Universität Leipzig
Institut für Informatik

Implementierung eines effizienten datenbankbasierten SPARQL-Prozessors und Erweiterungen zu SPARQL

Diplomarbeit

Leipzig, August 2007

vorgelegt von
Christian Weiske
Studiengang Informatik

Betreuender Hochschullehrer: Prof. Dr. Klaus-Peter Fähnrich
Institut für Informatik, Abteilung Betriebliche Informationssysteme

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Thema dieser Diplomarbeit ist die Implementierung eines effizienten, datenbankgestützten SPARQL-Prozessors. Weiterhin werden Erweiterungen zum SPARQL-Standard vorgeschlagen, da das Protokoll im täglichen Einsatz einige wichtige Funktionen vermissen lässt. Dieses Kapitel gibt Aufschluss über die Gründe und Ziele dieser Entwicklung.

1.1 Motivation

Das Internet ist ein ständig wachsender Informationsraum. Diese Informationen werden von Menschen geschrieben, oder von Programmen aus existierenden Daten in für Menschen verständlicher Form aufbereitet. Die zunehmende Anzahl an Websites und Texten erschwert es zusehends, benötigte Informationen zu erreichen.Programme können hierbei nur unzureichend Unterstützung geben, da die für Menschen gemachten Informationen für Maschinen nur sehr schwer zu interpretieren, und ohne Kontextinformationen auch nicht zu verstehen sind.

Das semantische Netz als Erweiterung des Internets wurde erdacht, um dieses Problem zu lösen. Informationen sollen als Daten - als in für Programme einfach zu verstehender Form - bereitgestellt werden. Die am weitesten verbreitetste Technologie dafür ist das *Resource Description Framework* (RDF). Mit diesem können die Daten nicht nur maschinenlesbar gespeichert werden, sondern auch auf verwandte Informationen an anderen Stellen verweisen.

Mit zunehmender Verbreitung von RDF benötigen wir Möglichkeiten, um die verfügbaren Informationen mit standardisierten Methoden abfragen zu können. Dadurch würde es möglich, dass alle Programme mit Unterstützung für solch eine Abfragesprache mit sehr geringem Aufwand auf Wissens-

basen zugreifen können. Eine solche Abfragesprache für RDF-Daten ist die *SPARQL Protocol And Query Language* (SPARQL).

Ein Ziel dieser Diplomarbeit ist es, eine eine effiziente Implementierung von SPARQL auf Basis einer relationalen Datenbank zu schaffen. Die in unserer Arbeitsgruppe *Agile Knowledge Engineering and Semantic Web* (AKSW) entwickelten Anwendungen arbeiten mit RDF Daten. Eine robuste Implementierung von SPARQL zur Abfrage dieser Daten ist deshalb unerlässlich für die weitere Entwicklung der Applikationen.

Im folgenden Abschnitt werden die Ziele genauer ausgeführt.

1.2 Ziele

Die *RDF API for PHP* (RAP) ist eine Funktionsbibliothek, die das Arbeiten mit RDF-Daten ermöglicht. Dazu stellt sie Methoden zum Parsen, Suchen, Manipulieren und Serialisieren von RDF-Modellen bereit. Sie erlaubt es, RDF-Daten nicht nur in Dateien, sondern auch in SQL-fähigen Datenbanksystemen abzulegen.

Ziel soll es sein, der bestehenden speicherzentrierten SPARQL Implementierung der *RDF API for PHP* einen auf Datenbanken optimierten SPARQL Prozessor zur Seite zu stellen. Durch die Verwendung einer Datenbankabstraktionsschicht soll es möglich sein, jedes beliebige SQL-fähige Datenbanksystem anzusprechen. Die Implementierung soll den kompletten Sprachstandard von SPARQL abdecken und so eine umfassende Interaktion mit den Daten ermöglichen.

Um die Kompatibilität mit SPARQL zu sichern wird der Prozessor vollständig mit automatisierten Tests ausgestattet werden. Weiterhin wird die Geschwindigkeit der hier erstellten Implementierung evaluiert und Vorschläge zur weiteren Optimierung gegeben.

Während der täglichen Arbeit mit SPARQL hat sich herausgestellt, dass sich das Protokoll aufgrund einiger Unzulänglichkeiten noch nicht vollständig für den praktischen Einsatz eignet. Ein weiteres Ziel dieser Arbeit ist es deshalb, Vorschläge zur Implementierung fehlender Funktionalität in SPARQL zu unterbreiten.

1.3 Aufbau

Dieser Einleitung folgend werden in Kapitel 2 die für das Verständnis dieser Arbeit notwendigen Grundlagen vorgestellt. In Kapitel 3 wird der aktuelle

SPARQL-Entwurf näher betrachtet sowie die RDF API for PHP und alternative SPARQL-Prozessoren vorgestellt.

Eine nähergehende Untersuchung des alten speicherbasierten SPARQL-Prozessors wird in Kapitel 4 vorgenommen. Anschließend wird in Kapitel 5 die Implementierung des neuen datenbankgestützten Prozessors beschrieben.

Dieser Prozessor wird in Kapitel 6 mit konkurrierenden State-of-the-Art Lösungen für SPARQL verglichen. Die sich bei der Evaluation ergebenden Verbesserungsmöglichkeiten des Prozessors werden dargelegt. Im Anschluss werden einige Applikationen vorgestellt, die den neuen Datenbankprozessor bereits einsetzen.

Durch den Einsatz des SPARQL-Prozessors in Anwendungen haben sich Erweiterungswünsche zu SPARQL ergeben, die in Kapitel 7 beschrieben werden. Die aus den Projekten destillierten Anforderungen werden auch auf *Conference on Social Semantic Web 2007* vorgestellt und in den *Lecture Notes in Informatics* (LNI) veröffentlicht [WA07].

Am Ende der Arbeit werden die Ergebnisse in Kapitel 8 zusammengefasst und ein Ausblick auf zukünftige Arbeiten gegeben.

Kapitel 2

Grundlagen

In diesem Kapitel werden Grundlagen dargestellt, die für das Verständnis der Arbeit wichtig sind. Dazu gehört das semantische Netz, RDF, URIs und IRIs sowie SQL.

2.1 Das semantische Netz

Der Begründer des World Wide Web, Tim Berners-Lee, beschrieb [BLHL01] das "Semantic Web" zuerst als

> ... an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.

Dieser Satz zeigt deutlich, dass das semantische Netz keine Neuerfindung ist, sondern auf dem existierenden World Wide Web aufbaut und es erweitert. Den vorhandenen, aber nur für Menschen verständlichen Informationen sollen klar strukturierte Daten zur Seite gestellt werden, die von Computerprogrammen verstanden und genutzt werden können.

Das semantische Netz baut auf einer Reihe von Technologien auf, um die Daten in semantisch strukturierter Form verfügbar zu machen. Abbildung 2.1 zeigt, welche Technologien verwendet werden und wie diese aufeinander aufbauen. Im Folgenden werden die für diese Arbeit wichtigen Konzepte näher erläutert.

Das semantische Netz verwendet *Unicode* [Uni96] zur Repräsentation von Zeichen und benutzt *Uniform Resource Identifier* (URI) zur eindeutigen Bezeichnung von physikalischen und abstrakten Ressourcen. Auf diesen baut die *Extensible Markup Language* (XML, [BPSM98]) auf, mit der Dokumente beliebigen Inhaltes nach einem standardisierten Schema notiert

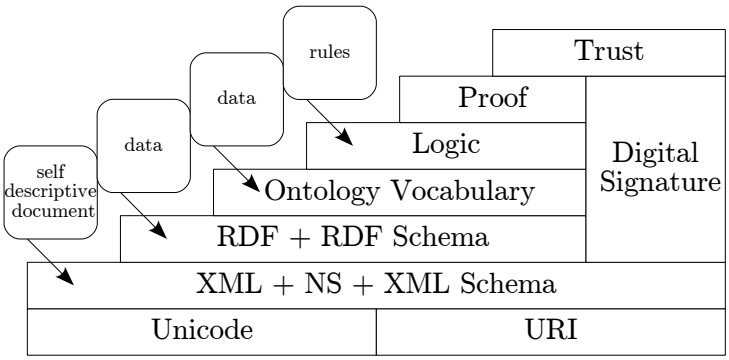


Abbildung 2.1: Das Schichtenmodell des semantischen Netzes

werden können; *XML-Schema* dient zur Festlegung der im Dokumenttyp erlaubten Elemente.

2.2 URI und IRI

Mit einem *Uniform Resource Identifier* (URI, [BLFM05]) kann ein beliebiges Objekt beschrieben werden - sei es wie eine Person in der realen Welt zu finden, oder nicht physisch wie eine Seite im Internet. *Internationalized Resource Identifier* (IRI, [DS05]) erweitert URI durch die Verwendung von in Unicode definierten Zeichen.

URIs können in zwei Gruppen aufgeteilt werden:

- *Uniform Resource Locator* (URL) beschreiben die exakte Position, an der ein Objekt im Netzwerk zu finden ist.
  `http://example.com/index.html` ist ein oft verwendetes Beispiel dafür.

- *Uniform Resource Name* (URN) dient zur eindeutigen Identifizierung von Objekten, die keine direkte Position haben. Ein Beispiel wäre `urn:isbn:3-423-084-774`, die ein Buch durch seine ISB-Nummer repräsentiert. Der URN verweist auf das Buch, ohne jedoch anzugeben, an welche Stelle es zu finden ist.

Der Name *Uniform Resource Identifier* gibt bereits an, wodurch sich URIs auszeichnen:

- *Uniform*: Alle URIs sind nach einem festen Schema aufgebaut.

- *Resource*: Alle möglichen "Ressourcen" können durch URIs beschrieben werden; es gibt keine Einschränkungen.

- *Identifier*: Durch die URI sind alle benötigten Informationen gegeben, die man benötigt, um es von anderen zu unterscheiden.

In RFC 3986 [BLFM05] wird der allgemeine Aufbau von URIs beschrieben. Sie sind hierarchisch von links nach rechts aufgebaut. Neben dem Schema und dem darauf folgenden Doppelpunkt, mit denen jede URI beginnen muss, sind nur noch der Schrägstrich `/`, das Fragezeichen `?` und die Raute `#` als Begrenzungszeichen festgelegt. Die Festlegung aller weiteren Details wird weiterführenden Spezifikationen, wie zum Beispiel der RFC 2616 [FGM+99] für HTTP-URIs, überlassen.

Die Schemanamen werden von der Internet Assigned Numbers Authority (IANA) vergeben und umfassen zum Beispiel `http`, `ftp` und `mailto`.

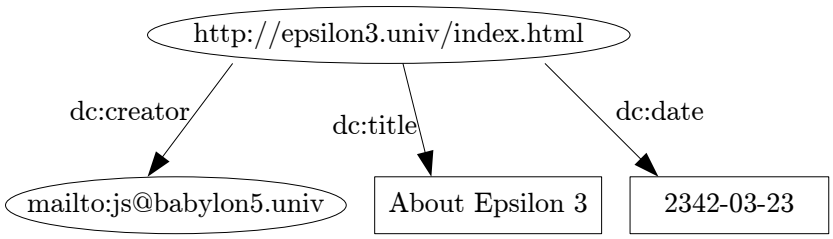
2.3 RDF

Mit Hilfe des *Resource Description Framework* (RDF, [MM02]) ist es möglich, Daten über Daten - Metadaten - auszudrücken. Dies kann zum Beispiel dazu benutzt werden, den Autor und Titel eines HTML-Dokumentes maschinenlesbar abzulegen. Mit RDF können beliebige Zusammenhänge beschrieben werden, ohne diese vorher semantisch einzuordnen. Beispiele für Metadatenschemata sind *Dublin Core* [PNNJ05] und die *Platform for Internet Content Selection* (PICS, [RM96]).

Informationen in RDF werden in Sätzen notiert; diese bestehen aus Subjekt, Prädikat und Objekt. Ein Subjekt kann eine *Blank Node* oder eine *Ressource* sein, ein Literal ist immer eine *Ressource*. Ein Objekt kann zusätzlich zu den schon im Subjekt erlaubten *Blank Node* und *Ressource* auch ein einfaches *Literal* sein.

Eine *Ressource* ist ein durch eine IRI (*Internationalized Resource Identifier*) beschriebenes physikalisches oder abstraktes Objekt. Objekte ohne Namen werden als *Blank Node* bezeichnet. *Literale* sind Zeichenketten eines bestimmten Datentyps, die zusätzlich mit einem Sprachetikett ausgezeichnet sein können.

Während RDF für Menschen gut mit einem Graphen wie in Abbildung 2.2 visualisiert werden kann, existieren für Computer verschiedene Serialisierungsformate wie *RDF/XML* [Bec04b], *Notation 3* [BL05] und *Turtle* [Bec04a].



Gemäß den Konventionen sind Ressourcen elliptisch, Literale rechteckig und Prädikate als Pfeile dargestellt. Das Präfix `dc:` ist die Abkürzung von `http://purl.org/dc/elements/1.1/`.

Abbildung 2.2: Darstellung von RDF als Graph

2.4 SQL

Die inzwischen zwanzig Jahre alte *Structured Query Language* (SQL, [Ins89]) ist die am weitesten verbreitete und durch Programme unterstützte Anfrage- und Manipulationssprache für relationale Datenbanken. Sie wird von fast allen Datenbanksystemen unterstützt und ist von den Organisationen ANSI und ISO standardisiert.

Die Sprache lehnt sich sehr an das Englische an. Eine typische Abfrage ist

```
‘‘Select name from persons where age > 40’’
```

Aus der satzähnlichen Notation kann der Inhalt der Anfrage sehr gut erkannt werden.

Innerhalb einer relationalen Datenbank werden die Daten in Tabellen gespeichert, die selbst wiederum beliebig viele Spalten eines ausgezeichneten Datentyps enthalten können. Anfragen können sowohl auf einer Tabelle durchgeführt werden, aber auch mehrere Tabellen überspannen. SQL stellt Sprachelemente bereit, um bestimmte Spalten auszuwählen, diese Daten zu filtern und mit anderen Anfrageresultaten zu kombinieren.

Aufgrund der weiten Verbreitung und des breiten Funktionsspektrums sind SQL-Datenbanken ein bevorzugter Weg, auch RDF-Daten abzuspeichern. Der während dieser Diplomarbeit erstellte SPARQL Prozessor fungiert als Schnittstelle zwischen in SPARQL gestellte Anfragen und den in SQL-Datenbanken gespeicherten Daten.

Kapitel 3

Stand der Technik

Aufbauend auf den im letzten Kapitel vorgestellten Grundlagen werden in diesem SPARQL, die RDF API for PHP und alternative Bibliotheken mit Unterstützung für SPARQL vorgestellt.

3.1 SPARQL

SPARQL, die "SPARQL Protocol And Query Language" ist die bedeutendste Abfragesprache für RDF-Daten. Sie wird in der Data Access Working Group des World Wide Web Consortiums entwickelt und befindet sich zur Zeit im Status der "Candidate Recommendation" ([PS06]), was bedeutet, dass der Standardisierungsprozess fast abgeschlossen ist. Die Sprache wurde von Eric Prud'hommeaux und Andy Seaborne erdacht und setzt die mit vorher entworfenen Abfragesprachen wie SquishQL [MSR02] und RDQL [Sea04] gewonnenen Erfahrungen und Ideen um.

Die SPARQL Protocol And Query Language ist dafür gedacht, Tripeldaten aus RDF-Datenbanken abzufragen. Dafür stellt sie verschiedene Funktionen bereit, die die Einschränkung und Kombination der Tripel ermöglichen.

3.1.1 Syntax

Die Syntax von SPARQL ist der von SQL ähnlich; die bekannten Klauseln `SELECT`, `FROM` und `WHERE` kommen auch hier zum Einsatz.

Eine Anfrage beginnt mit der Definition von Präfixen. Präfixe sind Abkürzungen für IRIs, die in der Anfrage zur Notation von Ressourcen verwendet werden. Auf diese Weise können die Abfragen für den Menschen übersichtlicher und lesbarer gestaltet werden. Mit dem im Beispiel 3.1 de-

finierten Präfix kann innerhalb der Anfrage `foaf:` anstatt der IRI benutzt werden.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

Abbildung 3.1: Definition eines Präfixes

Nach der Definition der Präfixe wird der Typ der Anfrage festgelegt. SPARQL definiert vier Anfrage- und Ergebnistypen:

- `SELECT` wählt die in der Anfrage festgelegten Variablen aus und gibt sie zurück.
- `CONSTRUCT` erstellt komplette RDF-Graphen mit Hilfe einer in der Anfrage definierten Vorlage.
- `DESCRIBE` gibt Beschreibungen der Ressourcen und Variablen zurück und
- `ASK` beantwortet die Anfrage mit Ja/Nein, je nachdem, ob es Ergebnistripel gibt oder nicht.

Das Finden von Ergebnissen erfolgt bei SPARQL durch die Suche nach übereinstimmenden *Graph Patterns*, Kombinationen von einem oder mehreren Tripeln. Teile der Tripel können durch Variablen besetzt sein, die im zurückgegebenen Ergebnis durch die gefundenen Werte ersetzt werden. Beispiel 3.2 demonstriert dies.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name, ?mbox
WHERE {
 ?x foaf:name ?name. ?x foaf:mbox ?mbox
}
```

?name	**?mbox**
John Sheridan	js@babylon5.univ
Susan Ivanova	si@babylon5.univ

Abbildung 3.2: Benutzung von Variablen in einem Graph Pattern

Die Auswahl der Tripel kann weiter durch Beschränkungen auf erlaubte Werte und Wertebereiche eingegrenzt werden. Dies ist durch das Schlüs-

selwort `FILTER` möglich: Die selektierten Tripel müssen den durch die Filterregeln definierten Einschränkungen genügen. Zu den von SPARQL bereitgestellten Funktionen gehören unter anderem `regex()`, `datatype()` und `isBound()`. Beispiel 3.3 gibt die Namen derjenigen Personen aus, die älter als 23 Jahre sind und deren Emailadresse auf `@babylon5.univ` endet.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
 ?x foaf:name ?name. ?x foaf:mbox ?mbox.
 ?x foaf:age ?age
 FILTER (?age > 23 && regex(?mbox, "@babylon5\.univ$"))
}
```

Abbildung 3.3: Einschränkung durch Filter

Da die verfügbaren RDF-Daten oftmals nur einen Bruchteil der möglichen Informationen darstellen, sind Lücken unvermeidlich. Um aber zum Beispiel Personen mit und ohne Emailadressen gleichzeitig abfragen zu können, stellt SPARQL das `OPTIONAL`-Konstrukt bereit. Hierdurch können Teildaten, die nicht überall vorhanden sind, mit in die Ergebnismenge einbezogen werden. Dies ist in Beispiel 3.4 zu sehen.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name, ?mbox
WHERE {
 ?x foaf:name ?name.
 OPTIONAL {
  ?x foaf:mbox ?mbox
 }
}
```

?name	**?mbox**
John Sheridan	js@babylon5.univ
Alfred Bester	

Abbildung 3.4: Abfrage unvollständiger Datenbestände

Mehrere Abfragen können mit Hilfe des `UNION`-Konstrukts in einer Ergebnismenge vereint werden. In Beispiel 3.5 werden exemplarisch alle Vor-

und Nachnamen einzeln ausgegeben.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
 {?x foaf:givenname ?name}
 UNION {?x foaf:family_name ?name}
}
```

?name
John
Susan
Sheridan
Ivanova

Abbildung 3.5: Abfrage unvollständiger Datenbestände

Zu guter Letzt kann die Anzahl der Ergebnisse mit dem Schlüsselwort `LIMIT` eingeschränkt werden, und das Fenster mit Hilfe von `OFFSET` verschoben werden. Beispiel 3.6 gibt die Zeilen 2 bis 12 der Ergebnismenge zurück.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE { ?x foaf:name ?name }
LIMIT 10 OFFSET 2
```

Abbildung 3.6: Beschränkung der Ergebniszahl

3.1.2 Zugehörige Technologien

Neben der eigentlichen Syntax und Semantik der Anfragesprache definiert das World Wide Web Consortium noch weitere Technologien, die zusammen mit SPARQL benutzt werden können.

Das *SPARQL Protocol for RDF* [Cla06] spezifiziert zum einen ein generisches Protokoll, um SPARQL-Anfragen von Clients an Server zu senden. Als zweites definiert die Recommendation die konkrete Syntax für die Implementierung des Protokolls auf Basis des HyperText Transfer Protocol (HTTP, [FGM+99]) und des Simple Object Access Protocol (SOAP,

[BEK+00]). Durch dieses Protokoll wird es möglich, dass die SPARQL Implementierungen verschiedener Hersteller nahtlos und ohne Probleme miteinander kommunizieren können.

Als Gegenstück zum definierten Anfrageprotokoll definiert das *SPARQL Query Results XML Format* [BB06] wie die Antwort einer Anfrage eines Clients an einen SPARQL-Server auszusehen hat. Während die Daten auch als Notation 3 oder RDF/XML ausgedrückt werden können, beinhaltet das SPARQL Query Results XML Format explizit die Möglichkeit, zurückgegebene Daten an die in der Anfrage festgelegten Variablen zu binden.

3.1.3 Historische Entwicklung

Im Zuge der Entstehung des semantischen Netzes gab es viele Versuche, den Bedarf nach einer für RDF geeigneten Anfragesprache zu erfüllen. Die meisten dieser frühen Entwicklungen wurden im Laufe der Zeit zugunsten von durchdachteren und leistungsstärkeren Sprachspezifikationen verworfen.

Die wohl ältesten Entwürfe einer solchen Abfragesprache stellen die von Libby Miller, Andy Seaborne und Alberto Reggior erdachte SquishQL [MSR02] sowie die am Forth Institue of Computer Science entwickelte RDF Query Language (RQL, [Kar01]) dar. Beide zeigen schon deutlich die Tendenz, die Syntax von RDF-Abfragesprachen ähnlich der Structured Query Language (SQL) zu gestalten.

Die von Andy Seaborne entwickelte RDF Data Query Language (RDQL, [Sea04]) vereint bereits die Ideen dieser beiden und einiger anderer in [Pru04] verglichenen Sprachentwicklungen. Im Gegensatz zu der daraus hervorgegangenen SPARQL Protocol And Query Language fehlen RDQL die Möglichkeit, alternative und optionale Suchausdrücke anzugeben sowie mehrere RDF-Graphen gleichzeitig abzufragen.

Eine andere Richtung schlagen Ansätze ein, die bestehende und auf XML basierende Anfragesprachen wie XPath [CD99] und XQuery [BCF+07] zu erweitern versuchen. Die von der XUL [Moz01] verwendete Syntax sowie die kommerziell entwickelte und an XPath angelehnte Sprache Versa [OO02] gehören dazu.

3.2 RDF API for PHP

Die *RDF API for PHP* (RAP, [Biz04]) ist eine in *PHP: Hypertext Preprocessor* (PHP) geschriebene Open-Source-Klassenbibliothek zur Arbeit mit RDF-Daten. Sie wurde im September 2002 von Chris Bizer in der ersten Version veröffentlicht und hat durch die Arbeit der inzwischen neun Ent-

wickler eine hohe Funktionalität und Stabilität erreicht. Mit ihr können die in Formaten wie RDF/XML [Bec04b] und Notation 3 [BL05] gespeicherten Daten geladen und wieder serialisiert werden. Weiterhin stellt die Bibliothek Methoden zur Abfrage und Manipulation der Daten bereit. Sie bietet auch Möglichkeiten, diese Daten in Netzwerken zu veröffentlichen.

Die verwendete Programmiersprache PHP ist eine der ältesten und die am weitesten verbreitete serverseitige Skriptsprache für Anwendungen im Internet. Sie kann und wird zur Erstellung einfachster, in HTML eingebetteter Gästebücher bis hin zur Implementierung komplexen Klassenbibliotheken und Applikationen benutzt. Seit Version 5 hat PHP ein Klassen- und Objektsystem, welches mit denen anderer Programmiersprachen wie Java und C++ vergleichbar ist.

Die RDF API for PHP besitzt bereits zwei Methoden, um die Daten von RDF-Modellen abzufragen; RDQL und SPARQL. Beide sind jedoch dafür optimiert, auf im Speicher gehaltenen RDF-Modellen zu arbeiten und unterstützen die Arbeit mit in SQL-Datenbanken vorgehaltenen Informationen nur sehr unperformant.

Die Anbindung der API an die Datenbanken erfolgt über eine Zwischenschicht, die von den Details der einzelnen SQL-Clientbibliotheken abstrahiert. Durch die Verwendung von *ADODB* [1] kann RAP mit allen von PHP unterstützten Datenbankservern zusammenarbeiten.

3.3 Alternative Bibliotheken und Werkzeuge

Wie bei vielen Themen gibt es auch bei den Bibliotheken zur Arbeit mit RDF eine Vielzahl an Implementierungen. Die Wichtigsten sind hier aufgeführt.

Eine nahezu vollständige Liste kann unter `http://esw.w3.org/topic/SemanticWebTools` eingesehen werden.

3.3.1 ARC

ARC [2] ist ein leichtgewichtiger und in PHP geschriebener SPARQL-Server, der direkt auf einer MySQL-Datenbank arbeitet. Zielsetzung des Projektes ist es, eine performante Implementierung für alltägliche Webapplikationen

[1] `http://adodb.sourceforge.net/`
[2] `http://arc.web-semantics.org/`

bereitzustellen. ARC bietet nur einen Bruchteil der in RAP zur Verfügung stehenden Funktionalität.

Anders als viele weitere Bibliotheken setzt ARC allein auf MySQL und bietet keine Unterstützung für andere Datenbanksysteme.

Die Bibliothek wird von der kleinen Softwareagentur *semsol* [3] entwickelt; die Community ist sehr klein.

3.3.2 Redland RDF libraries / librdf

librdf [4] ist eine Bibliothek zur Arbeit mit RDF-Daten, die eine lange Entwicklungszeit hinter sich hat und als robust und stabil gilt. Sie bietet Unterstützung für Modelle im Speicher wie in Datenbanken. In C geschrieben, kann sie durch die Bereitstellung von Anbindungen an die verschiedensten Programmiersprachen auch von PHP aus benutzt werden.

Redland wurde bis 2005 an der University of Bristol entwickelt, ist seitdem jedoch in alleiniger Hand von Dave Beckett, Entwickler mehrerer RDF-bezogener Technologien wie RDF/XML [Bec04b], SPARQL Query Result XML Format [BB06] und Turtle [Bec04a].

3.3.3 Jena

Das *Jena Semantic Web Framework* [5] eine in Java programmierte Bibliothek zur RDF-Verarbeitung. Sie bietet ähnliche Features wie die RAP und Redland. Jena wird aktiv vom *HP Labs Semantic Web Programme* [6] weiterentwickelt und besitzt eine große Community.

3.3.4 Sesame

Ein weiteres auf Java basierendes Framework für die Arbeit mit RDF Daten ist *Sesame* [7]. Sesame bietet sowohl Bibliotheken zur Verwendung in eigenen Clientapplikationen wie auch einen Server, auf den über HTTP oder die Sesame Repository API zugegriffen werden kann.

Wie auch die vorherigen Bibliotheken kann es seine Daten in relationalen Datenbanken ablegen, unterstützt allerdings nur eine eingeschränkte Anzahl an Datenbankmanagementsystemen.

---

[3] `http://www.appmosphere.com/`
[4] `http://librdf.org/`
[5] `http://jena.sourceforge.net/`
[6] `http://www.hpl.hp.com/semweb/`
[7] `http://www.openrdf.org`

Das Open-Source-Projekt wird von der Firma *Aduna* [8] entwickelt, die für die Software kommerziellen Support anbietet.

3.3.5 Virtuoso

Der *Virtuoso Universal Server* [9] ist, anders als die bisher vorgestellten Bibliotheken, keine reine Bibliothek und Lösung zur Arbeit mit RDF. Virtuoso ist ein "universeller" Server, der der einzige Punkt für die Speicherung jeglicher Daten sein soll. Um dies zu ermöglichen, implementiert der Server verschiedene Protokolle und Anfragesprachen, wie zum Beispiel WebDAV, SOAP, NNTP, SQL und XSLT. Die Verwendung der Datenbank von Applikationen, die auf RDF-Daten arbeiten, ist durch die Unterstützung von SPARQL und dem SPARQL Protocol möglich.

Zusätzlich zur offiziellen SPARQL-Recommendation implementiert der von Virtuoso bereitgestellter SPARQL-Prozessor auch einfache Gruppierung und Aggregation. Dies ist eine proprietäre Erweiterung und leider auch mangelhaft umgesetzt, wie in 7.6.1 näher erläutert.

Virtuoso wird von der Firma *OpenLink Software* [10] als kommerzielles Produkt entwickelt und vertrieben, steht jedoch in eingeschränkter Form als Open-Source-Projekt zur Verfügung.

[8] `http://www.aduna-software.com`
[9] `http://sourceforge.net/projects/virtuoso/`
[10] `http://openlinksw.com/`

Kapitel 4

Der speicherbasierte SPARQL-Prozessor

Die im letzten Kapitel vorgestellte RDF API for PHP besitzt einen eigenen Prozessor, um SPARQL-Anfragen zu bearbeiten. Dieser wird hier genauer analysiert.

4.1 Überblick

Die RDF API for PHP erhielt mit der am 16.01.2006 veröffentlichten Version 0.9.3 Unterstützung für die SPARQL Query Language und Teile des SPARQL-Protokolls. Diese Erweiterung wurde von Tobias Gauß im Rahmen seiner Diplomarbeit [Gau06] implementiert und setzt die SPARQL-Recommendation auf dem Stand von Ende 2005 um.

Ziel der Arbeit war es, die Abfrage von beliebigen Modellen und RDF-Datenmengen mittels SPARQL zu ermöglichen. Die RAP erlaubt es, Modelle aus den verschiedensten Datenformaten zu erstellen. RDF-Dateien in Form von RDF/XML, Notation 3 oder auch RSS-Feeds können sowohl in speicherbasierten Modellen (`MemModel`) als auch in Datenbanken (`DbModel`) abgelegt werden. Der SPARQL-Prozessor verwendet ausschließlich speicherbasierte Modelle, um die Antwortdaten für Anfragen zu errechnen. In der Datenbank gespeicherte Daten werden in ein Modell in den Hauptspeicher geladen.

Das SPARQL-Paket besteht aus zwei Komponenten: Dem SPARQL-Parser, und dem eigentlichen SPARQL-Prozessor `SparqlEngine`. Die beiden Klassen `Model` und `Dataset` haben jeweils eine Methode
`sparqlQuery(string $query [, string $resultForm = false])`
die zum Ausführen einer SPARQL Abfrage aufgerufen werden muss.

Zuerst wird der übergebene Anfragestring an den Parser weitergereicht, der die Anfrage in ein Objekt umwandelt. Dieses wird dann an einen Instanz des Prozessors übergegeben, der die eigentliche Berechnung ausführt. Der zweite Parameter von `sparqlQuery` entscheidet darüber, ob das Ergebnis als PHP-Array oder als XML-Dokument im SPARQL Query XML Format zurückgegeben wird.

4.2 Der SPARQL-Parser

Der für das Umwandeln des Anfragestrings in ein im Speicher gehaltenes `Query`-Objekt zuständige Parser arbeitet tokenbasiert. Das heißt, dass die Anfragezeichenkette in Teilstücke, so genannte Tokens, zerlegt wird. Eine Schleife iteriert nachfolgend über diese Teile und ruft je nach Kontext die passende Funktion zur Erstellung von Unterobjekten des Abfrageobjekts auf.

Die `FILTER`-Abschnitte der Anfragen werden nicht geparst, sondern als reine Zeichenketten im Anfrageobjekt behalten. Die dahinterstehende Idee war es, einen Filterausdruck als Befehl anzusehen, der nach einigen Modifikationen durch die PHP-interne Funktion `eval(string $command)` ausgeführt werden kann.

4.3 Die Klasse SparqlEngine

Das eigentliche Arbeitstier des SPARQL-Paketes ist der Prozessor in der Klasse `SparqlEngine`. In ihr werden aus den Modelldaten die Rückgabewerte nach den durch die Abfrage definierten Einschränkungen generiert.

Zuerst werden verschiedene Graphlisten aufgebaut, die den Tripeln in der Abfrage entsprechen. Diese Listen werden im Anschluss nach den Vorgaben der Anfrage miteinander verbunden und anschließend gefiltert.

An dieser Stelle liegen die zurückzugebenden Daten als PHP Arrays vor und müssen eventuell noch in XML umgewandelt werden. Dies passiert klassenintern durch eine spezielle Konverterfunktion.

4.4 Probleme des Prozessors

Durch den Anspruch des Prozessors, alle Datenmodelle der RAP unterstützen zu wollen, mussten Kompromisse hinsichtlich der Speicherverwendung gemacht werden. Die Implementierung der Filter ist durch die Verwendung

von `eval` kompakt und einfach gehalten, eröffnet aber ein großes Sicherheitsproblem. Die fixe Kodierung der Rückgabeformate schränkt die Flexibilität des Prozessors beim Einsatz in anderen Projekten stark ein.

Im Nachfolgenden werden diese Probleme genauer beschrieben.

4.4.1 Überproportional wachsender Speicherbedarf

Das Verbinden der Graphlisten zur Berechnung der Ergebnisse erfolgt im Hauptspeicher. Eine Liste mit zehn Tripeln kann, wenn sie mit einer gleich großen Liste verbunden wird, im schlechtesten Fall eine Größe von $10 \times 10 = 100$ Doppeltripeln erreichen. Bei komplexeren Anfragen und größeren Modellen wächst der benötigte Hauptspeicher stark an, so dass die Anforderungen an den RAM für größere Datenmengen und nichttriviale Anfragen kaum zu bewältigen sind.

Die in Kapitel 6 durchgeführten Benchmarks zeigen, dass es mit der verwendeten Hauptspeichergröße von 768 MiB nicht möglich ist den speicherbasierten Prozessor für Abfragen von über 10.000 oder mehr Tripeln zu verwenden. Diese Einschränkung tritt bei einem einzelnen Benutzer auf. Bei der Verwendung des Prozessors auf einem Server im Mehrbenutzerbetrieb mit vielen gleichzeitigen Anfragen sind die verwendbaren Datenmengen noch um einiges kleiner.

4.4.2 Sicherheitsproblem bei der FILTER-Behandlung

Die Auswertung von Filterausdrücken erfolgt im Prozessor in zwei Schritten:

1. Der aus dem Anfragestring extrahierte FILTER-Ausdruck wird mit regulären Ausdrücken so umgeformt, dass valider PHP-Code entsteht. Dies wird erreicht, indem Referenzen auf Variablen in SPARQL wie `?name` durch die Namen von PHP-Variablen aus der entsprechenden Graphliste ersetzt werden.

2. Die entstandene Zeichenkette wird an die PHP-interne Funktion `eval` übergeben und als normaler PHP-Code ausgeführt. Der Rückgabewert wird in eine Variable geschrieben, deren Inhalt nach der Ausführung bestimmt, ob das Tripel der Graphliste den Filter erfüllt.

Funktionen wie `regex` werden direkt an PHP weitergeleitet. Durch eine fehlende - und mit dem Parser auch nicht durchführbare - Überprüfung der Funktionsnamen ist es möglich, beliebige Funktionen mit eigenen Parametern aufzurufen. Eine harmlose Anfrage könnte zum Beispiel

```
print(string $ausgabe)
```

mit `Hallo Welt!` als Parameter ausführen. Es sind aber auch Funktionen wie `unlink` (Löschen von Dateien) oder `eval` selbst in Verbindung mit `fopen` oder `file_get_contents` denkbar. Mit der von PHP bereitgestellten Netzwerktransparenz kann damit Schadcode von fremden Servern nachgeladen und ausgeführt werden.

Durch das nicht durchgeführte Parsen der FILTER-Ausdrücke ist es leider auch nicht möglich, eine Prüfung der aufzurufenden Funktionen durchzuführen. Ein Lösungsansatz wäre es, den Filter auf bekannte "Problemfunktionen" wie `unlink` zu durchsuchen. Es ist allerdings sehr schwer, den korrekten Einsatz innerhalb von Parametern - zum Beispiel von `regex` - von einer bösartigen Verwendung zu unterscheiden. Weiterhin kann man die Funktionsnamen in PHP selbst aus mehreren Variablen zusammensetzen, was noch schwerer zu erfassen ist.

Die Möglichkeiten, solch triviale Sicherheitsfilter zu umgehen, sind zahllos. Es gibt nur eine Möglichkeit, eine sichere und nicht umgehbare Überprüfung der Funktionen zu implementieren: Das vollständige Parsen der FILTER-Ausdrücke in einen Objektbaum, wie es schon für den Rest der SPARQL-Anfrage gemacht wird. In diesem können die Funktionsnamen direkt überprüft werden, ohne Probleme mit deren validen Verwendung in Parametern zu bekommen.

4.4.3 Fest kodierte Rückgabeformate

`SparqlEngine` unterstützt zwei verschiedene Formate für Rückgabewerte: PHP-Arrays und das SPARQL XML Result Format. PHP-Arrays sind das im Abfrageprozess intern verwendete Datenformat. Für dessen Erstellung muss keinerlei Konvertierung durchgeführt werden; es wird keine zusätzliche Rechenzeit benötigt. Das XML-Format wird durch eine Konverterfunktion aus den Arrays generiert, wenn der `resultForm`-Parameter entsprechend gesetzt ist.

Die Aktivierung der XML-Konvertierung ist fest in der Engine einprogrammiert und kann ohne Änderung des Codes nicht erweitert werden, um zum Beispiel JSON [Cro06] als Rückgabeformat anzubieten. Weiterhin ist die Konvertierungsfunktion selbst in die Engine integriert, was eine klare Verletzung des Model-View-Controller Konzeptes (MVC, [Ree79]) und der Kapselung von Funktionen darstellt.

Viele Anwendungen benötigen die Rückgabewerte in einem speziellen Format oder andere Klassen für die Objekte im PHP-Array. *Ontowiki* [ADR06] hat zum Beispiel eine eigene RDFS-API mit angepassten Klassen für Literale, Ressourcen und Blank Nodes. Web 2.0-Applikationen benötigen das Ergebnis nicht als Array, sondern als serialisiertes Javascript (JSON).

All diese Formate sind nur mit einer zusätzlichen Programmschicht auf Seite der Applikation zu realisieren. Dies benötigt zusätzliche Rechenzeit und muss von jedem Projekt neu programmiert werden. Es sollte möglich sein, eigene Ergebnis"renderer" einzubinden, die vom Prozessor benutzt werden, um das gewünschte Format direkt zu liefern.

Kapitel 5

Implementierung des datenbankbasierten SPARQL-Prozessors

Aufbauend auf dem im vorherigen Kapitel näher beleuchteten speicherbasierten SPARQL-Prozessors wird ein neuer, datenbankbasierter Prozessor entwickelt. Dieser wird keine der aufgezeigten Nachteile des alten Prozessors haben.

5.1 Anforderungen

Die Anforderungsanalyse zum neuen Prozessor ergab eine Reihe an Punkten, die erfüllt werden müssen:

- Um die Umstellung von Applikationen so einfach wie möglich zu machen, soll sich der Prozessor nahtlos in die existierende Klassenlandschaft integrieren. Im Idealfall wird der neue Prozessor automatisch verwendet, ohne dass die Applikation angepasst werden muss.

- Der datenbankbasierte Prozessor soll so viele existierende Klassen und Code wie möglich wiederverwenden. Dies betrifft unter anderem den bereits existierenden SPARQL-Parser und die Klassen, mit denen die geparste Anfrage im Speicher repräsentiert wird.

- Parameter und Rückgabewerte der öffentlich sichtbaren Methoden haben das gleiche Format wie die des existierenden Prozessors.

- Der Prozessor soll so wenig Datenlogik wie möglich enthalten und nur dazu dienen, die SPARQL-Anfrage in SQL umzuwandeln. Die eigentliche Arbeit muss auf den SQL-Server ausgelagert werden.
- Die Möglichkeit zur Unterstützung neuer Rückgabeformate soll geschaffen werden. Dies hat so zu geschehen, dass der Code des Prozessors für neue Formate nicht geändert werden muss.

5.2 Implementierung

Da der neue Prozessor unabhängig vom alten arbeiten soll, bekommt er den Klassennamen `SparqlEngineDb` und lebt fortan in der Datei `api/sparql/SparqlEngineDb.php`.
Alle zugehörigen Klassen werden im Ordner `api/sparql/SparqlEngineDb/` aufbewahrt. Den Klassennamen wird das Präfix `SparqlEngineDb_` vorangestellt.

Für das Absetzen von SPARQL-Anfragen wurde bisher im entsprechenden Modell oder Dataset die Methode
`sparqlQuery(string $query [, string $resultForm = false])`
aufgerufen. Dieser Mechanismus wird beibehalten. Um den neuen Prozessor optimal zu integrieren, bekommt der speicherbasierte Prozessor eine *factory*-Methode, die abhängig vom verwendeten Modell/Dataset den korrekten Prozessor instantiiert. Beim Aufruf von `sparqlQuery` auf einem Modell der Klasse `DbModel` wird der neue datenbankbasierte, ansonsten der alte speicherbasierte Prozessor verwendet. Mit diesem Mechanismus bekommt die anfragende Applikation immer die passende und beste Möglichkeit, die RDF-Daten abzufragen.

5.2.1 Klassenhierarchie

Dreh- und Angelpunkt ist die Klasse `SparqlEngineDb` selbst. Sie nimmt `Query`-Objekte via `queryModel()` an und verwendet die Generatorklassen `SparqlEngineDb_SqlGenerator` und `SparqlEngineDb_FilterGenerator` zur Erstellung der SQL-Anfrage.

`SparqlEngineDb_TypeSorter` und `SparqlEngineDb_Offsetter` werden benötigt um eventuell - basierend auf der generierten SQL-Anfrage - mehrere Teilanfragen zu erstellen, die unterschiedliche Teile des Ergebnisses zurückliefern. `SparqlEngineDb_SqlMerger` fügt mehrere Teilanfragen in einem UNION-Konstrukt zusammen.

Die Ergebnisse der SQL-Anfragen an den Server stehen als ADODB Record Sets zur Verfügung und müssen in ein Zielformat umgewandelt werden.

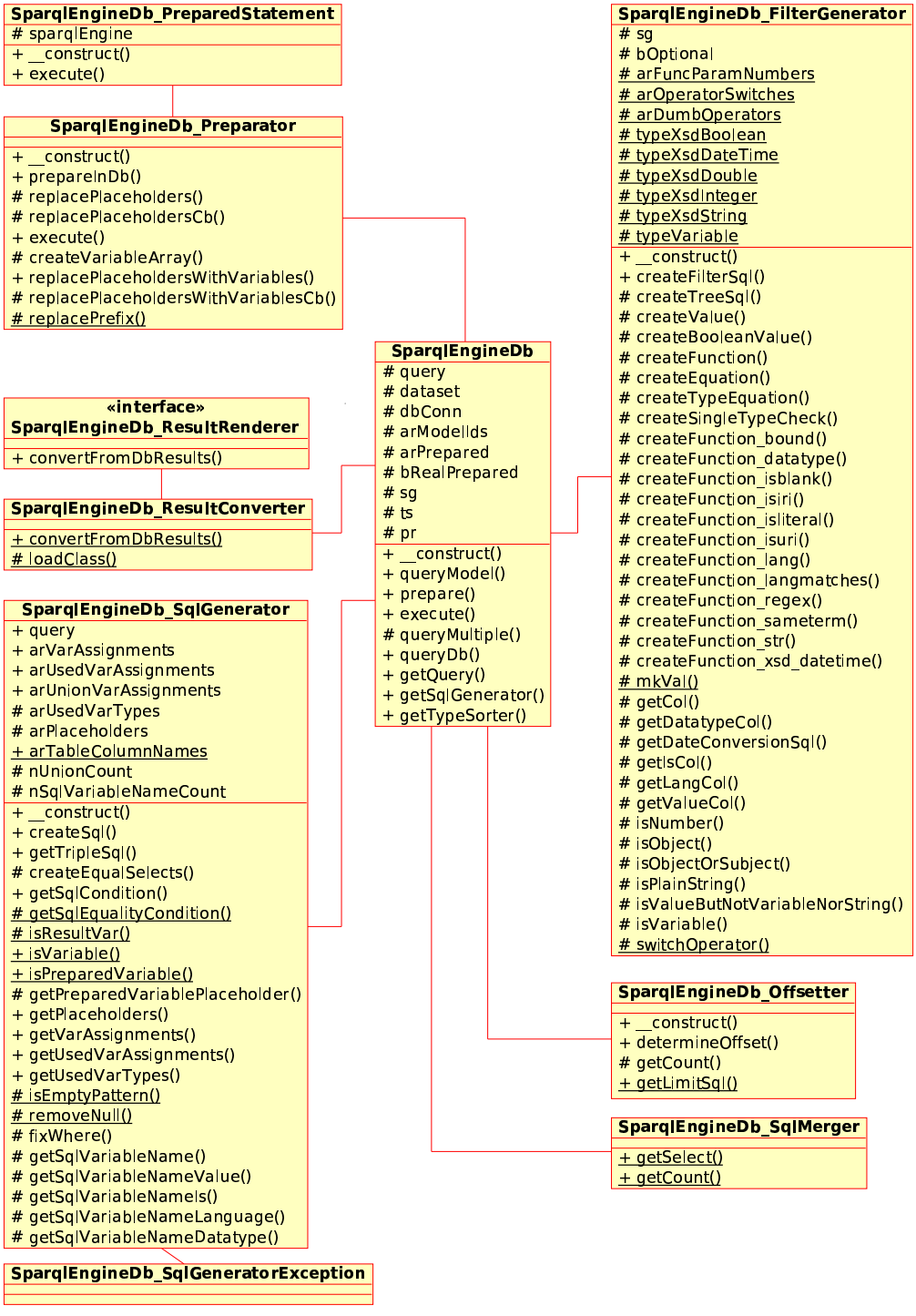


Diagramm: Klassendiagramm Seite 1

Abbildung 5.1: Klassendiagramm SparqlEngineDb

Die Klasse `SparqlEngineDb` übergibt die Datenbanksätze an den `SparqlEngineDb_ResultConverter`. Dieser ruft abhängig vom gewünschten Format den passenden Ergebnisrenderer auf, welcher das Interface `SparqlEngineDb_ResultRenderer` implementieren muss.

Für die Unterstützung von Prepared Statements ist der `SparqlEngineDb_Preparator` zuständig, der die vom Prozessor zurückgegebenen `SparqlEngineDb_PreparedStatement`s verarbeitet.

5.2.2 Die Klasse SparqlEngineDb

Die Klasse `SparqlEngineDb` ist die Hauptklasse des Pakets. Sie ist für die Koordination der Subklassen zuständig und wird vom Benutzer als einzige Klasse direkt angesprochen.

Der Prozessor unterstützt zur Instantiierung in `__construct($model, $arModelIds = null)` die Übergabe eines Modells oder Datasets. Weiter kann auch ein `DbStore` mit einem optionalem Array an Modell-IDs übergeben werden, auf denen die Abfrage ausgeführt werden soll.

Diese modellunabhängige Unterstützung für komplette Datenbanken ist notwendig, wenn Applikationen nicht nur auf einzelnen Modellen arbeiten müssen. Ontowiki greift zum Beispiel gleichzeitig neben den "Daten"-Ontologien auch noch die Systemontologien ab. Das mehrmalige Ausführen einer Anfrage auf verschiedenen Modellen ist prinzipiell möglich, kostet aber zum einen abhängig von der Anzahl der abzufragenden Modelle linear mehr Zeit, und zum anderen sind modellübergreifende Abfragen nicht oder nur mit zusätzlichem clientseitigen Rechenaufwand möglich.

`queryModel($dataset, Query $query, $resultform = false)` ist die Hauptfunktion des Prozessors. Ihr wird das Dataset, die Anfrage an sich und das gewünschte Ergebnisformat übergeben. Zuerst erstellt sie die intern benötigten Generator- und Ergebniskonverterobjekte. Danach wird die SQL-Anfrage mit Hilfe des `SparqlEngineDb_SqlGenerator` erstellt und der `SparqlEngineDb_TypeSorter` zur Teilung der SQL-Anfrage auf eventuell mehrere Anfragen benutzt. Die erstellten Anfragen werden mit `queryMultiple` auf der Datenbank ausgeführt und dem `SparqlEngineDb_ResultConverter` übergeben, der die Daten im gewünschten Format zurückliefert. Die Erfordernis des Teilens einer SQL-Anfrage in mehrere Anfragen wird in Abschnitt 5.2.4 erklärt.

`prepare(Dataset $dataset, Query $query)` macht fast dasselbe wie `queryModel` mit dem Unterschied, dass nur das SQL-Statement erstellt wird. Im Fall, dass die Anfrage *datenunabhängig* ist - keine `ORDER BY`-Klauseln mit Sortiervariablen enthält - werden auch noch mehrere Anfragen, die eventu-

ell für sortierte Daten notwendig sind, erstellt. Die Methode gibt ein Objekt der Klasse `SparqlEngineDb_PreparedStatement` zurück, dessen Methode `execute($arVariables, $resultform = false)` der Benutzer dann zum Ausführen des Statements verwenden kann. Details zur *Datenunabhängigkeit* von Anfragen sind in Abschnitt 5.2.4 zu finden.

Ein Prepared Statement hält intern eine Referenz auf den SPARQL-Prozessor, um beim Aufruf von `execute` die namentlich gleich benannte Funktion der `SparqlEngineDb` auszuführen. Im Fall einer datenunabhängigen Anfrage werden die SQL-Anfragen ausgeführt, die Ergebnisse konvertiert und zurückgegeben. Bei datenabhängigen Anfragen muss die SQL-Anfrage mit dem `TypeSorter` in mehrere Anfragen aufgeteilt werden, um dann zum SQL-Server gesendet werden zu können.

`queryMultiple($arSqls)` hat die Aufgabe, alle Anfragen im übergebenen Array an den SQL-Server zu senden und die Ergebnisse zurückzuliefern. Beim Ausführen einer Anfrage werden im Normalfall alle Ergebnisse zurückgegeben. Dies kann die Performance vor allem bei sehr großen Datenmengen negativ beeinflussen. In diesem Fall gibt es die Möglichkeit, sich via `LIMIT` und `OFFSET` nur einen Teil der Ergebnisse zurückgeben zu lassen. Diese Restriktionen können leider nicht ohne Weiteres auf mehrere SQL-Anfragen angewendet werden, da nicht bekannt ist, wie groß die Anzahl der Ergebnisse der Einzelanfragen ist. Um dieses Problem möglichst effizient zu lösen, wird in `queryMultiple` der `SparqlEngineDb_Offsetter` herangezogen, um die für die gesamte SPARQL-Anfrage geltenden Restriktionen auf die SQL-Anfragen zu verteilen. Die Ergebnisse der ausgeführten Anfragen werden in einem Array gesammelt und zurückgegeben.

Die letzte Methode der Klasse, `queryDb($arSql, $nOffset, $nLimit)`, ist für die Ausführung einer einzelnen SQL-Anfrage zuständig und wird von `queryMultiple` verwendet. Sie ruft abhängig von gesetzten LIMITs und OFFSETs die passende Methode der ADODB auf und gibt die Ergebnisse zurück. Im Fall eines Fehlers wird eine Exception geworfen.

5.2.3 Erstellung der SQL-Anfrage mit Sql- und Filtergenerator

SparqlEngineDb_SqlGenerator

Beginnend mit `createSql()` erstellt der Generator mit Hilfe von `getTripleSql(QueryTriple $triple, GraphPattern $graphPattern, $arResultVars)` zuerst einmal für jedes Tripel in der SPARQL-Query den entsprechenden `SELECT`, `FROM` und `WHERE`-Teil der SQL-Anfrage. Dabei muss im `SELECT`-Teil für jede Variable abhängig vom Typ (Subjekt, Prädikat oder

Objekt) die passenden Spalten ausgewählt werden. Bei Prädikaten gibt es nur den Wert des Prädikats an sich, da ein Prädikat immer eine IRI ist. Bei Subjekten kommt noch eine zweite Spalte, der Typ (Blank Node oder Ressource), hinzu. Objekte haben zusätzlich zu diesen beiden Spalten noch je eine Spalte für den Datentyp und die Sprache (Abbildung 5.2).

```
SELECT
 t0.subject as "t0.value_v_s", t0.subject_is as "t0.is_v_s",
 t0.predicate as "t0.value_v_p",
 t0.object as "t0.value_v_o", t0.object_is as "t0.is_v_o",
  t0.l_language as "t0.lang_v_o",
  t0.l_datatype as "t0.type_v_o"
```

Abbildung 5.2: Auswahl von Subjekt, Prädikat und Objekt

Die Auswahl der Tabelle im `FROM`-Teil ist zweigeteilt: Beim ersten Tripel wird die Tabelle `statements` ganz normal ausgewählt, während bei allen darauf folgenden Tripeln die Tabelle per `LEFT JOIN` und der Festlegung auf die gleiche modelID passieren muss (Abbildung 5.3).

```
FROM statements as t0
 LEFT JOIN statements as t1 ON t0.modelID = t1.modelID
```

Abbildung 5.3: Auswahl zweier Tabellen

Bei der Erstellung des ersten `WHERE`-Teils muss überprüft werden, ob die Variablen in einem vorherigen Tripel schon einmal verwendet wurden. Ist dies der Fall, müssen die Spalten der Typen auf Gleichheit geprüft werden, was in `getSqlEqualityCondition($arNew, $arOld)` getan wird. Dies muss für alle Kombinationen von $(Subjekt|Praedikat|Objekt)\{2\}$ unterschiedlich gehandhabt werden, da die Anzahl an verfügbaren Spalten bei Prädikaten nun einmal geringer ist als bei Subjekten und Objekten.

Wird in einem Tripel anstatt einer Variable ein expliziter Wert gesetzt, wird das dazugehörige SQL mit
`getSqlCondition($bject, $strTablePrefix, $strType)` erstellt.

Erst jetzt werden alle Tripel noch einmal durchlaufen, um die `FILTER` zu erstellen. Grund dafür ist, dass bei der sofortigen Filtergenerierung im ersten Durchgang die Variablen, mit denen die Filter arbeiten und auf sie verweisen, noch nicht erstellt sind. Für die Umwandlung der SPARQL-Filter in

SQL-Klauseln für den `WHERE`-Teil ist der `SparqlEngineDb_FilterGenerator` zuständig. Dieser ist in Abschnitt 5.2.3 beschrieben.

Um die Verwendung der erstellten SQL-Teile auch in `UNION`s zu ermöglichen, müssen alle `SELECT`-Anweisungen die gleiche Anzahl an selektierten Spalten haben. Diese müssen sich in der gleichen Reihenfolge befinden. Die Methode `createEqualSelects($arSelect)` kümmert sich darum, dass diese Regeln eingehalten werden und füllt eventuelle Lücken mit `NULL`-Werten auf. Beispiel 5.4 zeigt dies anhand einer SPARQL-Anfrage mit UNION-Teil.

SparqlEngineDb_FilterGenerator

Der FILTER-Abschnitt einer SPARQL-Anfrage ist sehr ähnlich aufgebaut wie die Gleichungen in `if`-Abfragen bei normalen Programmiersprachen. Wahrheitswerte (wahr/falsch) werden mit Hilfe von Operatoren wie *und* (`&&`) und *oder* (`||`) kombiniert; weiterhin können sie mit einem Ausrufezeichen `!` negiert werden. Die Wahrheitswerte können geklammert werden, um komplexe Entweder/Oder-Varianten der Daten zu erlauben. Nur Ergebnisse, die die Kombination an Wahrheitswerten erfüllen, werden zurückgegeben.

Wahrheitswerte werden durch Operatorfunktionen oder dem Vergleich normaler Funktionsergebnisse generiert. Operatorfunktionen sind Funktionen, die als Ergebnis einen booleschen Wert, also wahr/falsch, zurückgeben. Normale Funktionen geben ein Ergebnis eines bestimmten Datentyps zurück, das durch Auffinden der Gleichheit `=` oder Ungleichheit `!=` zu einem anderen Ergebnis in einen Wahrheitswert umgewandelt wird.

Einen FILTER kann man als binären Baum auffassen, der durch Klammerung von Wahrheitswerten und Operatoren sowie der Verschachtelung von Funktionen entsteht. Die Beispiele 5.5, 5.6 und 5.7 zeigen solch einen Baum. Jeder Zusammenschluss zweier Äste kann als Wahrheitswert aufgefasst werden.

Sind mehrere Operatoren ohne Klammern hintereinander aufgereiht, so kann man diese mit Hilfe der in der SPARQL-Recommendation festgelegten Operatorenrangfolge paarweise klammern und den binären Baum trotz fehlender expliziter Gruppierung erstellen. Beispiel 5.7 zeigt dies.

Diese binären Bäume können recht einfach auf SQL abgebildet werden. Das Hauptproblem war, dass der bisherige SPARQL-Parser die Filter nur als String einlas, aber nicht auftrennte oder gar solch einen Baum erstellte. Für diese Funktionalität habe ich den Parser um drei Funktionen erweitert:
`parseConstraintTree($nLevel = 0, $bParameter = false)`
ruft sich selbst während des Parsens immer wieder rekursiv auf, um den

```
SELECT ?name ?mbox
WHERE {
 { ?x foaf:name ?name . ?x  foaf:mbox  ?mbox }
 UNION
 { ?x foaf:mbox2 ?mbox}
}
```

Teil 1	?x	?name	?x	?mbox
Teil 2	-	-	?x	?mbox

```
(SELECT
 t0.object as "t0.value_v_name",
 t0.object_is as "t0.is_v_name",
 t0.l_language as "t0.lang_v_name",
 t0.l_datatype as "t0.type_v_name",

 t1.object as "t1.value_v_mbox",
 t1.object_is as "t1.is_v_mbox",
 t1.l_language as "t1.lang_v_mbox",
 t1.l_datatype as "t1.type_v_mbox"
 ...
) UNION (
SELECT
 NULL as "t0.value_v_name",
 NULL as "t0.is_v_name",
 NULL as "t0.lang_v_name",
 NULL as "t0.type_v_name",

 t1.object as "t1.value_v_mbox",
 t1.object_is as "t1.is_v_mbox",
 t1.l_language as "t1.lang_v_mbox",
 t1.l_datatype as "t1.type_v_mbox"
 ...
)
```

Abbildung 5.4: Auffüllen der SELECT-Auswahl mit NULL-Werten

```
FILTER( (?mbox1 = ?mbox2) && (?name1 != ?name2) )

                +-------- && --------+
                |                     |
          ?mbox == ?mbox2       ?name1 != ?name2
```

Abbildung 5.5: Repräsentation eines einfachen FILTERs durch einen Baum.

```
FILTER( !bound(?foe) && regex(str(?mbox), "@work.example") )

            +-------- && --------+
            |                    |
           (!)                 regex
            |              +-----+------+
          bound            str       "@work.example"
            |              |
          ?foe           ?mbox
```

Abbildung 5.6: Repräsentation eines komplexeren FILTERs durch einen Baum.

```
FILTER( ?a = ?b || ?a = ?c && ?d = ?e && ?d = ?f )

                 +------ && ----+
                 |              |
          +---- && ----+     ?d = ?f
          |            |
    +--- || ---+    ?d = ?e
    |          |
?a = ?b    ?a = ?c
```

Abbildung 5.7: Implizite Klammerung durch Operatorenrangfolge

Filterbaum zu erstellen. Sie schiebt den Zeiger auf das aktive Token immer weiter und verwertet das aktive Token je nach Kontext, indem es neue Unterbäume erstellt oder existierende erweitert. Zur Hilfe kommt ihr dabei `balanceTree(&$tree)` die dafür sorgt, dass die Operatorenreihenfol-

ge bei aufeinanderfolgenden, nichtgeklammerten Operatoren angewendet wird und der binäre Teilbaum wirklich nur jeweils maximal zwei Äste hat. `fixNegationInFuncName(&$tree)` ist noch dafür zuständig, laut Spezifikation erlaubte, mehrfach angegebene Negationen wie `!!!bound(?s)` aufzulösen.

Die Umwandlung dieses Binärbaumes in SQL ist nun relativ einfach zu bewerkstelligen, da man zu jeder Zeit einen Knoten durch die Kombination der beiden Äste mit einem Opertor abbilden kann. Dies kann man mit sehr wenig Code rekursiv abbilden.

Es gibt drei Typen von Knoten und Blättern an diesem Baum:

1. Die *Gleichung* verbindet zwei Kindknoten durch einen Operator miteinander. Sie kommt nur als Knoten, niemals als Blatt vor.

2. Die *Funktion* kommt auch niemals als Blatt vor. Sie hat einen oder mehrere Parameter; die Liste mit erlaubten Funktionen ist in der Recommendation definiert. Beispiele sind `bound` und `regex`.

3. Ein *Wert* tritt nur als Blatt auf und ist entweder die Referenz auf eine Variable - wie zum Beispiel `?mbox` - oder ein explizit angegebener Wert verschiedener Typen wie `"@work.example"@en` oder `23.42`.

Bei der Erstellung einer Gleichung in `createEquation($tree)` muss darauf geachtet werden, dass linker und rechter Wert vom selben Typ sind. Weiterhin muss bei der Typprüfung eventuell ein Typecast verwendet werden, um etwa Integer- und Floatwerte miteinander vergleichen zu können - dies muss aufgrund der unnormalisierten Datenbankstruktur und der als BLOB abgespeicherten Werte per Hand gemacht werden. Ohne Typekonvertierung würde beispielsweise die Überprüfung von `1 = 1.0` ein negatives Ergebnis liefern.

Bei `FILTER`-Patterns in einem `OPTIONAL`-Teil der SPARQL-Anfrage muss zudem noch beachtet werden, dass es möglich ist, dass die Werte überhaupt nicht existieren - also `NULL` sein können. In dem Fall muss die Gleichung trotzdem erfüllt werden können.

Bei der Abbildung von Funktionen in `createFunction($tree)` wird zuerst überprüft, ob die angeforderte Funktion überhaupt erlaubt ist. Dieser Sicherheitsmechanismus verhindert den Missbrauch der SPARQL-Anfrage als Einfallstor in den Server und löst damit das in Abschnitt 4.4.2 beschriebene Sicherheitsproblem des alten SPARQL-Prozessors. Nach der Überprüfung der Parameterzahl wird eine funktionsspezifische Methode aufgerufen (`createFunction_*()`), die die Funktionalität der angeforderten Funktion direkt in SQL abbildet.

Der komplizierteste Teil der Filterbehandlung ist die Abbildung von Werten. Diese müssen je nach übergeordneter Funktion den passenden Datentyp haben und im Fall eines Operators einen booleschen Wert. Die Methoden `createValue($tree, $bDumbParent)` und `createBooleanValue($tree)` werden dafür verwendet.

Werte des Typs `http://www.w3.org/2001/XMLSchema#dateTime` sind aufgrund der unnormalisierten Werte in der Datenbank nur schwer in einen SQL-DateTime-Wert umzuwandeln. Im String des von SPARQL unterstützten Formats `%Y-%m-%dT%H:%i:%s%Z` kann jede beliebige Zeitzone angegeben werden. Je nach Zeitzone muss die Stunde, eventuell auch die Minuten und der Tag - und im schlimmsten Fall der Monat und das Jahr - angepasst werden. Dies ist ohne direkte Unterstützung der Umwandlung von stringförmigen Zeitangaben in DateTime-Werte durch den SQL-Server nicht machbar. Zur Zeit der Programmierung gibt es zwar die Möglichkeit, solche Angaben umzuwandeln, allerdings werden Zeitzonen nicht mit berücksichtigt. Ich habe deshalb nur das Parsen der Zeitzonen nicht mit implementiert, was dazu führt, dass Zeitangaben mit unterschiedlichen Zonen gleich sein können.

In den Abbildungen 5.8 und 5.9 wird beispielhaft gezeigt, wie SPARQL-Filterausdrücke in SQL umgewandelt aussehen. An diesen wird deutlich, wie sehr das unnormalisierte Werte- und Datenkbankformat die SQL-Anfragen verkompliziert, da alle möglichen Varianten an Werten abgefragt werden müssen. Mit einem normalisierten, auf SPARQL angepassten Datenbanklayout würde sich die Komplexität der Anfragen drastisch verringern.

SparqlEngineDb_QuerySimplifier

Im Januar 2007 wurde die SPARQL-Syntax dahingehend erweitert, dass das Erstellen von komplexen kombinierten Anfragen mit weniger Schreibaufwand verbunden ist. Dies macht die Anfragen übersichtlicher und einfacher zu verstehen, bedeutet aber auch, dass der Parser angepasst und sich der SPARQL-Prozessor auf die neuen Möglichkeiten verstehen muss.

Die Anfragesyntax wurde um Unterabfragen erweitert. Beispiel 5.10 zeigt, wie mit drei Tripeln eine `UNION`-Anfrage aufgebaut werden kann, die einer vier-Tripel-Anfrage entspricht. Die Unteranfragen lassen sich selbst wieder mit Unteranfragen versehen, so dass auf einfache Art sehr komplexe Anfragen möglich werden, wie in Beispiel 5.11 zu sehen.

Die Kombination der Rückgabe von normalen Spalten mit Spalten aus Ergebnissen von Subqueries ist in SQL nicht möglich. Deshalb musste ein Weg gefunden werden, die verschachtelten `UNION`-Anfragen in SPARQL auf das flache SQL-Modell zu transformieren. Genau diese Aufgabe übernimmt der `SparqlEngineDb_QuerySimplifier`.

```
SPARQL:
FILTER ( ?price < 30 )

SQL:
AND (
 CAST(
  (CASE WHEN t0.object_is
     = "http://www.w3.org/2001/XMLSchema#dateTime"
    THEN STR_TO_DATE(t0.object, "%Y-%m-%dT%H:%i:%sZ")
    ELSE t0.object END
  )AS DECIMAL
 ) < 30
 AND (
  t0.l_datatype = "http://www.w3.org/2001/XMLSchema#double"
   OR t0.l_datatype
    = "http://www.w3.org/2001/XMLSchema#integer"
 )
)
```

Abbildung 5.8: Abbildung eines SPARQL-Filter in SQL

Schaut man sich das Beispiel 5.10 genauer an, so erkennt man, dass Verschachtelung aufgelöst werden kann, indem die beiden Teile der `UNION` jeweils das übergeordnete Tripel zugeordnet bekommen. Dadurch fällt die oberste Anfrage“ebene” weg, und die `UNION` rückt hoch auf die oberste Ebene, wie in Beispiel 5.12 dargestellt.

Beim “Flachklopfen” der Anfragen muss darauf geachtet werden, dass die Verschachtelung von Anfragen vorkommen kann. Solch komplexe Anfragen müssen von innen nach außen simplifiziert werden; so dass die am tiefsten verschachtelten Tripel zuerst mit den Nächsthöheren zusammengeführt werden.

Die Vereinfachung der Anfrage im QuerySimplifier beginnt damit, dass man die Klasse instantiiert und `simplify(Query $query)` aufruft. Diese Methode macht mehrere Dinge: Zuerst werden alle leeren GraphPattern-Objekte aus der Liste der Anfrage entfernt. Als nächstes wird ein Ausführungsplan erstellt der festlegt, in welcher Reihenfolge welche GraphPatterns miteinander verschmolzen werden sollen. Folgend wird der Plan ausgeführt, und die neue Liste der Patterns an das Objekt zurückgegeben.

`createPlan(&$arPatterns)` beschafft sich mit Hilfe der Methode `getNumbers(&$arPatterns)` eine Liste der Patterns, die am tiefsten ver-

```
SPARQL:
FILTER ("false"^^xsd:boolean || ?v)

SQL:
AND (
 (
  (
    (t0.l_datatype = ''
     t0.l_datatype =
      'http://www.w3.org/2001/XMLSchema#string'
    )
    AND t0.object != ''
  ) OR (
    t0.l_datatype =
     'http://www.w3.org/2001/XMLSchema#boolean'
    AND t0.object != 'false'
  ) OR (
   (t0.l_datatype =
    'http://www.w3.org/2001/XMLSchema#integer'
    || t0.l_datatype =
     'http://www.w3.org/2001/XMLSchema#double')
   AND CAST(t0.object AS DECIMAL) != 0
  ) OR (
    t0.l_datatype != ''
    AND t0.l_datatype !=
     'http://www.w3.org/2001/XMLSchema#string'
    AND t0.l_datatype !=
     'http://www.w3.org/2001/XMLSchema#boolean'
    AND t0.l_datatype !=
     'http://www.w3.org/2001/XMLSchema#integer'
    AND t0.l_datatype !=
     'http://www.w3.org/2001/XMLSchema#double'
    AND t0.object != ''
  )
 )
 || FALSE
)
```

Abbildung 5.9: Abbildung eines SPARQL-Filter in SQL, 2

```
... WHERE {
  ?s ?p ?o .
  { ?s1 ?p1 ?o1 }
  UNION
  { ?s2 ?p2 ?o2 }
}
```

Abbildung 5.10: Eine einfache Unterabfrage

```
... WHERE {
  {
    {
      ?s1 ?p1 ?o1
      {
         {?s12 ?p12 ?o12. ?s13 ?p13 ?o13}
         UNION
         {?s14 ?p14 ?o14. ?s15 ?p15 ?o15}
         UNION
         {?s16 ?p16 ?o16. ?s17 ?p17 ?o17}
      }
    }
    UNION
    {
       {?s22 ?p22 ?o22. ?s23 ?p23 ?o23}
       UNION
       {?s24 ?p24 ?o24. ?s25 ?p25 ?o25}
    }
    ?s ?p ?o
}
```

Abbildung 5.11: Eine komplexe Unterabfrage

schachtelt sind. Nun wird basierend auf dieser Liste eine nach der Tiefe der Verschachtelung sortierte neue Liste mit den Pattern erstellt, die Unterabfragen haben und bearbeitet werden müssen.

Jetzt wird der erstellte Plan ausgeführt. Er muss bei der Abarbeitung dynamisch angepasst werden, da durch das Auflösen von Unterabfragen neue Pattern zu übergeordneten GraphPattern zugeordnet werden müssen, und sich die Zahl der Pattern insgesamt verringert.

```
          / ?s1 ?p1 ?o1
?s ?p ?o |
          \ ?s2 ?p2 ?o2

           -----------
----------
           -----------

Umwandeln:
---------- -----------
---------- -----------

?s ?p ?o. ?s1 ?p1 ?o1
?s ?p ?o. ?s2 ?p2 ?o2
```

Abbildung 5.12: Auflösen von Unterabfragen

5.2.4 Sortieren der Ergebnisse

Aufgrund der allgemeinen Natur der RAP und des damit verbundenen generischen Datenbanklayouts traten beim Abbilden des SPARQL Sortierprozesses auf SQL einige Probleme auf. Deren Gründe und Lösungen werden nachfolgend beschrieben.

SparqlEngineDb_TypeSorter

Das von den RAP-Klassen unterstützte Datenbanklayout hat fünf Tabellen, von denen zwei für den SPARQL-Prozessor interessant sind: Zum einen ist dies die Tabelle `models`, in der den Modell-URIs eine fixe Identifikationsnummer zugeordnet wird, und zum anderen die Tabelle `statements`, in der alle Tripeldaten mit der Referenz auf das Modell abgespeichert werden. Abbildung 5.13 zeigt das Layout der Statementtabelle.

Während alle Prädikate ganz klar *Ressourcen* sind, können die Werte bei Subjekt und Objekt unterschiedliche Datentypen haben. Bei Subjekten ist die Unterscheidung von zwei Typen notwendig: *Blank Nodes* (Unbenannte Knoten; in den `*_is`-Spalten der Tabelle mit dem Zeichen `b` gekennzeichnet) und *Ressourcen* (mit Typzeichen `r` versehen). Objekte hingegen können von einer von drei verschiedenen Typen sein: *Blank Nodes* (`b`), *Ressourcen* (`r`) und *Literale* (in der Tabelle mit `l` bezeichnet). Literale können optional einen Sprachidentifikator und einen Datentyp haben.

Die für die Daten anzuwendende Reihenfolge beim Sortieren ist in der

Spaltenname	Datentyp
modelID	integer
subject	varchar
subject_is	varchar(1)
predicate	varchar
object	blob
object_is	varchar(1)
l_language	varchar
l_datatype	varchar

Abbildung 5.13: Layout der Statements-Tabelle

SPARQL-Recommendation in Sektion 9.1 definiert. Zuerst wird nach Typ der Daten geordnet:

1. Ungebundene Werte
2. Blank Nodes
3. Ressourcen / IRIs
4. Literale
5. Literale ohne Datentyp

Die weitere Reihenfolge innerhalb der Typklassen ist - außer bei Literalen - die normale alphanumerische Sortierreihenfolge (Zeichen für Zeichen). Bei Literalen ist die Reihenfolge der Datentypen untereinander nicht definiert.

Innerhalb der einzelnen Datentypen definiert in Sektion 11.3 die Tabelle "SPARQL Binary Operators" die genaue Reihenfolge. Dies wäre zum Beispiel numerische Ordnung für Datentypen wie `xsd:integer` und `xsd:float`, während es die zeitliche Reihenfolge bei Datums- und Zeitangaben vom Typ `xsd:dateTime` ist.

Bei der Implementierung der `ORDER BY`-Funktionalität nach diesen Regeln stand ich vor zwei Problemen:

1. Die Reihenfolge der Typen der RDF-Daten ist nicht durch eine Sortieranweisung in SQL zu behandeln.
2. Die Objektwerte werden unabhängig vom Datentyp im gleichen Feld `object` als BLOB gespeichert und können nicht ohne weiteres sortiert werden.

Das erste Problem entsteht dadurch, dass die Reihenfolge "Blank Nodes (`b`) - Ressourcen (`r`) - Literale (`l`)" nicht mit einem einfachen `ORDER BY` im SQL zu erreichen ist, da Literale mit dem Kürzel `l` bei normaler Stringsortierung vor den Ressourcen mit `r` eingeordnet werden. Für dieses standen verschiedene Lösungsansätze zur Auswahl:

1. Man definiert in der Datenbank eine Hook-Funktion, die eine spezielle Sortierreihenfolge implementiert, in der das `r` vor dem `l` eingeordnet wird. Dies ist bei einigen Datenbanken machbar, muss allerding durch proprietäre Syntaxerweiterungen zu SQL geschehen. Auf diesem Weg muss für jedes Datenbankmanagementsystem eigener Code implementiert werden; bei manchen Datenbanken sind solche Hooks überhaupt nicht möglich - auf diesen könnte der SPARQL-Prozessor nicht eingesetzt werden.

2. Das Datenbankschema der RAP wird umdefiniert, so dass Literale und Ressourcen so repräsentiert werden, dass die Sortierreihenfolge der Abkürzungen der in der SPARQL Recommendation definierten Reihenfolge entspricht. Diese Lösung hat zum Nachteil, dass die RAP an sich geändert werden muss und mit ihr tausende auf der RAP aufbauenden Programme, die die bisherigen Typenwerte erwarten.

3. Es werden mehrere Anfragen generiert, die die Ergebnismenge auf einen bestimmten Typ beschränken (Blank Nodes, Literale oder Ressourcen). Diese werden in der korrekten Reihenfolge an den SQL-Server geschickt und die Ergebnismengen verkettet. Um die Geschwindigkeit nicht zu verringern und so viel wie möglich Arbeit auf dem SQL-Server verrichten zu lassen, werden diese Anfragen durch ein `UNION` direkt auf dem Server miteinander verbunden. Dadurch kann man die SPARQL-Befehle `LIMIT` und `OFFSET` durch die gleichnamigen SQL-Anweisungen abbilden.

   Diese ideale Lösung funktioniert leider nicht, da `UNION` in SQL als Mengenoperation definiert ist, bei der die Reihenfolge der Ergebnisse nicht festgelegt ist. Bei Tests wurden die Ergebnisse in den verschiedensten Reihenfolgen zurückgeliefert, so dass bei der Umsetzung mehr Aufwand betrieben werden musste.

4. Es werden wie beim vorherigen Lösungsansatz mehrere Anfragen generiert, die die Ergebnismengen auf einen bestimmten Typ beschränken. Diese Mengen werden clientseitig vom SPARQL-Prozessor in der korrekten Reihenfolge zusammengefügt.

Der letzte Lösungsansatz ist der einzige, der ohne Störung anderer Applikationen und ohne Abwärtskompatibilität zu brechen umzusetzen ist. Er bringt jedoch höheren Aufwand auf Clientseite und eine erschwerte Behandlung von `LIMIT` und `OFFSET`-Anweisungen mit sich, was zum Entstehen des `SparqlEngineDb_Offsetter` führte (Abschnitt 5.2.4).

Die Sortierung der Objektwerte bei Literalen mit dem gleichen Datentyp konnte leider auch nicht mit einem `ORDER BY` in SQL gelöst werden, da der SQL-Server aufgrund der Speicherung der Daten im Feld `object` vom Typ BLOB keinerlei Anhaltspunkte hat, wie diese zu sortieren sind. Ein `ORDER BY` sortiert die Daten eines Blobs nach Zeichen für Zeichen nach deren ASCII-Wert. Während dies bei stringbasierten Datentypen wie `xsd:string` korrekt ist, kann dieser Ansatz bei Zahlen zur falschen Ergebnissen führen.

Da die Daten *unnormalisiert* in der Datenbank gespeichert werden, können gleiche Werte durch verschiedene Strings repräsentiert werden. Für die Zahl "eins" kann zum Beispiel `1`, `01` oder auch `1.0` in der Datenbankspalte hinterlegt sein. Eine Sortierung der Zahlen als Zeichenkette führt schon bei den Werten `5` und `09` zu Fehlern, da zuerst die `5` mit der `0` verglichen wird und die `09` vor die `5` eingeordnet werden würde.

Auch in diesem Fall würde es enorm helfen, wenn die Daten je nach Datentyp in einer speziellen Spalte stehen würden, die im Datenbankschema schon dem richtige SQL-Typ statt BLOB zugeordnet ist. Eine Alternative wäre, nur normalisierte Daten in der Datenbank zu erlauben. Beide Ansätze wären mit der Umstellung der RAP an sich verbunden, und würden existierende Datenbanken oftmals invalid und deren Normalisierung erforderlich machen.

Da der Prozessor aufgrund des Problems der Sortierung der RDF-Typen sowieso schon mehrere Anfragen stellen und deren Ergebnisse selbst miteinander verketten muss, wählte ich denselben Weg für die Datentypengruppen der Literale. Der `SparqlEngineDb_TypeSorter` fragt die Datenbank zuerst nach den zu erwartenden Datentypen der Ergebnisspalten und generiert daraufhin für jede Datentypkombination eine Anfrage. Die Sortierung innerhalb eines einzigen Datentyps kann mit Hilfe von `CAST`-Anweisungen gelöst werden, so dass nur noch ein Mapping der XSD-Datentypen in SPARQL zu CAST-Anweisungen in SQL implementiert werden muss. Auch hier ist es durch die nicht definierte Reihenfolge der Ergebnisreihen bei `UNION`-Anweisungen in SQL nicht möglich, eben dieses zu verwenden.

Die kombinierten Lösungen der RDF-Datentypen und Literaldatentypensortierung führen zu einem spezifikationskonformen Sortierverhalten des datenbankbasierten SPARQL-Prozessors. Jedoch verursachen die mehrfachen Anfragen einen Geschwindigkeitsverlust des Prozessors gegenüber Lösungen, die auf für SPARQL optimierte Datenbankschemata aufsetzen. Wei-

terhin wird die manuelle Behandlung von `LIMIT` und `OFFSET`-Anweisungen notwendig, da diese über mehrere SQL-Anfragen abgebildet werden müssen.

Im Code wird durch die Methode `willBeDataDependent()` geprüft, ob die Anfrage überhaupt abhängig von den Daten ist und auf mehrere Anfragen aufgeteilt werden müsste. Ist dies der Fall, kann die Hauptmethode `getOrderifiedSqls($arSqls)` aufgerufen werden. Sie bestimmt durch Aufruf von `getSpecialOrderVariables()` zuerst die Variablen in der `ORDER BY`-Anweisung, die einer speziellen Behandlung bedürfen.

Sobald die Namen der Variablen feststehen, werden in `getTypeSets($arSpecialVars, $strFrom, $strWhere)` Anweisungen in SQL erstellt, mit denen die Kombinationen an RDF-Typen und Literaldatentypen im Ergebnis abgefragt werden können. `orderTypeSets($arTypes)` verwendet diese Informationen, um die Reihenfolge der Datentypkombinationen festzulegen. Die Methode `getSqlOrderBy($arTypeSet = array())` generiert für jede der Datentypkombinationen in Abhängigkeit von den zu erwartenden Datentypen die korrekte `ORDER BY`-Anweisung in SQL.

`getOrderifiedSqls($arSqls)` gibt zum Schluß einen Array mit SQL-Anweisungen zurück, die - in der gegebenen Reihenfolge ausgeführt - zusammengenommen die komplette, sortierte Ergebnismenge der SPARQL-Abfrage liefern.

SparqlEngineDb_Offsetter

Der Offset-Prozessor löst das Problem, dass es durch die Verwendung mehrerer vom `SparqlEngineDb_TypeSorter` generierten SQL-Anfragen nicht mehr möglich ist, `LIMIT` und `OFFSET` direkt in SQL umzusetzen.

Beispiel 5.14 zeigt, wie die `LIMIT/OFFSET`-Restriktionen in SPARQL direkt in SQL umgesetzt werden, da es nur eine SQL-Anweisung gibt. In Beispiel 5.15 ist mehr Aufwand nötig, da das Ergebnis mehrere Datentypen beinhaltet, die wie in Abschnitt 5.2.4 beschrieben unterschiedlich sortiert werden müssen.

```
SPARQL:
 SELECT ?p WHERE { ?s ?p ?o } LIMIT 10 OFFSET 2

SQL:
 SELECT predicate FROM statements LIMIT 2,12
```

Abbildung 5.14: Umsetzung von LIMIT und OFFSET in SQL bei einer Anweisung

```
SPARQL:
 SELECT ?p WHERE { ?s ?p ?o } ORDER BY ?o LIMIT 10 OFFSET 2
```

RDF-Typ	Literaldatentyp	Menge	Anteil im SPARQL-Ergebnis
Blank Node	-	1	0
Ressource	-	3	2
Literal	xsd:integer	2	2
Literal	xsd:string	20	6

```
SQL:
 SELECT object FROM statements WHERE object_is = "r"
  ORDER BY object LIMIT 1,3;
 SELECT object FROM statements WHERE
  object_is = "l" AND l_datatype = "xsd:integer"
  ORDER BY CAST(object AS INT) LIMIT 0,2;
 SELECT object FROM statements WHERE
  object_is = "l" AND l_datatype = "xsd:string"
  ORDER BY object LIMIT 0,6;
```

Abbildung 5.15: Umsetzung von LIMIT und OFFSET mit mehreren SQL-Anweisungen

Die verschiedenen Mengen an Ergebnissen müssen nun auf die im der SPARQL-Anfrage festgelegten Grenzen zurechtgeschnitten werden. Es ist prinzipiell möglich, alle Daten ohne Begrenzung vom Datenbankserver herunterzuladen, in einen Array zu schreiben und dann clientseitig einen bestimmten Teil des Arrays herauszuschneiden und zurückzugeben. Allerdings werden bei dieser Variante vor allem bei großen Datenbeständen unnötigerweise Millionen von Tripeldaten vom Server zum Client geschickt, die am Ende sowieso nicht verwendet werden. Dies würde die Performance belasten und den Prozessor in sehr großem Maße von der Größe des Hauptspeichers des Clients abhängig machen. Zudem können durch die Optimierung des Offset-Prozessors Anfragen oft komplett weggelassen werden, da sie nicht zur benötigten Ergebnismenge beitragen.

5.2.5 Ergebnisrenderer

Das Konvertieren der SQL-Ergebnisse in durch die von den jeweiligen Applikationen benötigte Ergebnisformat erfolgt mit Hilfe von Ergebnisrenderern.

Der verwendete Renderer wird durch den beim Aufruf der Prozessormethode `queryModel` als dritten Parameter übergebenen Wert festgelegt. Es werden die folgenden Werttypen akzeptiert:

1. Kein Wert oder `false` - das Standardergebnisformat von PHP-Arrays wird verwendet.

2. Ein Objekt, welches das Interface `SparqlEngineDb_ResultRenderer` implementiert.

3. Ein String, der den Klassennamen des Renderers enthält.

4. Aus Kompatibilitätsgründen führt ein String mit dem Wert "xml" dazu, dass der XML-Ergebniskonverter verwendet wird - auch wenn dies nicht der komplette Klassenname ist.

Der SPARQL-Prozessor übergibt das ADODB-Ergebnisobjekt dem `SparqlEngineDb_ResultConverter`, der die Unterscheidung zwischen den verschiedenen Parametertypen vornimmt und den eigentlichen Renderer je nach Bedarf instantiiert und aufruft.

Mit Hilfe der Renderer ist es möglich, dass eine Applikation die Ergebnisse durch die Bereitstellung eines Ergebnisrenderers direkt im gewünschten Format erhält. Aufgrund der direkten Verwendung der von der ADODB erstellten Ergebnisobjekte muss auf diesem Weg keinerlei unnötige Konvertierung geschehen. Dies macht die Verwendung von Renderen schneller als die nachträgliche Konvertierung der im Normalfall zurückgelieferten PHP-Arrays in das gewünschte Format.

Kapitel 6

Evaluation

Nach der Analyse des speicherbasierten Prozessors in Kapitel 4 und der Beschreibung der Implementation des neuen Prozessors in Kapitel 5 widmet sich dieses Kapitel dem Vergleich der beiden mit konkurrierenden Implementierungen.

6.1 Geschwindigkeitsevaluation

Ein Teilziel dieser Arbeit besteht darin, einen möglichst effizienten datenbankbasierten SPARQL-Prozessors zu erstellen. In dieser Hinsicht ist es natürlich interessant zu erfahren, wie sich der neue gegenüber dem alten speicherbasierten Prozessor verhält. Weiterhin wurde der Prozessor des Semantischen Web-Frameworks für Java, die Jena ARQ/SDB, sowie die PHP-Implementierung ARC, die in C programmierte Redland librdf und das kommerziell entwickelte Virtuoso in den Vergleich einbezogen. (Siehe die Beschreibung der Werkzeuge in Abschnitt 3.3)

Über den in Jena integrierten SPARQL-Prozessor *ARQ* und der Werkzeuge der *SPARQL Database Tools* (SDB), kann ähnlich wie bei der RAP per Kommandozeile mit SPARQL auf Modelldaten zugegriffen werden. Während ARQ nativ auf Modellen im Speicher agiert, kann sie auch mit Daten auf Datenbankservern arbeiten, was sie zum Vergleichskandidaten für den neuen SPARQL-Prozessor macht.

ARC ist in PHP programmiert und nach Aussagen des Autors auf Geschwindigkeit hin optimiert. Da sie auch nativ auf einer MySQL-Datenbank arbeitet, wird der Vergleich des neuen Prozessors mit ARC interessante Aufschlüsse über die Vor- und Nachteile der beiden Implementierungen bringen.

Redland librdf ist die einzige in C implementierte API und wird als schnellste RDF-Bibliothek angesehen. Der Vergleich wird zeigen, inwieweit

dies stimmt und ob die anderen Implementierungen vielleicht besser optimiert sind und deshalb an Geschwindigkeit gewinnen.

Virtuoso als universeller Datenspeicher integriert die Unterstützung für SPARQL nativ. Die Evaluation wird Aufschluss darüber geben, inwieweit dies Vorteile gegenüber den anderen Implementierungen hat, die auf externen Datenbanken aufsetzen.

6.1.1 Testumgebung

Die Testumgebung war Gentoo Linux auf einem Desktopcomputer mit einem Athlon XP 1700+ und einem Gigabyte Arbeitsspeicher. PHP wurde in Version 5.2.2, MySQL in Version 5.0.38 eingesetzt. Die Java Virtual Machine war von Sun in der Version 1.5.0-11. Sowohl PHP als auch Java wurde ein Hauptspeicher von 768MiB zugewiesen.

Zum Einsatz kam die aktuellste Version von RAP im SVN, Revision 463, was der offiziell veröffentlichten Version 0.9.5 mit einigen zusätzlichen Features entspricht. Das Jena-Framework wurde in der Version `SDB-0.0-beta-1.zip` benutzt, wobei das `beta` sich nur auf den Zustand der Kommandozeilentools bezieht; die intern verwendete Jena-API ist auf dem Stand vom 7.2.2007.

Die aktuell auf der ARC-Homepage verfügbare Version war die 2006-10-14. Beim Test wurde die Redland/librdf in Version 1.0.6 benutzt, und Virtuoso 5.0.1.

6.1.2 Anfragen und Datengröße

Um die Geschwindigkeit der Implementierungen beurteilen zu können, wurden verschiedene SPARQL-Anfragen wiederholt auf die MySQL-Datenbank mit einem Modell fester Größe ausgeführt. Die Modellgröße wurde nach einem abgeschlossenem Testlauf verringert, so dass die Testergebnisse Anfragen auf Modellen der Größe 5 bis hin zu 200.000 Tripeln vergleichen.

Die Modelldaten wurden mit Hilfe des "Univ-Bench Artificial Data Generator" (UBA) der Lehigh University [GPH05] generiert. Dieser Datengenerator wurde schon von den verschiedensten Benchmarks im RDF-Bereich eingesetzt und schien daher auch für diese Evaluation geeignet.

Die getesteten SPARQL-Anfragen sollten es möglich machen, Rückschlüsse auf die Implementierung der Funktionalität im jeweiligen Prozessor zu geben um so abschätzen zu können, weshalb der Prozessor die jeweiligen Werte erreicht hat.

Durch die Anfragen erhoffte ich Antworten auf die folgenden Fragen zu bekommen:

- Wie verhält sich die Implementierung bei kleinen und großen Datenmengen?
- Werden die bei großen Datenmengen oft verwendeten `LIMIT`-Angaben während der Auswertung, oder erst zum Schluss beachtet?
- Inwieweit passiert die Datenauswertung auf der Datenbank?
- Wie ist die Filterauswertung umgesetzt?
- Werden Optimierungen bei kombinierten Filtern vorgenommen?
- Unterstützt die Implementierung komplexe Anfragen?

Dreizehn verschiedene SPARQL-Anfragen wurden nach den oben angegebenen Punkten ausgewählt. Die ersten zehn Anfragen wurden nach dem Gesichtspunkt der Datenunabhängigkeit ausgesucht; sie verwenden keine datenspezifischen Präfixe und machen auch sonst keine Annahmen über den Aufbau der Testdaten. Der Hauptteil der Anfragen wurde trivial gehalten, um den Einfluss des in der Anfrage zu testenden SPARQL-Merkmals auf die benötigte Rechenzeit zu maximieren.

1. `COUNT {?s ?p ?o}`

   Diese (zur Zeit nur von der RAP unterstützte) Anfrage zählt einfach nur die Anzahl der vorhandenen Tripel in der Datenbank.

2. `SELECT ?s WHERE {?s ?p ?o} LIMIT 10`

   Eine sehr einfache Anfrage, die das Subjekt der ersten zehn in der Datenbank gefundenen Tripel zurückliefert. Hierbei zeigt sich, inwiefern der Prozessor das `LIMIT` mit Auslesen der Tripel kombiniert, oder zuerst alle Tripel ausliest und dann alle bis auf die ersten zehn verwirft.

3. `SELECT ?s WHERE {?s ?p ?o} LIMIT 1000`

   Diese SPARQL-Anfrage ist gleich der vorherigen, nur dass das `LIMIT` jetzt auf 1000 Tripel hochgesetzt wird. Die lässt einen Anstieg der Rechenzeit erwarten, wenn die nur die Tripel geladen werden, die auch als Rückgabewert verwendet werden. Ist die Geschwindigkeit gleich der aus der letzten Anfrage, so wird das `LIMIT` erst nach dem Laden der kompletten Tripelliste angewendet.

4. ```
   SELECT ?s WHERE {?s ?p1 ?o1. ?s ?p2 ?o2.
   FILTER(?o2 != ?o1)} LIMIT 10
   ```
   ```

An dieser Anfrage wird die Geschwindigkeit von über Kreuz verbundenen Tripelgruppen getestet. Da die Ergebnisgröße (ohne Limit) mit der Anzahl der Tripel quadratisch ansteigt, ist es sehr wichtig, dass das Limit so frühzeitig wie möglich angewendet wird.

5. `SELECT ?s WHERE {?s ?p1 ?o1. ?s ?p2 ?o2. FILTER(?o2 != ?o1)} LIMIT 1000`

   Dieselbe Anfrage wie die vorherige, wieder mit einem Limit von 1000 anstatt von 10. Hier ist ein ähnliches Ergebnis wie bei 2 zu erwarten, durch die große Zahl an möglichen Lösungen aber um ein Vielfaches langsamer.

6. `SELECT ?s WHERE {?s ?p ?o. FILTER(regex(?o, ‘‘Education’’))} LIMIT 10`

   Regex-Filter werden aufgrund mangelnder, dem SQL-Befehl `LIKE` entsprechenden, Funktionalität in SPARQL hauptsächlich für Stringvergleiche eingesetzt. Diese Funktion ist allerdings sehr kostenintensiv, da der reguläre Ausdruck erst kompiliert werden muss. Auch die Anwendung des kompilierten Ausdrucks ist sehr viel rechenintensiver als normale Stringvergleiche.

7. `SELECT ?s WHERE {?s ?p ?o. FILTER(!isBlank(?o) && regex(?o, ‘‘Education’’))} LIMIT 10`

   Hier wird die Optimierung des Zusammenspiels der Filterfunktionen getestet. In der Anfrage muss der reguläre Ausdruck nicht berechnet werden, wenn schon klar ist, dass das Objekt nicht leer ist.

8. `SELECT ?s WHERE {?s ?p ?o. FILTER(!isBlank(?o) && regex(?o, ‘‘Education’’) || isURI(?o))} LIMIT 10`

   Bei dieser Anfrage muss der reguläre Ausdruck sehr wenig eingesetzt werden, da es in den Daten sehr viele URIs gibt. Die Optimierung der Filterreihenfolge hat großen Einfluss auf die benötigte Zeit.

9. `SELECT DISTINCT ?s ?p ?o WHERE {?s ?p ?o} ORDER BY ?o ?s ?p LIMIT 10`

   Mit dieser Anfrage wird das Verhalten der SPARQL-Implementierungen beim Sortieren von Daten verschiedenster Typen getestet. Durch die Angabe des `DISTINCT`-Schlüsselwortes sollen möglichst heterogene, verschiedentypige Daten für das Sortieren zur Verfügung stehen.

10. `SELECT ?s ?p ?o WHERE {?s ?p ?o}`
    `ORDER BY ?o LIMIT 10 OFFSET 40`

    Diese Abfrage bildet einen typischen Anwendungsfall bei Webanwendungen nach. Große Datenmengen werden auf Internetseiten typischerweise tabellarisch und sortiert auf mehrere Seiten verteilt dargestellt. Dies wird in SPARQL durch `ORDER BY`, `LIMIT` und `OFFSET` umgesetzt.

Um die SPARQL-Implementierungen auch unter realistischen Bedingungen zu testen, sind noch drei auf die Daten des Lehigh-University Benchmarks angepasste Anfragen evaluiert worden. Hierbei wurde vor allem darauf geachtet, möglichst komplexe Anfragen zu verwenden.

1. ```
   PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
   PREFIX test: <http://test/#>
   SELECT
    ?univ ?dpt ?prof ?assProf ?publ ?mailProf ?mailAssProf
   WHERE {
    ?dpt test:subOrganizationOf ?univ.
    ?prof test:worksFor ?dpt.
    ?prof rdf:type <http://test/#FullProfessor.
    ?publ test:publicationAuthor ?prof.
    ?prof test:headOf ?dpt.
    ?publ test:publicationAuthor ?assProf.
    ?prof test:emailAddress ?mailProf.
    ?assProf test:emailAddress ?mailAssProf.
    FILTER(?prof != ?assProf)
   }
   LIMIT 10
   ```

   Diese Anfrage enthält sieben miteinander verknüpfte Tripel, deren Bedingungen alle erfüllt sein müssen um in das Ergebnis aufgenommen zu werden. Anfragen dieser Art werden bei realen Anwendungen am meisten benutzt.

2. ```
   PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
   PREFIX test: <http://test/#>
   SELECT ?publ ?p1 ?p1mail ?p2 ?p2mail ?p3 ?p3mail
   WHERE {
    ?publ test:publicationAuthor ?p1.
    ?p1 test:emailAddress ?p1mail
    OPTIONAL {
   ```

```
  ?publ test:publicationAuthor ?p2.
  ?p2 test:emailAddress ?p2mail.
  FILTER(?p1 != ?p2)
 }
 OPTIONAL {
  ?publ test:publicationAuthor ?p3.
  ?p3 test:emailAddress ?p3mail.
  FILTER(?p1 != ?p3 && ?p2 != ?p3)
 }
}
LIMIT 10
```

In dieser Anfrage werden zusätzlich zum Hauptautor der Publikation noch zwei eventuelle Mitautoren geladen, falls es sie denn gibt. Es kommt auch zu einem Ergebnis, wenn die gesuchte Publikation nur einen oder zwei Autoren hat.

3.
```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX test: <http://test/#>
SELECT ?prof ?email ?telephone
WHERE {
 {
  ?prof rdf:type test:FullProfessor.
  ?prof test:emailAddress ?email.
  ?prof test:telephone ?telephone
 }
 UNION
 {
  ?prof rdf:type test:Lecturer.
  ?prof test:telephone ?telephone
 }
 UNION
 {
  ?prof rdf:type test:AssistantProfessor.
  ?prof test:emailAddress ?email
 }
}
LIMIT 10
```

Diese Anfrage verbindet drei Teilanfragen zu einem Ergebnissatz. Ein

Problem bei dieser Anfrage ist, dass in der ersten Unteranfrage drei Ergebnisvariablen ausgewählt werden, während in den beiden folgenden nur jeweils zwei Variablen belegt werden.

6.1.3 Durchführung

Zur Durchführung der Laufzeitmessungen verwendete ich ein PHP-Script, welches eine Liste der Anfragen und der Anzahl der gewünschten Tripelzahlen enthielt. Dieses löscht am Anfang einer Testreihe alle nicht benötigten Tripel aus der Datenbank und beginnt danach, zehn Mal ein weiteres Script mit der jeweiligen SPARQL-Anfrage als Kommandozeilenparameter aufzurufen. In diesem Script wird der SPARQL-Prozessor instantiiert und die Anfrage ausgeführt. Die für die Instantiierung und Ausführung benötigte Zeit wird direkt in diesem Programm gemessen, um die für das Laden der ZEND-Engine - dem Unterbau von PHP - benötigte Zeit außer acht zu lassen.

Beim Ausführen der Messungen für die *Jena-SDB* stellte ich mit Erstaunen fest, dass die das Programm selbst bei den trivialsten Anfragen mindestens sechs Sekunden benötigte. Weitere Nachforschungen ergaben, dass die Java Virtual Machine beim Laden der Klassendateien sehr langsam ist. Abhilfe schaffte hier ein Script, welches vor der eigentlichen Anfrage einen "Trockenlauf" durchführt, der dafür sorgt, dass zur Messung alle Klassen bereits in den Speicher geladen sind. Um die Chancengleichheit zu wahren, verwendete ich diese Technik auch bei den Benchmarks mit den PHP-Prozessoren, was auch dort zu einer spürbaren Verbesserung der Geschwindigkeit führte.

ARC hatte Probleme mit dem kreuzweisen Verbinden der Datensätze und brach mit einem Fehler bei diesen Anfragen ab. Auch bei den drei komplexen Anfragen stieg ARC aus und lieferte keine Ergebnisse.

Redland/librdf zeigte beim Test einen Fehler beim Freigeben von Speicher: Abhängig von der Größe der Datenbank dauerte es bis zu 10 Sekunden, bis sich das Kommandozeilenprogramm `rdfproc` nach dem Anzeigen der Ergebnisse beendete. Da es sich hierbei höchstwahrscheinlich um einen behebbaren Bug handelt, benutzte ich die *Redland Bindings*, um die Bibliothek von PHP aus zu benutzen. Dabei verzichtete ich auf die Verwendung von `librdf_free_query_results`, da diese nachweislich die Verlangsamung produzierte. Weiterhin hat Redland Probleme mit `UNION`-Anfragen und lieferte keine Ergebnisse, obwohl die Teilanfragen einzeln ausgeführt sehr wohl Resultate lieferten.

Virtuoso als einziges kommerzielle entwickeltes Produkt wurde mit Hilfe des ODBC-Treibers und des mitgelieferten Kommandozeilenprogramms

`isql` angesprochen. Es stellte sich heraus, dass Virtuoso einen Bug in der Behandlung von regulären Ausdrücken hat und die drei Filteranfragen nicht ausführen konnte. Beim Ausführen der komplexen `OPTIONAL`-Anfrage brach das mit einer Fehlermeldung ab, die laut Programmieren auf zu restriktive Variablenprüfungen zurückzuführen ist.

Die hier angenommene Voraussetzung, dass die Klassen bei der Ausführung von Anfragen bereits in den Speicher geladen wurden, wird auch bei Applikationen in der Praxis erfüllt. In den meisten Anwendungen werden sehr viele SPARQL-Anfragen gestellt, so dass die Zeit zur ersten Initialisierung nicht weiter ins Gewicht fällt. Weiterhin gibt es vor allem im PHP-Bereich Opcode-Caches wie den E-Accelerator [1] oder den Advanced PHP Cache APC [2] , die das Parsen der Dateien unnötig machen, da alles schon im Speicher gecached wird.

6.1.4 Benchmarkergebnisse

Die Zeitachse der Diagramme ist logarithmisch dargestellt, da die Laufzeitunterschiede zwischen den Prozessoren teilweise das 10.000-fache betrugen. Die Y-Achse zeigt die Zeit in Sekunden, während die X-Achse die Anzahl der Tripel in der Datenbank abbildet. Weiterhin konnte der alte Prozessor der RAP nur bis 10.000 Tripel getestet werden, da der Speicher für mehr Daten nicht ausreichte.

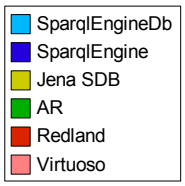


Abbildung 6.1: Diagrammlegende

Auf den Diagrammen ist, wie in der Legende 6.1 zu sehen, der neue Prozessor hellblau, der alte dunkelblau, die Jena SDB gelb und ARC in grün dargestellt. Redland/librdf ist rot und Virtuoso in rosa abgebildet.

Beim Diagramm 6.2 wird sichtbar, dass der alte Prozessor auf Modelle im Speicher angepasst ist. Die stark steigenden Zeiten weisen darauf hin,

[1] `http://eaccelerator.net/`
[2] `http://pecl.php.net/package/APC`

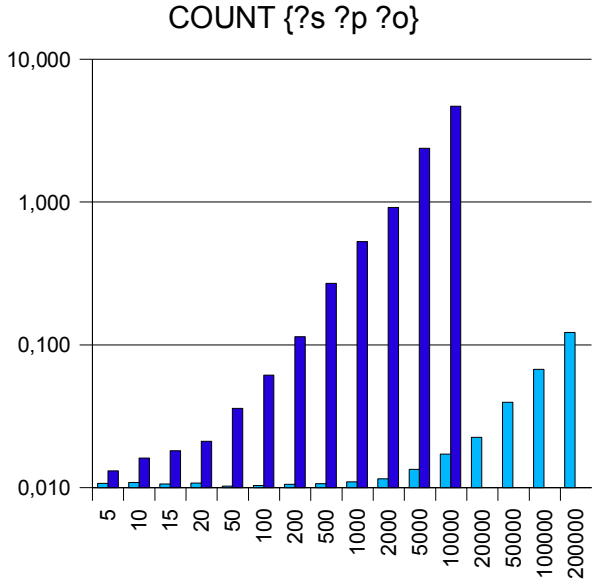


Abbildung 6.2: Zählen der Tripel in der Datenbank

dass zuerst alle Tripel aus der Datenbank in den Speicher geladen werden, um dort gezählt zu werden. Der neue Prozessor ist bei bis zu 1000 Tripeln im Bereich von 10ms für die Anfrage, danach benötigt der SQL-Server selbst mehr Zeit, um die Tripel zu zählen.

Beim Vergleich der beiden Diagramme in Abbildung 6.3 fällt auf, dass der alte Prozessor unabhängig vom Limit dieselbe Zeit benötigt. Diese Zeit hängt linear mit der Anzahl der Datensätze in der Datenbank zusammen. Auch hier werden zuerst alle Tripel von der Datenbank in den Speicher geladen, um dann daraus auszuwählen.

Während der Zeitaufwand bei der *Jena SDB* bis 500 Tripel gleich bleibt, steigt dieser bei mehr Datensätzen unerklärlicherweise immer weiter an - im Gegenzug dazu der neue Prozessor, dessen Zeitbedarf bei 10 zurückzugebenden Tripeln konstant bei 14ms liegt. Bei 1000 Rückgabewerten steigt der Zeitbedarf auch an, was allerdings nicht an der Ausführung der Anfrage an sich, sondern damit zu tun hat, dass die Resultate aus der Datenbank zum Client transferiert und dort in PHP-Objekte umgewandelt werden müssen. Sobald die Anzahl von 1000 Tripeln erreicht ist, ist die Zeit wieder konstant bei 180ms - unabhängig von der Größe der Datenbank, während die Jena SDB für die 100.000 und 200.000 jeweils 1.9 und 3.4 Sekunden benötigt.

Sowohl der neue Prozessor als auch *ARC* und *Redland* benötigen nahezu konstante Zeit unabhängig von der Korpusgröße. Bei den großen Datenban-

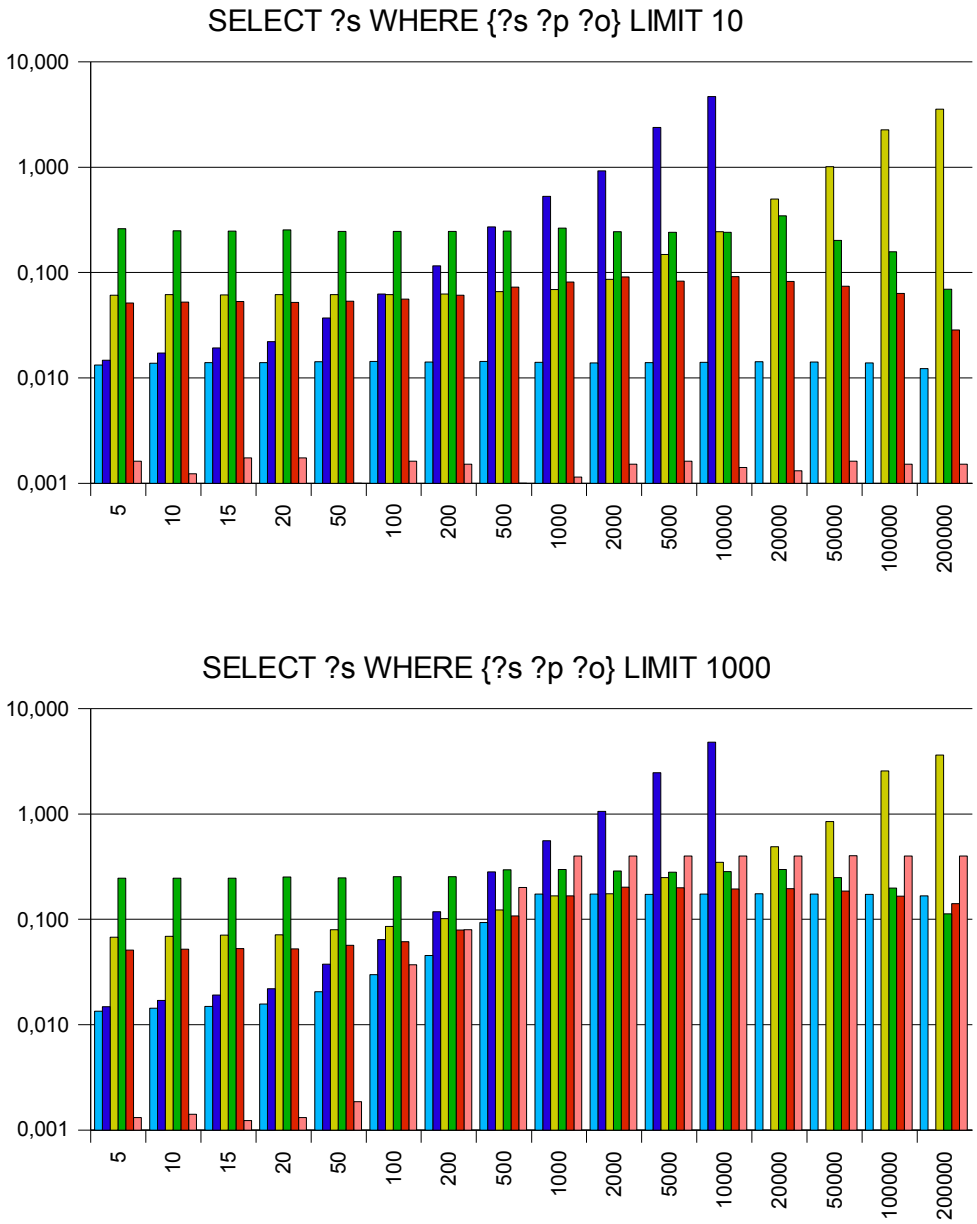


Abbildung 6.3: Einfaches Auswählen von Tripeln

ken weisen alle drei eine Tendenz zu weniger Zeitverbrauch auf, was wohl an der unterliegenden MySQL-Datenbank liegt.

Bei vielen Ergebnissen zeigt sich bei ARC der Vorteil der Designentscheidung, nur PHP-Arrays anstatt von Objekten für die Repräsentation von Ergebnissen zu verwenden - dies führt in PHP generell zu einer Geschwindigkeitssteigerung.

Virtuoso glänzt mit Antwortzeiten von 1-2 Millisekunden, die erst bei einem Limit von 1000 und einer Tripelzahl von 1000 sprunghaft auf 100-400ms ansteigt. Die größere Zeit könnte an der Implementierung des `isql`-Programms liegen, welches mehr Ergebnisse darstellen muss. Dies erklärt allerings nicht den sprunghaften Anstieg bei der Erhöhung der Tripelzahl von 50 auf 100; eine passendere Schlussfolgerung wäre deshalb, dass der Zwischenspeicher von Virtuoso nur eine bestimmte Größe hat. Ab 100 Tripeln reicht dieser nicht mehr aus, und Virtuoso befindet sich zeitlich in einer Riege mit den anderen Prozessoren.

Beim kreuzweisen Verbinden zweier Tripelsätze in Abbildung 6.4 erreicht der speicherbasierte SPARQL-Prozessor die Grenzen des Arbeitsspeichers; bei 10.000^2 Datensätzen bricht die Ausführung des Benchmarkscripts ab, da der Speicher voll ist. Bis dahin sieht man auf dem Diagramm allerdings einen wie zu erwartenden quadratischen Anstieg des Zeitverbrauchs des Prozessors; er braucht bei der Datenbankgröße von 10.000 Tripeln die geschlagene Zeit von fast 21 Minuten, bis das Ergebnis zurückliefert wird.

Die *Jena SDB* benötigt bis 500 Tripel eine nahezu konstanter Zeit von unter 150 Millisekunden. Ab dieser Grenze steigt aber auch ihre Ausführungszeit linear mit der Größe der Datenbank an, wobei es keinen Unterschied macht, ob nur 10 oder 1000 Tripel angefordert werden. Zeitgleich mit dem alten Prozessor braucht die Jena SDB auch hier bei den größten Datenbanken die längste Zeit zur Ausführung; 1 Minute und 20 Sekunden wird bei 200.000 Tripeln benötigt.

ARC steigt bei diesen Anfragen mit einem Fehler aus, so dass sie hier nicht verglichen werden kann.

Redland als C-Bibliothek zeigt einen ähnlichen Zeitverlauf wie der neue datenbankbasierte Prozessor und kommt bei 1000 Ergebnissen durch die PHP-Objekte entstandenen Geschwindigkeitsverlust nah an den Prozessor heran, wobei Redland doch immer einen Tick langsamer ist.

Virtuoso hat wieder bei einer kleinen Tripelzahl eine extrem kleine Antwortzeit; diese steigt jedoch ab 1000 Tripeln wieder in den Bereich des neuen Prozessors und *Redland*. Bei einem großen Limit von 1000 Ergebniszeilen und einer Datenbankgröße von 20.000 Tripeln benötigt sie sogar doppelt oder dreimal soviel Zeit wie die beiden anderen Prozessoren.

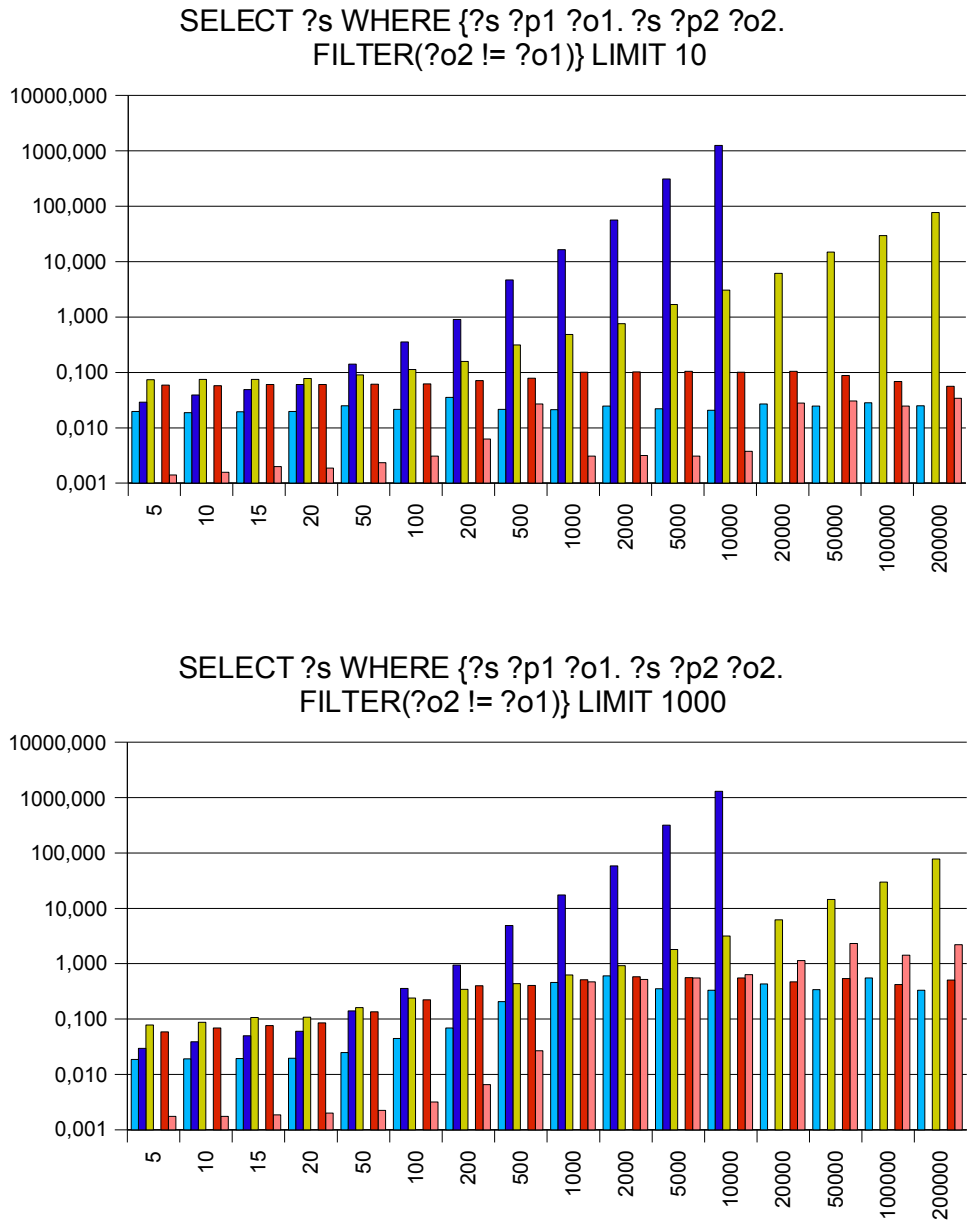


Abbildung 6.4: Kreuzweises Verbinden von Tripeln

Die ersten beiden Diagramme in Abbildung 6.5 unterscheiden sich erwartungsgemäß nur minimal im einstelligen Millisekundenbereich, da es bei zweiten Anfrage keinerlei Blank Nodes gibt, durch die das Testen des regulären Ausdrucks unterbleiben könnte.

Bei der dritten Anfrage kann durch die Überprüfung auf den Typ URI, der in den Datensätzen reichlich vorhanden ist, die Ausführung von `isBlank` und `regex` unterbunden werden. Dies machen sich aber nur der neue Prozessor, ARC und Redland zu nutze; die beiden anderen SPARQL-Prozessoren zeigen auch hier das gleiche Leistungsverhalten wie in den ersten beiden Anfragen. *Redlands librdf* und *ARC* zeigen bei 200.000 beziehungsweise 100.000 Tripeln eine drastische Verbesserung, die jedoch nicht erklärt werden kann. *Virtuoso* brach die Ausführung dieser Anfragen mit einem Fehler ab.

Beim Sortieren der Tripel zeigt sich in Abbildung 6.6 - mit einer Ausnahme - bei allen Prozessoren das gleiche Bild: Die für die Anfragen benötigte Zeit nimmt proportional mit der Anzahl der Tripel in der Datenbank zu.

Erstaunliche Unterschiede ergeben sich beim quantitativen Vergleich der benötigten Zeiten. *ARC* und *Virtuoso* benötigen 11 und 12 Sekunden für 200.000 Tripel die mit Abstand wenigste Zeit. *Jena SDB* wendet mit 25 Sekunden die doppelte Zeit auf, während der neue Prozessor auch noch einmal fast die doppelte Zeit rechnet (45 Sekunden).

Redland ist mit 84 Sekunden abgeschlagen auf dem letzten Platz. Der alte SPARQL-Prozessor der RAP kann aufgrund von Speicherlimitierungen nur bis 10.000 Tripel getestet werden, braucht allerdings mit großem Abstand viel länger als alle anderen Prozessoren.

Auch beim Test in Abbildung 6.7 wächst die aufgewendete Zeit proportional zur Größe der Datenbank, und auch hier unterscheiden sich die benötigten Zeiten der Prozessoren drastisch.

ARC zeigt bis 50 Tripel ein normales Verhalten; die Zeit fällt dann allerdings auf durchschnittlich 10 Millisekunden ab, was nur durch einen Programmfehler erklärt werden kann.

Im Gegensatz zum letzten Test ist der neue Prozessor hierbei allerdings gleichauf und mit 3,1 Sekunden sogar etwas schneller als Virtuoso (4,7 Sekunden) bei 200.000 Tripeln.

Jena zeigt auch quantitativ fast das exakt gleiche Verhalten wie im letzten Test und folgt mit 24,8 Sekunden. *Redland* benötigt bei diesem Test mit 400 Sekunden erstaunlicherweise mehr Zeit als beim vorangegangenen Test. Alle anderen Prozessoren sind um ein Vielfaches oder zumindest ein wenig

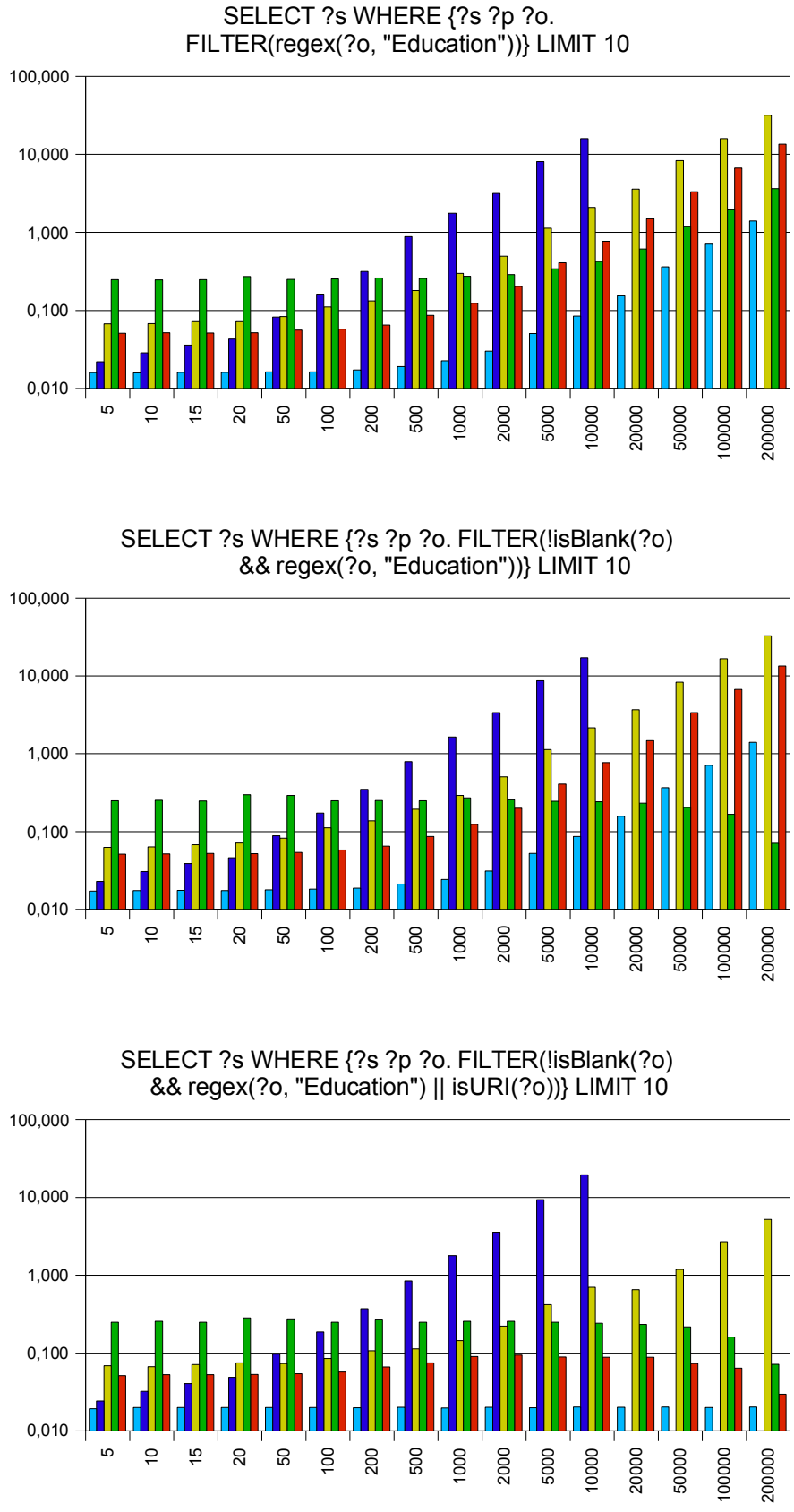


Abbildung 6.5: Filterbenchmarks

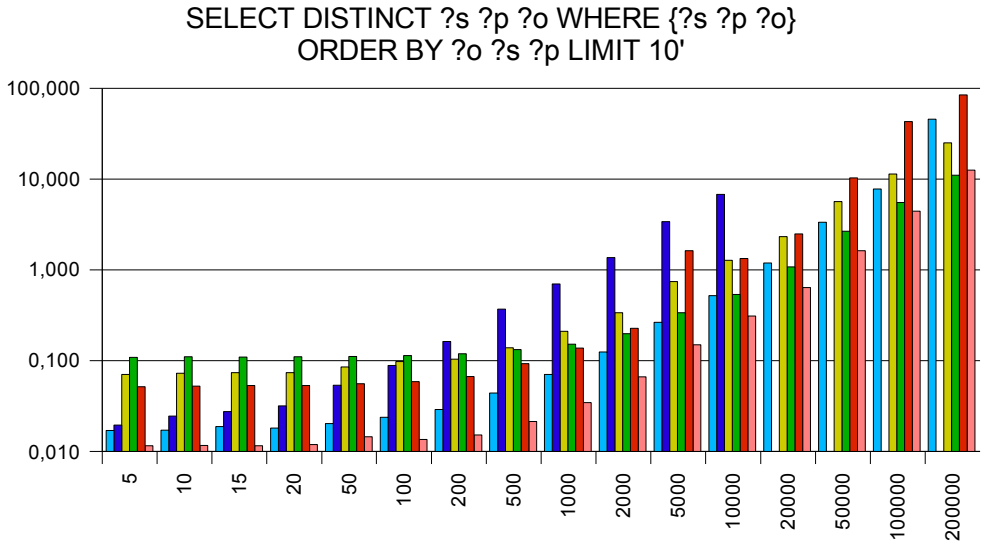


Abbildung 6.6: Sortieren auf distinkten Tripeldaten

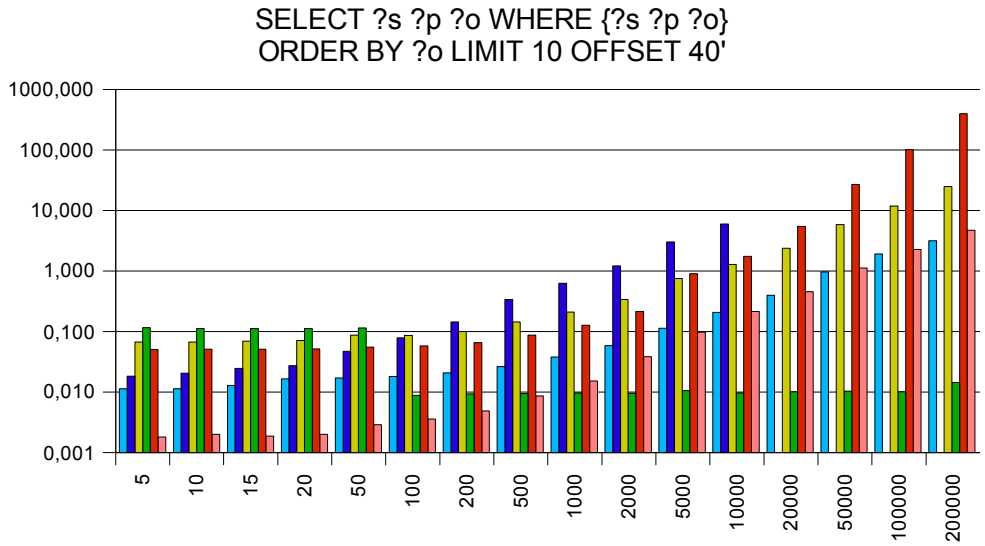


Abbildung 6.7: Sortieren mit Datenfenster

schneller als vorher.

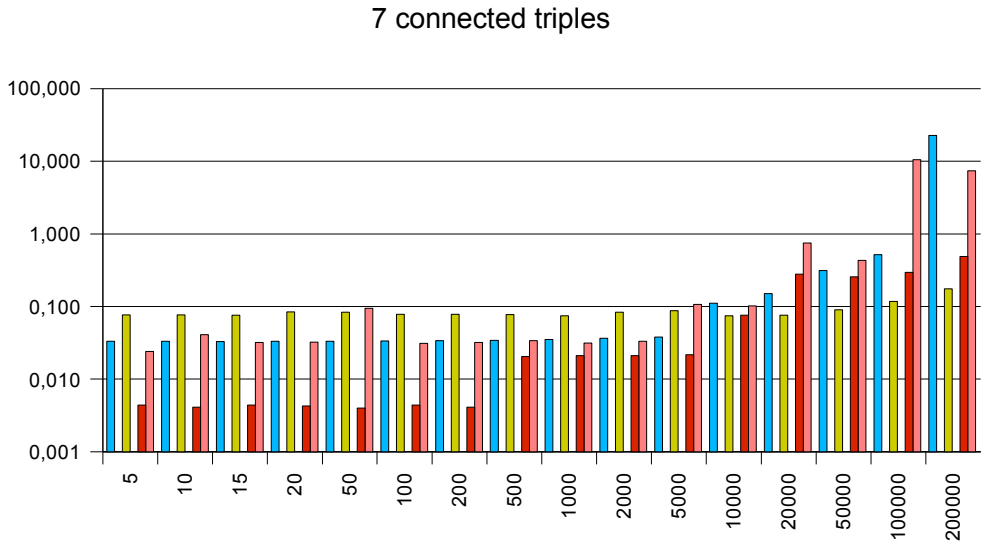


Abbildung 6.8: Komplexe Abfrage mit sieben Tripeln

Die Abfrage der sieben miteinander verknüpften Tripeln in Abbildung 6.8 brechen sowohl der alte Prozessor als auch *ARC* mit einem Fehler ab. Die anderen Prozessoren zeigen ein voneinander abweichendes Verhalten.

Jena führt die Anfragen unabhängig von der Korpusgröße mit fast konstanter Zeit von 75 Millisekunden aus; erst ab 200.000 Tripeln tritt eine Verdoppelung des Zeitaufwandes ein. Bei *Redland* tritt diese drastische Erhöhung des Zeitaufwandes schon bei 20.000 Tripeln auf; ist bis 5.000 Tripel aber mit großem Abstand bester der Disziplin (4 - 20 Millisekunden). Der neue Prozessor und *Virtuoso* benötigen mit um die 33 Millisekunden in etwa die gleiche Zeit für die Anfragen. Bei 100.000 Tripeln tritt bei Virtuoso, und 200.000 Tripeln beim neuen Prozessor ein Ausreißer auf 10 beziehungsweise 22 Sekunden auf, der nicht oder nur durch Anomalien bei der Anordnung der Daten in der Datenbank erklärt werden kann.

Nur der neue Prozessor, *Jena SDB* und *Redland* führten die Anfrage in Abbildung 6.9 korrekt aus. Der datenbankbasierte Prozessor zeigt bei kleinen Korpera konstanten Zeitverbrauch, ab 5000 Tripel steigt dieser jedoch nahezu linear mit der Tripelzahl an. Bei *Jena* und *Redland* steigt die Zeit

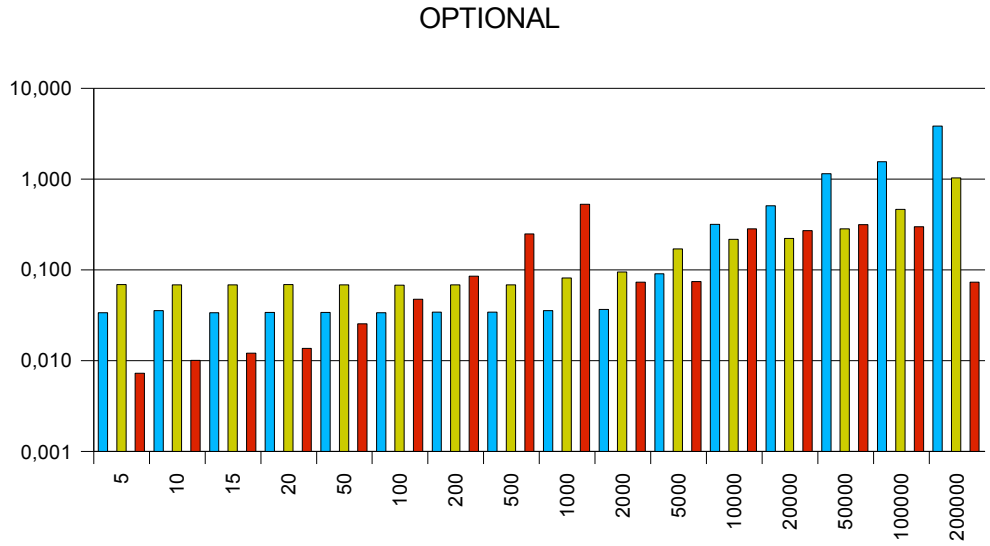


Abbildung 6.9: Komplexe Abfrage mit OPTIONAL

zwar auch mit wachsender Datenbankgröße, jedoch nur leicht anstatt linear. Es fällt auf, dass *Redland* ab 20.000 Tripeln ein fast konstantes Zeitverhalten zeigt und im Mittel schnellster Prozessor vor Jena ist.

Diese letzte Anfrage in Abbildung 6.10 wurde wieder nur von drei Prozessoren korrekt bearbeitet; dem neuen Prozessor, *Jena SDB* und *Virtuoso*. *Jena* glänzt mit fast konstantem Zeitverhalten von 70 bis 90 Millisekunden. *Virtuoso* benötigt bis 2000 Tripel unter 10 Millisekunden; der Zeitverbrauch steigt dann jedoch stark auf bis 450 Millisekunden für 200.000 Tripel an. Der neue Prozessor hat bis 2000 Tripel einen konstanten Verbrauch von 37 Millisekunden. Wie bei Virtuoso steigt die benötigte Zeit danach an, jedoch viel stärker als bei Virtuoso. Er endet mit 18 Sekunden für 200.000 Tripel.

6.1.5 Einschätzung

Die Ergebnisse des Benchmarks sind für den alten speicherbasierten Prozessor nicht überraschend: Tobias Gauß schrieb schon in seiner Arbeit zum Prozessor:

> Aus den Laufzeiten lässt sich auch erkennen, dass die Verwendung der RAP SPARQL-Engine für sehr große Modelle mit sehr

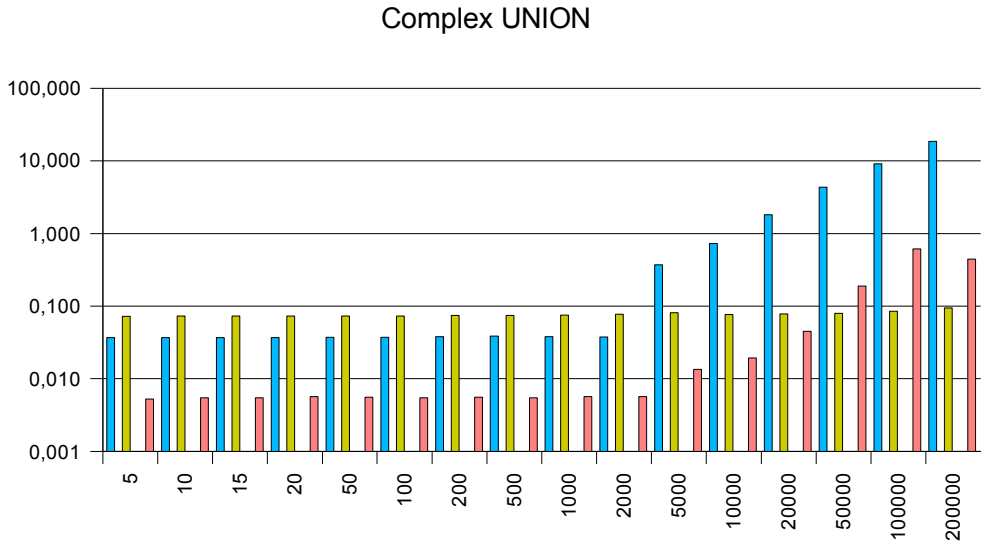


Abbildung 6.10: Komplexe Abfrage mit UNION

komplexen Abfragen aufgrund der Laufzeit problematisch werde könnte.

Selbst bei den kleinen Modellen mit 5 oder 10 Tripeln ist der neue datenbankbasierte SPARQL-Prozessor 1.1 bis 2-fach schneller. Grund hierfür ist, dass der Prozessor die Datenbank trotz allem als Datenspeicher nutzt und die Kosten des Auslesens der Daten gegenüber denen des internen Filterns zu hoch sind.

Ein Benchmark, der den Einsatz des Prozessors mit komplett im Speicher gehaltenen Modellen einschätzt, würde zumindest bei kleineren Datengrößen einen Vorteil für ihn bringen. Weiterhin zeigt die Auswertung deutlich, dass die Obergrenze der Datenbankgröße, bei der noch vernünftiges Arbeiten möglich ist (Reaktionszeiten von unter einer Sekunde), je nach Anfrageart bei 200 bis 2000 Tripeln liegt. Dies mag für sehr kleine Projekte ausreichend sein, ist schon bei "normalen" Datenbeständen zu wenig.

Beim Vergleich des neuen Prozessors mit der *Jena SDB* ergibt sich ein differenzierteres Bild: Zuerst einmal ist die initiale Ladezeit der auf Java basierten Jena API (100ms) um durchschnittlich das Fünffache höher als beim PHP-basierten neuen Prozessor (19ms). Trotz dieses Vorsprungs zieht sie bei den Anfragen 3 und 5 bei 1000 und 2000 Tripeln gleichauf. Dies könnte

daran liegen, dass das Erstellen der Ergebnisobjekte im PHP im Standard-Ergebnisrenderer sehr langsam im Vergleich zur vorgegebenen Ausgabe der Tripel als Text auf der Konsole in der Jena ist. Mit dem Vergleich eines neu zu schreibenden `SparqlEngineDb_ResultRenderer_Console` könnte ein anderes, für den neuen Prozessor besseres Ergebnis erzielt werden. Weiterhin stellt man fest, dass auch Jena ARQ/SDB das Limit erst am Ende der Berechnungen anwendet, anstatt bei 200.000 Tripeln schon nach den ersten 1.000 gefundenen und benötigten Tripeldaten abzubrechen.

Beim Vergleich der Ergebnisse der Anfragen 7 und 8 lässt sich feststellen, dass der Prozessor eine Queryoptimierung durchführt und diese auch funktioniert. Insgesamt ist die Jena ARQ/SDB klar langsamer als der neue Prozessor, könnte bei Einbeziehung der Limits in die Optimierung aber einiges wettmachen. Weiterhin ist anzumerken, dass Jena neben des neuen Prozessors der einzigste war, der alle Anfragen korrekt ausgeführt hat.

ARC zeigte sich im Test als schneller, aber sehr unvollständig implementierter SPARQL-Prozessor. Nur sechs von zwölf Tests wurden korrekt ausgeführt; bei einigen lieferte ARC einfach ein "false" zurück, bei anderen gar komplett falsche Ergebnisse. Die relativ hohe Geschwindigkeit bei der Rückgabe vor allem großer Ergebnismengen hängt damit zusammen, dass ARC intern PHP-Arrays anstelle von Objekten verwendet. Schaut man sich alle Diagramme an, so ist zu sehen, dass ARC wie Redland und der neue Prozessor die Arbeit stark auf die SQL-Datenbank verlagern.

Redland/librdf hat durch die Implementierung in C das Potential, um einiges schneller zu sein als die in PHP und Java implementierten SPARQL-Prozessoren. Dieses Potential wird von Redland allerdings nicht voll ausgenutzt; die Geschwindigkeit im Vergleich zu anderen Systemen ist durchschnittlich. Weiterhin brach sie beim Ausführen der UNION-Anfrage ab, löste aber alle anderen Anfragen ohne Probleme. Das hier als Kommunikationsschicht zur C-Bibliothek verwendete PHP-Modul weist sehr viele Speicherlecks auf, und Verbindungen zum SQL-Server werden nicht wieder geschlossen. Insgesamt ist Redland eine robuste Implementierung mit kleinen Problemen.

Virtuoso als einziges getestetes kommerzielles Produkt ist im Test auch der schnellste SPARQL-Prozessor. Die Regex-Tests offenbarten einen Bug, der laut Virtuoso-Entwicklerteam in naher Zukunft behoben wird. Eine nähere Betrachtung der Ergebnisse zeigt, dass Virtuoso bei kleinen Datenbankgrößen extrem kurze Antwortzeiten von fünf oder weniger Millisekunden liefert. Bei größeren Datenbanken steigt diese Zeit allerdings relativ

stark an, so dass Virtuoso bei Größen ab 5000 Tripeln in ähnlichen Zeitbereichen operiert wie die anderen Prozessoren, manchmal sogar langsamer ist.

Der neu entwickelte Prozessor erweist sich im durchgeführten Benchmark als stabil und schnell. Er gewinnt vor allem dadurch Geschwindigkeit, dass die komplette Auswertung der Daten auf dem Datenbankserver erfolgt, der für diese Aufgabe optimiert ist.

6.2 Verbesserungsmöglichkeiten

Die Benchmarks zeigen auf, an welchen Stellen der Code noch verbessert werden kann, um mehr Geschwindigkeit zu erreichen.

1. Beim Sortieren fällt auf, dass die für das Sortieren notwendige Zeit überproportional mit der Anzahl der verwendeten Sortiervariablen ansteigt.

2. Optionale Klauseln sind auf kleinen Datenbanken sehr schnell, werden aber mit zunehmender Datenbankgröße um einiges langsamer. Andere Prozessoren erreichen einen fast konstanten Zeitverlauf, oder verbrauchen mit steigender Größe nicht so sehr viel mehr Zeit.

3. Die Behandlung von `LIMIT` in Verbindung bei `UNION`-Klauseln passiert erst zum Schluss, nachdem die volle Ergebnismenge berechnet ist. Dies führt dazu, dass der Prozessor 18 Sekunden für eine Anfrage benötigt, während andere Prozessoren nach weniger als einer halben Sekunde Ergebnisse liefern können.

4. Die Geschwindigkeit von reguläre Ausdrücken kann erhöht werden, indem bei einfachen Ausdrücke passende `LIKE`-Klauseln verwendet werden.

6.3 Unittests

Die RDF API for PHP ist eine komplexe Softwarebibliothek, die aus vielen Teilen besteht. Sie hat eine Größe von 51028 Zeilen PHP-Code [3] , die auf verschiedene Teilbereiche wie “Modelle”, “RDQL”, “SPARQL” und “Syntax” verteilt sind.

[3]Ermittelt mit dem Programm `slocccount` am 14.08.2007

Bei der geschlossenen Entwicklung von Software in einer Firma gibt es oftmals einige Programmierer, die über Jahre hinweg an derselben Applikation arbeiten und deren Logik, Design und Fehler genau kennen. Anders ist dies bei RAP und Open-Source-Software im Allgemeinen: Durch die offene Bereitstellung des Quellcodes gibt es viele Programmierer, die der RAP neue Funktionen hinzufügen und gefundene Fehler ausbessern. Entwickler benutzen RAP meist projektbezogen für einen begrenzten Zeitraum, oder sind zwar längerfristig damit beschäftigt, dann aber nur mit einem bestimmten Teilbereich.

Fehler im Code werden teilweise erst gefunden, wenn der ursprüngliche Programmierer nicht mehr an der RAP arbeitet. Es ist dann an anderen Entwicklern, die Fehlerursache in denen für sie meist unbekannten Teilen des Quellcodes zu finden und zu beheben. Gerade bei komplexen Problemen kann es dazu kommen, dass durch das Reparieren eines Fehlers neue entstehen. [4]

Um dauerhaft sicherzustellen, dass die RAP funktioniert und sich einmal behobene Fehler nicht wieder neu einschleichen, verfügt sie über so genannte Modultests, englisch "Unit tests". Dabei handelt es sich um ausführbare Codefragmente, die das Verhalten einer einzelnen Komponente überprüfen und mit festgelegten Vorgaben vergleichen. Das so spezifizierte Verhalten kann durch das Ausführen der Tests reproduziert und wiederholt überprüft werden. Damit kann jeder Programmierer sofort sehen, ob seine Veränderungen am Programmcode dazu führen, dass das Programm nicht mehr so arbeitet wie geplant.

Die Modultests der RAP verwenden das SimpleTest Framework [5]. Ein Großteil der Funktionalität der RAP wird durch die Tests überprüft, die an sich eine Größe von 11042 Zeilen Quellcode haben. Auch der neue SPARQL-Prozessor wird mit Hilfe dieser Tests auf Korrektheit überprüft.

Tobias Gauß legte bei seiner Arbeit am speicherbasierten SPARQL Prozessor den Grundstein dafür. Die SPARQL-Tests wurden in jeweils drei Teile aufgespalten - Anfrage, Daten und erwartete Rückgabewerte -, die für jeden Test jeweils in einer einzelnen Datei abgelegt wurden. Durch diese Aufteilung war es möglich, die gleichen Tests mit relativ wenig Aufwand für den neuen SPARQL-Prozessor weiterzuverwenden.

[4] Dazu der bekannte Vers "99 little bugs"

> 99 little bugs in the code,
> 99 little bugs...
> Knock one down, and test it again,
> 101 little bugs in the code...

[5] `http://simpletest.org/`

Die Tests umfassen verschiedene Gebiete:

- 18 vom W3C bereitgestellte Tests ("DAWG-Tests")
- 9 Sortiertests
- 3 Limit- und Offsettests
- 45 Filtertests
- 74 Tests vom Jena-Projekt ("ARQ-Tests")
- 2 Tests für ASK-Funktionalität

Von diesem 151 Tests besteht der neue SPARQL-Prozessor 148. Die drei fehlschlagenden Tests sind die folgenden:

- `expr-1` wird nicht bestanden, weil die Behandlung von optionalen Werten in SPARQL und LEFT JOINs in SQL in Details unterschiedlich ist.
- `ex11.2.3.1_1` fällt durch, da die Verwendung von Zeitzonen in SQL sehr schwierig ist. Dies ist in Abschnitt 5.2.3 beschrieben.
- `ex11.2.3.2_0` schlägt fehl weil es in SQL nicht möglich ist, in Filterausdrücken UNION-Abschnitte zu überspannen.

Die Modultests decken jeden Teilaspekt des neuen SPARQL-Prozessors ab, so dass zu erwarten ist, dass in Zukunft nur wenige Fehler gefunden werden.

6.4 Einsatz des SPARQL-Prozessors

Der datenbankbasierte SPARQL-Prozessor entstand aus dem Bedürfnis unserer Arbeitsgruppe nach einer robusten SPARQL-Unterstützung für den Einsatz in eigenen Anwendungen. Diese Anwendungen waren das Testfeld, in dem der Prozessor von Anfang an im täglichen Einsatz erprobt wurde. Durch die Rückmeldungen konnten bis dahin unentdeckte Fehler gefunden und behoben werden. Weiterhin wurden neue Funktionen entwickelt, die sich beim Einsatz des Prozessors in unseren Anwendungen als wichtig herausstellten.

Zur Zeit kommt der datenbankbasierte SPARQL-Prozessor bereits in einigen Anwendungen zum Einsatz, die nachfolgend kurz vorgestellt werden.

6.4.1 Ontowiki

Ontowiki[ADR06] ist ein semantisches Wiki, welches vollständig auf RDF-Daten arbeitet.

Von Anfang an verwendete Ontowiki RAP, um Daten zu im- und exportieren. Für den Zugriff auf einen Teil der Daten konnte die API der RAP verwendet werden, bei komplexeren Abfragen musste jedoch direkt per SQL auf die Datenbank zugegriffen werden.

Die jetzt durchgängige Verwendung von SPARQL als Schnittstelle zur Datenbank brachte mehrere Vorteile:

- Die interne API von Ontowiki ist inzwischen sehr sauber, da nicht mehr verschiedene Datenbankzugriffswege benutzt werden müssen.
- Die zur Datenabfrage benötigte Logik in Ontowiki konnte reduziert werden und konzentriert sich nur noch darauf, SPARQL-Anfragen zu erstellen.
- Durch die Verwendung von komplexen SPARQL-Anfragen konnte die Gesamtzahl der für den Seitenaufbau nötigen Datenbankanfragen reduziert werden, was zu schnelleren Reaktionszeiten führte.
- Eine weitere Geschwindigkeitssteigerung erfuhr das Programm durch die Verwendung eines eigenen Ergebnisrenderers. Ontowiki verwendet intern andere Ressourcenbasisklassen als RAP, so dass Ergebnisse bisher immer konvertiert werden mussten. Mit dem Ergebnisrenderer entfällt dieser Schritt, was sich vor allem bei Anfragen mit vielen Ergebnissen positiv bemerkbar macht.
- Mit der ausschließlichen Verwendung von SPARQL zum Zugriff auf die Daten ist es möglich, andere SPARQL-Endpunkte als Datenquelle für Ontowiki zu verwenden.

6.4.2 LDAP2SPARQL

LDAP2SPARQL [6] stellt eine LDAP-Schnittstelle für SPARQL Endpunkte bereit. Mit seiner Hilfe ist es möglich, mit LDAP-fähigen Applikationen auf RDF-Datenbanken zuzugreifen.

An den Dienst gestellte LDAP-Anfragen werden analysiert und in SPARQL-Anfragen umgewandelt. Diese werden an den SPARQL-Prozessor der RAP übergeben. Das Ergebnis wird in das LDAP-Format umgewandelt und an den anfragenden Client zurückgegeben.

[6] `http://aksw.org/Projects/LDAP/Backend`

Bei der im Rahmen dieser Diplomarbeit durchgeführten Erweiterung des SPARQL-Pakets der RAP wurde unter anderem der SPARQL-Parser verbessert, so dass jetzt alle vom LDAP-Konverter gestellten Anfragen ohne Probleme ausgeführt werden. Weiterhin ist es durch den neuen Prozessor möglich, trotz sehr großen Datenbeständen in sehr kurzer Zeit auf die LDAP-Anfragen zu antworten.

6.4.3 Vakantieland

Vakantieland [7] ist eine Sammlung an Informationen für Touristen in den Niederlanden und erhält alle Daten aus einer RDF-Datenbank.

Um beim Datenzugriff unabhängig von einem bestimmten Datenbanksystem zu sein, wird SPARQL verwendet. Anfangs wurde der speicherbasierte SPARQL-Prozessor der RAP verwendet. Dies führte dazu, dass man bei größeren Datenmengen teilweise über eine halbe Minute auf den Seitenaufbau warten musste.

Durch den Einsatz des neuen SPARQL-Prozessors sank die Reaktionszeit auf unter eine Sekunde. Weiterhin können jetzt intern komplexere Anfragen benutzt werden, die vom alten Prozessor nicht unterstützt wurden.

[7] `http://www.vakantieland.nl`

Kapitel 7

Erweiterungen zu SPARQL

Während der Arbeit am Prozessor und der Umstellung von Applikationen wie zum Beispiel Ontowiki [ADR06] auf SPARQL wurden einige Schwachstellen und fehlende Funktionen im aktuellen Entwurf der Abfragesprache entdeckt. Vor allem im Vergleich mit SQL, der über zwanzig Jahre alten generischen Query Language, steckt SPARQL noch in den Kinderschuhen. Durch zunehmende Verbreitung werden auch an SPARQL zunehmend größere Anforderungen gestellt werden. Vor allem SQL wird als weit verbreiteter und akzeptierter Standard zum Vergleich herangezogen werden.

Die Idee liegt nahe, SPARQL an SQL anzugleichen - ohne dabei die Grundidee aus den Augen zu verlieren, nämlich dass SPARQL für die Arbeit auf Graphen und Tripeln optimiert ist. Sieht man sich jetzt die Liste meiner Verbesserungsvorschläge an, wird klar, wohin die Richtung gehen sollte: SPARQL als SQL für Graphen und Tripel.

Dazu schlage ich die Erweiterung von SPARQL um folgende Dinge vor:

- Prepared Statements
- Getrennte Modelle
- Operatoren in SELECT und CONSTRUCT-Patterns
- Mathematische Funktionen
- Umbenennen von Variablen
- Aggregate und Gruppierung
- Verkettung von Anfragen

7.1 Prepared Statements

Prepared Statements sind in SQL ein Weg, um das Senden und Ausführen von ähnlichen Anfragen zu beschleunigen. Sie sind zwar kein offizieller ANSI-SQL-Standard, aber nahezu alle Datenbankhersteller implementieren sie. Dies lässt vermuten, dass in einer der nächsten SQL-Versionen prepared statements standardisiert werden.

Der Entwickler definiert die "Vorlage" einer Anfrage mit Platzhaltern für Werte, die später eingesetzt werden sollen. Diese Vorlage wird an den SQL-Server geschickt. Ist dies getan, sendet der SQL-Client nur noch eine Liste mit Werten, die anstelle der Platzhalter verwendet werden sollen. Der Server führt den Query daraufhin aus.

Diese Methode hat folgende zentrale Vorteile:

- Sobald die Anfrage an den Server geschickt wurde, kann dieser die Anfrage optimieren und deren Ausführung planen. Muss man mehrere ähnliche Anfragen stellen, wird die SQL-Anfrage folglich nur ein einziges Mal geparst und optimiert. Für die Ausführung müssen nur noch die entsprechenden Werte eingesetzt werden.

- Weiterhin muss nicht mehr die komplette Anfrage an den Server geschickt werden, sondern nur noch die Liste mit Werten der Platzhalter. Dies reduziert die Netzwerklast.

- *SQL-Injection*-Attacken werden verhindert [WMK06], weil Werte nicht mehr aus Versehen ohne Quotes und/oder unescaped verwendet werden.

Sobald mehrere ähnliche Anfragen gemacht werden müssen, sind Prepared Statements von Vorteil. Je größer die Zahl der Anfragen ist, desto mehr Rechenzeit wird gespart.

7.1.1 Implementierung in SQL

Fast alle aktuellen SQL-Serverimplementierungen unterstützen Prepared Statements. Da diese jedoch noch nicht im SQL-Standard aufgenommen wurden, wird je nach Implementierung eine von zwei Notationen angeboten:

1. *Positionsbasierte anonyme Platzhalter* werden zum Beispiel in MySQL verwendet [1]. In die Anfrage wird als Platzhalterzeichen ein Fra-

[1] `http://dev.mysql.com/doc/refman/5.0/en/sqlps.html`

gezeichen `?` eingesetzt. Da alle Platzhalter dasselbe Zeichen verwenden, muss der Entwickler beim Ausführen des Prepared Statements die Werte der Platzhalter in der richtigen Reihenfolge angeben.

2. Als Alternative dazu werden *Benannte Platzhalter* unter anderem in Oracle verwendet [2]. Anstatt eines generischen Fragezeichens für alle werden Platzhalter mit einem Doppelpunkt und einem darauf folgenden eindeutigen Namen notiert: `:name`. Der Vorteil dieser Variante liegt darin, dass der Entwickler sich nicht merken muss, in welcher Reihenfolge die Parameter in der Anfrage verwendet werden (die bei händischen Anfrageoptimierungen oder -änderungen oft verändert wird). Weiterhin müssen mehrfach verwendete gleiche Werte nur einmal angegeben und auch nur einmal vom Client zum Server übertragen werden, was die Belastung des Netzwerks nochmals verringert.

Für den Entwickler als auch für die Performance ist Variante 2 optimal.

7.1.2 Prepared Statements in SPARQL

Als Notationsvariante für Prepared Statements schlage ich wegen deren Vorteile die Verwendung von benannten Platzhaltern vor. Durch die Verwendung des Doppelpunktes in SPARQL als Trennzeichen zwischen Namensraumspräfix und Bezeichner ist es allerdings nötig, anstatt des von SQL bekannten `:` ein alternatives Zeichen zu verwenden. Da Platzhalter im Prinzip - wenn auch auf einer anderen Ebene - Variablen einer Anfrage sind, liegt es nahe, das Fragezeichen als Variablenpräfix zu verwenden. Um Platzhalter von normalen Variablen unterscheiden zu können, werden zwei Fragezeichen verwendet.

Die SPARQL-Anfrage

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?o
  { ?s foaf:name ?o }
LIMIT 3 OFFSET 2
```

wird durch die Ersetzung des Prädikates durch einen Platzhalter zu folgender Anfrage:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?o
```

[2] `http://download-east.oracle.com/docs/cd/B19306_01/appdev.102/b14250/oci05bnd.htm`

```
  { ?s ??type ?o }
LIMIT 3 OFFSET 2
```

Platzhalter sind natürlich auch in `CONSTRUCT`-Patterns und bei `LIMIT` und `OFFSET` verwendbar.

7.1.3 Erfahrungen mit der Testimplementierung

Im Rahmen meiner Arbeit implementierte ich Unterstützung für Prepared Statements im neuen Prozessor. Ich benutzte den Profiler, um Aussagen über die Verteilung der Rechenzeit bei einer SPARQL-Abfrage zu bekommen.

Es zeigt sich, dass - bereinigt um die für das Parsen der PHP-Dateien benötigte Zeit - das Parsen der SPARQL-Query die meiste Zeit verbraucht:

Model::_prepareSparql	16 ms
Model::_parseSparqlQuery	172 ms
SparqlEngineDb::queryModel	101 ms

Abbildung 7.1: Profilingergebnisse für das Absetzen einer SPARQL-Anfrage

Das Parsen der Anfrage dauert mit Abstand am längsten, sogar länger als die eigentliche Ausführung auf dem Datenbankserver. Dies liegt vor allem daran, dass der `SparqlParser` in PHP geschrieben ist, während der für die Ergebnisberechnung zuständige SQL-Server in C programmiert wurde.

Durch die Verwendung von Prepared Statements kann man also die zur Ausführung benötigte Zeit um einen relativ großen Anteil verringern. Dies erlaubt es auch bei zeitkritischen Anwendungen, mehr Anfragen auszuführen und komplexere Programme zu schreiben, die trotzdem zeitnah auf den Benutzer reagieren können.

7.2 Getrennte Modelle

Es gibt verschiedene Methoden in SQL, Daten voneinander zu trennen:

- *Verschiedene Datenbanken*: Auf dem SQL-Server werden mehrere Datenbanken angelegt. Ein angemeldeter Benutzer kann nur auf einer Datenbank gleichzeitig und nicht datenbankübergreifend arbeiten. Die Umschaltung erfolgt mittels des Befehls `USE datenbankname;`. Dies ist die stärkste Form der Datentrennung.

- *Verschiedene Tabellen*: Innerhalb einer Datenbank werden mehrere Tabellen erstellt. Daten dieser Tabellen können miteinander auch in einer Abfrage kombiniert werden. Da die Daten aber in unterschiedlichen Tabellen liegen, lassen sie sich unabhängig auslesen.

- *Schlüsselspalte in einer Tabelle*: Innerhalb einer einzigen Tabelle gibt es eine Spalte, die die Zuordnung der Datenreihe zu einer Gruppe definiert. Bei Anfragen werden ohne explizite Angabe der Gruppe die Daten aus allen Gruppen verwendet. Dies ist die schwächste Form der Datentrennung; die Vermischung kann nur explizit aufgehoben werden.

SPARQL selbst unterstützt eine sehr schwache Trennung von Daten durch *Named Graphs*. Dies ist zu vergleichen mit der Schlüsselspaltenlösung in SQL, da ohne explizite Angabe der zu verwendenden Graphen alle Daten verwendet werden. Verschiedene SPARQL-Implementierungen bieten bereits die Trennung der RDF-Daten nach *Modellen* an. Dies ist leider nicht standardisiert, so dass jede Implementierung ihren eigenen Weg geht, was den Wechsel zwischen den verschiedenen APIs schwer macht.

Zur Lösung des Problems und weiteren Standardisierung schlage ich die Adaption des von SQL bekannten Datenbankschemas vor. Zwischen den Modellen/Datenbanken soll mit dem Kommando `USE` gewechselt werden können.

```
USE <modelURI>;
```

Abbildung 7.2: Auswahl eines Modells

Diese Möglichkeit erlaubt dem Benutzer, die Daten strikter voneinander zu trennen, so dass nicht zusammengehörige Informationen niemals miteinander verwendet werden können. Weiterhin erlaubt dies auch eine einfache Implementierung von Zugriffskontrollsystemen in SPARQL-Servern.

7.2.1 Auflistung der vom Server vorgehaltenen Modelle

Wenn mehrere klar getrennte und durch Modell-URIs zu identifizierende Datenmodelle durch SPARQL unterstützt werden, muss es einen Weg geben, programmatisch eine Liste der Modelle zu erhalten. Die Anfrage sollte sich nahtlos in das existierende Abfragekonzept einordnen; Modelle soll-

ten durch RDF-Tripel repräsentiert werden und durch normale SPARQL-Anfragen gelistet werden können.

Dafür könnte die Bereitstellung eines speziellen Systemmodells genutzt werden, welches ähnlich dem Informationsschema relationaler Datenbanken funktioniert:

```
USE <http://example.com/sysmodel>.

PREFIX sysmodel: <http://example.com/sysmodel-schema/>
SELECT ?model
WHERE { ?x rdf:type sysmodel:model }
```

Abbildung 7.3: Auflisten aller Modelle eines SPARQL-Servers

7.3 Operatoren in SELECT und CONSTRUCT Patterns

`SELECT` und `CONSTRUCT`-Anfragen sind zur Zeit darauf beschränkt, nur komplette Werte für Subjekt, Prädikat und Objekt zurückzugeben. Bei einigen Anwendungen ist es nötig, sich Teile der Daten oder die Daten in veränderter Form zurückgeben zu lassen.

Ein Beispielfall wäre eine Software, die den Blick auf einen RDF-Speicher gewährt und ein lokalisiertes Interface hat. Um den Benutzer die Daten in einer bestimmten Sprache anzeigen zu können, muss die Software herausfinden, welche Sprachen überhaupt vorhanden sind. Bis jetzt ist es nicht möglich, eine Liste aller verwendeten Datentypen oder Sprachen mit einem SPARQL-Query zu bekommen (außer man lässt sich alle Daten zurückgeben und filtert auf Clientseite).

Das Problem kann gelöst werden, indem man Operatoren auch bei der Variablenselektion von `SELECT` und der Tripeldefinition bei `CONSTRUCT` Patterns zulässt:

```
SELECT DISTINCT lang(?o)
WHERE { ?s ?p ?o }
```

Abbildung 7.4: Auflisten aller Sprachen eines Modells

Durch diese Erweiterung wird es nötig, die Umbenennung von Variablen zu unterstützen. Siehe dazu Abschnitt 7.5.

7.4 Mathematische Funktionen

Die SPARQL-Empfehlung erlaubt "extensible value testing" (erweiterbare Wertprüfungen) durch die Möglichkeit für SPARQL-Implementierungen, eigene Operatoren/Funktionen zu definieren. Es wird das Beispiel einer Funktion `aGeo:distance` gezeigt, die es erlaubt, die Distanz zweier Orte voneinander zu bestimmen und damit zum Beispiel alle Städte innerhalb eines gewissen Radius um eine gegebene Stadt zu erhalten (Beispiel 7.5).

```
PREFIX aGeo: <http://example.org/geo#>

SELECT ?neighbor
WHERE { ?a aGeo:placeName "Grenoble" .
        ?a aGeo:lat ?axLoc .
        ?a aGeo:long ?ayLoc .

        ?b aGeo:placeName ?neighbor .
        ?b aGeo:lat ?bxLoc .
        ?b aGeo:long ?byLoc .

        FILTER ( aGeo:distance(
            ?axLoc, ?ayLoc, ?bxLoc, ?byLoc
        ) < 10 ) .
      }
```

Abbildung 7.5: Auflistung alle Städte um Grenoble

Dieses Beispiel und die Erfahrung zeigen, dass vor allem mathematische Funktionen in Anfragen benötigt werden. Bevor die verschiedenen Implementierungen eigene Präfixe oder Namen einführen, sollten grundlegende mathematische und trigonometrische Funktionen im SPARQL-Standard definiert werden. Dadurch nimmt die Notwendigkeit nach Datenfilterung auf Clientseite und die Implementierung proprietärer Funktionen dramatisch ab.

Die proprietäre Funktion `aGeo:distance` im Städtebeispiel lässt sich

durch die kombinierte Verwendung der Quadratwurzel `sqrt`, Potenzfunktion `pwr` und Subtraktion ersetzen (Beispiel 7.6).

```
PREFIX aGeo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
SELECT ?neighbor
WHERE { ?a aGeo:placeName "Grenoble".
        ?a aGeo:location ?axLoc. ?a aGeo:location ?ayLoc.
        ?b aGeo:placeName ?neighbor.
        ?b aGeo:location ?bxLoc. ?b aGeo:location ?byLoc.
        FILTER ( sqrt(
           pwr(?axLoc - ?bxLoc, 2)
           + pwr(?ayLoc - ?byLoc, 2)
           ) < 10 ) .
      }
```

Abbildung 7.6: Auflisten von Städten nahe Grenoble durch simple Mathematik

7.5 Umbenennen von Variablen

Mit der Einführung von zusammengesetzten und komplexen Rückgabewerten bei `SELECT`-Anfragen (7.3, 7.4, 7.6) wird es nötig, Rückgabevariablen umzubenennen. Dies verbessert die Lesbarkeit auf Client-Seite ungemein.

Auch bei dieser Erweiterung sollte SPARQL die in SQL bekannte Syntax durch `Ausdruck AS Name` übernehmen (Beispiel 7.7).

```
SELECT DISTINCT substr(str(?o), 0, 10) as ?name
WHERE { ?s ?p ?o }
```

Abbildung 7.7: Umbenennen einer Variable

7.6 Aggregate und Gruppierung

Gruppierungs- und Aggregatfunktionen sind ein weiterer Punkt auf der für Datenbanksysteme wichtige, aber von SPARQL nicht unterstützten Features.

Aggregate erlauben es, aus einer Liste von Zahlen die Extrema herauszusuchen (`MIN`, `MAX`), sie zu zählen (`COUNT`) und andere statistisch relevante Daten zu erhalten (`MEAN`, `AVG`, ...).

Gruppierung erlaubt es, die Aggregatfunktionen nicht nur auf die gesamte Liste an Daten anzuwenden, sondern auch auf Teile der Liste mit einem gemeinsamen Wert.

7.6.1 Gruppierung in Virtuoso

Das Datenbanksystem Virtuoso hat seine eigene proprietäre Erweiterung für Gruppierung und Aggregate in SPARQL (Beispiel 7.8) [3]. Diese ist sehr limitiert und unterstützt nur implizite Gruppierung.

```
SELECT count (?p) sum (?o) count (distinct ?o)
WHERE {?s ?p ?o};
```

Abbildung 7.8: Aggregate und Gruppierung in Virtuoso

Probleme der Virtuoso-Implementierung sind folgende:

- Variablenauswahl und Gruppenerstellung sind vermischt, wodurch die Anfragen schwerer zu verstehen sind.
- Es ist nicht möglich Variablen zu verwenden, ohne sie implizit zu einer Gruppe zu machen.
- Gruppierung funktioniert auf globaler Ebene, nicht für einzelne Graph Patterns.
- Operatoren können nicht als Gruppierungsfunktion benutzt werden.

Der Verdacht liegt nahe, dass die Implementierung so gemacht wurde, dass möglichst wenig am SPARQL-Parser verändert werden musste und die Funktionalität komplett über den SELECT-Teil der Anfrage abgewickelt werden kann. Dadurch wurde die Verwendbarkeit der Erweiterung stark beschränkt, und oben genannte Einschränkungen treten auf.

[3] `http://docs.openlinksw.com/virtuoso/rdfsparqlaggregate.html`

7.6.2 Vorschlag für besseres Gruppieren in SPARQL

Durch die Verwendung einer Syntax ähnlich SQL würde Anfragen zum einen lesbarer machen sowie die Nachteile des Impliziten Gruppierens aufheben.

Es muss noch entschieden werden, ob `GROUP BY` nur auf globaler Ebene oder auch lokal innerhalb der Anfrage erlaubt sein soll, um damit die Gruppierung innerhalb von Subgraphen zu ermöglichen. Das bereits existierende `ORDER BY` ist nur global möglich, obwohl die Sortierung für Subgraphen durchaus Vorteile und Verwendungszweck hätte.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name, count(?mbox) as ?count
WHERE { ?x foaf:name ?name.
        ?x foaf:mbox ?mbox. }
GROUP BY ?name
```

Abbildung 7.9: Aggregate und globale Gruppierung mit GROUP BY

7.7 Verkettung von Anfragen

Nach Erweiterung von SPARQL um getrennte Modelle 7.2 und Datenmanipulation [4] sollte es möglich sein, mehrere Anfragen zusammenzufassen. Anwendungsfälle sind der Wechsel zu einem anderen Modell mit gleich darauf folgender normalen Abfrage, und auch das mehrfache Einfügen von Daten, die in einer "Dump"-Datei gespeichert sind.

Da das Semikolon in SPARQL noch keinen speziellen Zweck erfüllt, eignet es sich auch aufgrund seiner Verwendung als Trennsymbol in SQL für diesen Zweck.

[4] `http://esw.w3.org/topic/SparqlUpdateLanguage`

```
USE <http://example.com/myModelURI>;

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE { ?x foaf:name ?name.
        ?x foaf:mbox ?mbox. }
```

Abbildung 7.10: Mehrere Befehle in einer einzigen Anfrage

Kapitel 8

Zusammenfassung und Ausblick

Wie bei jedem Softwareprojekt ist auch die Arbeit am neuen SPARQL-Prozessor der RDF API for PHP bei weitem nicht abgeschlossen. Auch wenn durch das permanente Testen des Prozessors durch Unittests die größtmöglichen Anstrengungen für Kompatibilität und Korrektheit unternommen wurden, werden vor allem beim Betrieb in Randfällen Fehler auftauchen.

Für die Kompatibilität des Prozessors beim Arbeiten mit Datumsangaben verschiedener Zeitzonen sollte noch eine Lösung gefunden werden, die bei entsprechender Unterstützung von Zeitzonen durch den SQL-Server trivial zu implementieren wäre. Der SPARQL-Parser wurde von Tobias Gauß manuell implementiert und von mir um neue Features erweitert. Er erkennt alle bekannten Anfragetypen korrekt, wird die Spezifikation jedoch niemals so vollständig umsetzen wie ein mit einem Parsergenerator automatisch aus der BNF-Darstellung der SPARQL Syntax generierter Parser. Dieser müsste aus Kompatibilitätsgründen mit den beiden SPARQL-Prozessoren jedoch dieselben Rückgabewerte liefern, was einigen Programmieraufwand bedeutet.

Eine weitere Möglichkeit, die Verwendbarkeit des neuen Prozessors mit großen Daten- und Ergebnismengen zu verbessern, wäre es, die Ergebnistupel nur bei direktem Zugriff auf die Objekte von der Datenbank zu übertragen. Startet man zur Zeit eine Anfrage, die 10.000 Datensätze als Ergebnis liefert, so werden zuerst alle Datensätze übertragen und in PHP-Objekte verpackt. Während dies einen schnellen Zugriff auf einmal übertragene Daten erlaubt, ist die zum Erstellen des Ergebnisses benötigte Zeit sehr hoch. Diese Form der Ergebnisrückgabe wurde aus Gründen der Kompatibilität mit dem alten Prozessor gewählt. Ein alternativer zeigerbasierter Ansatz, wie er von fast allen Datenbanksystemen unterstützt wird, würde jedoch

einige Vorteile mit sich bringen.

In Bezug auf die Geschwindigkeit kann der Prozessor trotz guter Werte noch weiter optimiert werden. Denkbar sind zum Beispiel der Einsatz von `LIKE` anstatt `REGEX` bei einfachen Filtern, oder der Einsatz von SQL Prepared Statements bei den mehrfach generierten Anfragen durch den `TypeSorter`.

Den Vergleich mit konkurrierenden SPARQL-Implementierungen braucht der neue Prozessor nicht zu scheuen. In der Evaluation zeigte er sich als robust und schnell, nur geschlagen vom einzigen kommerziellen SPARQL-Server. Neben der Geschwindigkeit ist auch die Standardkonformität des Prozessors durch die Unittests sehr hoch.

Der Einsatz des Prozessors in Applikationen unserer Arbeitsgruppe hat gezeigt, dass er einfach in Anwendungen zu integrieren ist und fehlertolerant arbeitet.

Das Ziel der Diplomarbeit, die RDF API for PHP um eine performante SPARQL-Implementierung zu erweitern, wurde erreicht. Gerade die Arbeit mit mittleren und großen RDF-Datenbanken ist jetzt auch im Mehrbenutzerbetrieb mit minimaler Arbeitsspeicherbelastung und hoher Geschwindigkeit möglich. Durch die Integration in die freie RAP ist es vielen Webapplikationen möglich, eine semantische Datenhaltung in Anspruch zu nehmen.

Anhang A

Kurzzusammenfassung

Die Vision des Internets als weltumspannendes Informationsnetz ist zur Wirklichkeit geworden. Um die Idee des weltweiten *Daten*netzes wahr werden zu lassen, wurde das *Resource Description Framework* RDF entwickelt. Die dieses Netz speisenden Daten sind zu großen Teilen in Datenbanken abgelegt.

Eine Abfragesprache für RDF-Daten ist die vom World Wide Web Consortium im Standardisierungsprozess befindliche *SPARQL Protocol And Query Language*. Die RDF API for PHP ist eine in der Skriptsprache PHP programmierte Funktionsbibliothek zum Arbeiten mit Daten im RDF-Format. Ziel dieser Arbeit ist es, eine effiziente, datenbankgestützte Implementierung von SPARQL auf Basis der RAP-Bibliothek zu schaffen.

Nach den Grundlagen des semantischen Netzes werden neben SPARQL und der RDF API for PHP auch konkurrierende Bibliotheken mit SPARQL-Unterstützung vorgestellt. Der bereits existierende, speicherbasierte SPARQL Prozessor des RAP-Frameworks wird analysiert und dessen Probleme identifiziert. Im Folgenden werden die Anforderungen und die Implementierung des neuen datenbankgestützten Prozessors im Detail beschrieben. Abschließend kommt es zur Evaluation der Geschwindigkeit des neuen Prozessors im Vergleich zu anderen SPARQL-fähigen Datensystemen.

Während der täglichen Arbeit mit SPARQL hat sich herausgestellt, dass sich das Protokoll aufgrund einiger Unzulänglichkeiten noch nicht vollständig für den praktischen Einsatz eignet. Aus diesem Grund werden abschließend Erweiterungen zu SPARQL vorgestellt, die sich in der Praxis als hilfreich erwiesen haben.

Der im Rahmen dieser Diplomarbeit umgesetzte SPARQL Prozessor erweist sich als robust und schnell. Er liegt trotz der Umsetzung mit einer Skriptsprache in der gleichen Leistungsdimension wie andere State-of-the-Art-Implementierungen und ist diesen häufig sogar überlegen. Er wird

bereits in mehreren Anwendungen produktiv eingesetzt und ist als Open-Source-Projekt einer internationalen Anwender- und Entwicklergemeinde zugänglich.

Literaturverzeichnis

[ADR06] Sören Auer, Sebastian Dietzold, and Thomas Riechert. OntoWiki - A Tool for Social, Semantic Collaboration. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, volume 4273 of *Lecture Notes in Computer Science*, pages 736–749. Springer, 2006.

[BB06] Dave Beckett and Jeen Broekstra. SPARQL Query Results XML Format. W3c candidate recommendation, World Wide Web Consortium (W3C), April 2006.

[BCF+07] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, Januar 2007.

[Bec04a] Dave Beckett. Turtle - Terse RDF Triple Language. Website, Januar 2004. `http://www.dajobe.org/2004/01/turtle/`.

[Bec04b] David Beckett. RDF/XML Syntax Specification. W3c recommendation, RDF Core Working Group, World Wide Web Consortium, 2004.

[BEK+00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. W3C Note NOTE-SOAP-20000508, World Wide Web Consortium, May 2000.

[Biz04] Chris Bizer. RAP (RDF API for PHP). Website, November 2004. `http://www.wiwiss.fu-berlin.de/suhl/bizer/rdfapi/`.

[BL05] Tim Berners-Lee. Notation3: Logic and Rules on RDF. Technical report, Word Wide Web Consortium, 2005.

[BLFM05] Tim Berners-Lee, R. Fielding, and L. Masinter. RFC 3986, Uniform Resource Identifier (URI): Generic Syntax, 2005.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[BPSM98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation REC-xml-19980210, World Wide Web Consortium, 1998.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. Technical report, World Wide Web Consortium, 1999.

[Cla06] Kendall Grant Clark. SPARQL Protocol for RDF. Technical report, W3C, 2006.

[Cro06] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). Internet informational RFC 4627, July 2006.

[DS05] M. Duerst and M. Suignard. RFC 3987, Internationalized Resource Identifiers (IRIs), January 2005.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol - HTTP/1.1*. Network Working Group, The Internet Society, 1999. RFC 2616.

[Gau06] Tobias Gauß. SPARQL Query Language for RDF - Implementierung für das RAP Toolkit. Master's thesis, Freie Universität Berlin, 2006.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[Ins89] American National Standards Institute. *Information Systems - Database Language - SQL*. ANSI X3.135-1992. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1989.

[Kar01] Greg Karvounarakis. RDF Query Language, 2001.

[MM02] F. Manola and Eric Miller. RDF Primer, 2002.

[Moz01] MozillaFoundation. XML User Interface Language (XUL). Website, 2001. `http://www.mozilla.org/projects/xul/`.

[MSR02] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In Ian Horrocks and James A. Hendler, editors, *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 423–435. Springer, 2002.

[OO02] Mike Olson and Uche Ogbuji. Versa. Website, 06 2002.

[PNNJ05] A. Powell, M. Nilsson, A. Naeve, and P. Johnston. Dublin Core Metadata Initiative - Abstract Model, 2005. White Paper.

[Pru04] Eric Prud'hommeaux. RDF Query Survey. Technical report, W3C, 2004.

[PS06] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF (Working Draft). W3c working draft, World Wide Web Consortium (W3C), 2006.

[Ree79] Trygve M. H. Reenskaug. Models - Views - Controllers, December 1979. `heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf`.

[RM96] Paul Resnick and James Miller. PICS: Internet Access Control Without Censorship. *Communications of the ACM*, 39(10):87–93, 1996.

[Sea04] Andy Seaborne. RDQL - A Query Language for RDF (Member Submission). Technical report, W3C, January 2004.

[Uni96] Unicode Consortium. *The Unicode Standard, Version 2.0.* Addison-Wesley, Reading, MA, 1996.

[WA07] Christian Weiske and Sören Auer. Implementing SPARQL Support for Relational Databases and Possible Enhancements. In *Lecture Notes in Informatics*, volume 113. Gesellschaft für Informatik e.V., 2007.

[WMK06] Ke Wei, M. Muthuprasanna, and Suraj Kothari. Preventing SQL Injection Attacks in Stored Procedures. In *ASWEC*, pages 191–198. IEEE Computer Society, 2006.

Abbildungsverzeichnis

Anhang B

Unittestergebnisse

In der folgenden Tabelle sind alle verfügbaren 151 Unittests aufgelistet, die in der SPARQL-Kategorie der RAP Testsuite verwendet werden. Der neue Prozessor besteht grün hinterlegte Tests; rot markierte Tests zeigen Fehler bei der Ausführung.

dawg-tp-01	expr-3	q-base-prefix-2	test-3-07
dawg-tp-02	query-bev-1	q-base-prefix-3	test-4-01
dawg-tp-03	query-bev-2	q-base-prefix-4	test-4-02
dawg-tp-04	query-bev-5	q-base-prefix-5	test-4-03
ex2-1a	query-bev-6	q-construct-1	test-4-04
ex2-2a	q-str-1	q-construct-2	test-4-05
ex2-3a	regex-query-001	q-reif-1	test-4-06
ex2-4a	regex-query-002	q-reif-2	test-4-07
q-opt-1	regex-query-003	test-0-01	test-5-01
q-opt-2	ex11.2.3.6_0	test-0-02	test-5-02
dawg-query-001	query-bev-3	test-0-03	test-5-03
dawg-query-002	query-bev-4	test-0-04	test-5-04
dawg-query-003	q-str-2	test-1-01	test-6-01
dawg-query-004	q-str-3	test-1-02	test-6-02
q-select-1	q-str-4	test-1-03	test-6-03
q-select-2	q-blank-1	test-1-04	test-7-01
q-select-3	q-datatype-1	test-1-05	test-7-02
query-sort-5	ex11_0	test-1-06	test-7-03
query-sort-4	ex11_1	test-1-07	test-7-04
query-sort-3	ex11.2.3.1_1	test-1-08	test-9-02
query-sort-2	ex11.2.3.1_1	test-1-09	test-B-01
query-sort-1	ex11.2.3.2_0	test-1-10	test-B-02
query-sort-6	q-uri-1	test-2-01	test-B-03
query-sort-4	q-langMatches-1	test-2-02	test-B-04
query-sort-datetime	q-langMatches-2	test-2-03	test-B-05
LimitOff_1	q-langMatches-3	test-2-04	test-B-08
LimitOff_2	query-eq-1	test-2-05	test-B-09
LimitOff_3	query-eq-2	test-2-06	test-B-10
bound1	query-eq-3	test-2-07	test-B-11
ex3	query-eq-4	test-2-08	test-B-12
ex11.2.3.1_0	query-eq-5	test-2-09	test-B-13
ex11.2.3.2_1	query-eq-graph-1	test-2-10	test-B-15
ex11.2.3.3_0	query-eq-graph-2	test-3-01	test-B-17
ex11.2.3.4_0	query-eq-graph-3	test-3-02	test-B-18
ex11.2.3.5_0	query-eq-graph-4	test-3-03	test-B-19
ex11.2.3.7_0	customUnion1	test-3-04	test-B-20
expr-1	customUnion2	test-3-05	ask-01
expr-2	q-base-prefix-1	test-3-06	count-02

Abbildung B.1: Liste mit allen Unittests

Anhang C

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Ort Datum Unterschrift