.wikipedia.org/wiki/WikiWikiWeb - Online-Ressource, letzter Abruf: 15.09.2006.

„Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzen Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.“

(Ort, Datum) Unterschrift