SPARQL Anfragesprache für RDF-Daten

Michael Zeising

Juli 2008

Seminar in Informatik: Semantic Web
Lehrstuhl für Angewandte Informatik IV – Datenbanken und Informationssysteme

Einführung

Die Repräsentation von Wissen in einem RDF-Graphen bildet den Grundstein des *Semantic Web*. Um wiederum Informationen aus diesem Graphen ziehen zu können, ist eine Abfragesprache notwendig. Hierzu entstanden in den letzten rund zehn Jahren verschiedenste Ansätze für formale Sprachen, von sehr SQL-ähnlichen wie *RDQL* und *Squish* bis hin zu pfadbasierten Sprachen wie *Versa*. Die deklarativen Sprachen mit SQL-ähnlicher Syntax waren dabei stets die mit Abstand erfolgreichsten. [Dod05] Das *World Wide Web Consortium* hat sich um die Standardisierung bemüht und im April 2006 eine Sprache namens SPARQL vorgeschlagen. Es handelt sich um eine deklarative Sprache, die viele Konzepte seiner Vorgänger rdfDB, RDQL und SeRQL in sich vereint. [McC05]

SPARQL wird von der *RDF Data Access Working Group* des W3C entwickelt. Der Standard wurde erstmals im April 2006 als *Candidate Recommendation* veröffentlicht aber wegen zu vieler offener Fragen im Oktober 2006 wieder zum *Working Draft* herabgestuft [PSa, PSb]. Im Juni 2007 wurde SPARQL wiederum zur *Candidate Recommendation* [Hera], im November 2007 zur *Proposed Recommendation* und im Januar 2008 wurde es dann schließlich zur *Recommendation*, also zu einem Standard [Herb]. Dieser Status verdeutlicht, dass SPARQL nun den selben Reifegrad besitzt wie etwa XML oder HTML.

Die Spezifikation besteht aus drei Teilen. Den ersten und umfangreichsten Teil *SPARQL Query Language for RDF* bildet die Abfragesprache selbst. Im Teil *SPARQL Query Results XML Format* wird das Format der Abfrageergebnisse festgelegt. Wie der Name vermuten lässt, wird ein XML-Dialekt beschrieben. In *SPARQL Protocol for RDF* wird ein Protokoll für den Austausch von Abfragen und Ergebnissen beschrieben. Hier wird im Wesentlichen ein Web-Service auf der Basis der WSDL 2.0 spezifiziert. Im folgenden soll ein Einblick in den ersten Teil, die Abfragesprache SPARQL gegeben werden.

Grundlegende Konzepte

Graphmuster

SPARQL ist eine deklarative Sprache. Statt also im Graphen zu navigieren, wird ein *Graphmuster* formuliert. Ein solches Graphmuster unterscheidet sich nur dadurch von einem gewöhnlichen RDF-Graphen, als dass Subjekte, Prädikate und Objekte durch Variablen ersetzt werden können. Ein Graphmuster passt auf einen Teilgraphen, wenn die Ausdrücke aus diesem Teilgraphen durch die Variablen ersetzt werden können. Die Lösung ist eben dieser Teilgraph (vgl. Abb. 1). [PSc]

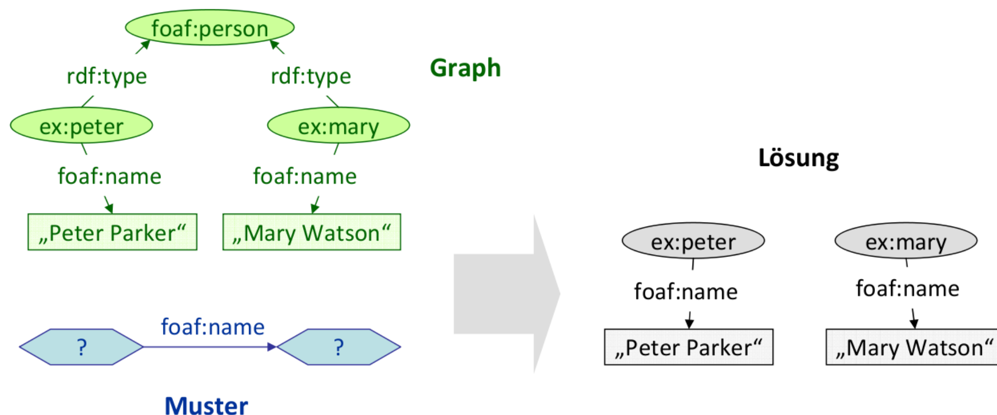

Abbildung 1: Illustration des Mustervergleichs

Um diese Graphmuster formulieren zu können wird die kompakte Turtle-Syntax für RDF-Graphen verwendet und um Variablen erweitert. Eine Variable beginnt mit einem ? und hat einen Bezeichner. Alle drei Bestandteile eines Tripels können nun durch Variablen ersetzt werden. Der Bezeichner ist beliebig aber eindeutig: gleiche Bezeichner repräsentieren den selben Knoten.

Eine einfache Anfrage

Das Graphmuster bildet nun die Grundlage für eine `SELECT`-Anfrage.

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

Mit dem Schlüsselwort `PREFIX` lassen sich Namensräume definieren. Zur besseren Lesbarkeit können dann statt den vollständigen URIs Abkürzungen vergleichbar zu den XML *QNames* verwendet werden[1]. Die `SELECT`-Klausel stellt die Projektion dar. Hier werden die Variablen ausgewählt, die im Ergebnis der Anfrage auftauchen sollen. Wie in SQL ist auch hier ein * erlaubt. Es werden dann alle Variablen ausgewählt die im Muster auftauchen. In der `WHERE`-Klausel wird schließlich das Graphmuster angegeben. Bei der Auswertung der Anfrage wird das Muster wie eine Schablone über alle Teilgraphen gelegt. Passt das Muster auf einen Teilgraphen, werden die Variablen *gebunden*. Das Ergebnis einer `SELECT`-Anfrage ist also eine Menge von Bindungen. Sie kann am einfachsten in einer Tabelle dargestellt werden. Die Spalten entsprechen den Variablen, die Zeilen den entstandenen Bindungen.

[1] Für bessere Lesbarkeit werden in allen folgenden Beispielen die `PREFIX`-Klauseln weggelassen.

person	name
<http://www.example.org/peter>	"Peter Parker"
<http://www.example.org/mary>	"Mary Watson"

Wie in Turtle kann auch im SPARQL-Graphmuster das Schlüsselwort a als Alternative für das Prädikat http://www.w3.org/1999/02/22-rdf-syntax-ns#type verwendet werden. Wir wollen ein weiteres Beispiel für eine Anfrage betrachten. Als Basis soll folgender Graph dienen:

```
ex:peter a foaf:Person ;
         foaf:name "Peter Parker" .
ex:mary  a foaf:Person ;
         foaf:gender "female" ;
         foaf:interest <http://www.w3.org/TR/rdf-sparql-query/> ;
         foaf:knows ex:peter .
```

Es soll nun nach den Namen aller weiblichen Personen gesucht werden, die Peter Parker kennen und sich für SPARQL interessieren. Die Abfrage lautet also:

```
SELECT ?name
WHERE {
  ?person a foaf:Person .
  ?person foaf:name ?name .
  ?person foaf:knows ex:peter .
  ?person foaf:interest <http://www.w3.org/TR/rdf-sparql-query/> .
}
```

Ohne Zweifel handelt es sich bei SPARQL um eine formale Sprache. Man erkennt aber an diesem Beispiel, dass im RDF-Modell bereits wesentlich natürlichere Abfragen erfolgen können, als es auf einem relationalen Modell möglich wäre.

Sprachelemente

Nachdem die grundlegende Idee und Funktionsweise der Sprache erläutert wurde, soll nun ein Überblick über ihre wichtigsten Bestandteile und Ausdrucksmittel gegeben werden.

Optionale Muster

Im Graphen aus Abb. 2 sollen die Namen und E-Mail-Adressen aller Personen erfragt werden.

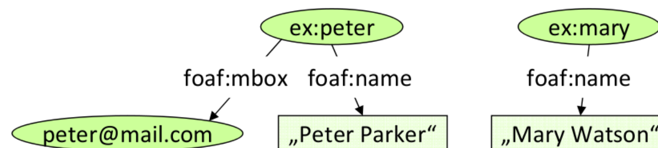

Abbildung 2: Datengraph zur Illustration von optionalen Mustern

Ein naiver Ansatz könnte also lauten:

```
SELECT ?name ?mail
WHERE {
  ?person rdf:type foaf:Person .
  ?person foaf:name ?name .
  ?person foaf:mbox ?mail .
}
```

Die Ontologie schreibe nun für alle Personen einen Namen vor, nicht aber eine E-Mail-Adresse. Man muss also in Betracht ziehen, dass das Muster `?person foaf:mbox ?mail .` nicht auf alle Personen im Graphen passt. Personen zu denen keine Adresse bekannt ist, werden also nicht im Ergebnis dieser Anfrage auftauchen.
Um nun aber alle Personen im Graphen abzudecken, muss das entsprechende Muster als optional deklariert werden. Hierzu dient das Schlüsselwort `OPTIONAL`:

```
SELECT ?name ?mail
WHERE {
  ?person rdf:type foaf:Person .
  ?person foaf:name ?name .
  OPTIONAL { ?person foaf:mbox ?mail . }
}
```

Wenn das optionale Muster nicht zutrifft, wird die entsprechende Variable `mail` nicht gebunden. Es entsteht also folgendes Ergebnis:

name	**mail**
"Peter Parker"	`<peter@mail.com>`
"Mary Watson"	

Das Ergebnis von optionalen Mustern stellt also ein Art zusätzliche Information dar, was bei der Interpretation durch ein Programm unbedingt beachtet werden muss.

Alternative Muster und Duplikatelimination

Es im Graphen in Abb. 3 nach den Personen gefragt werden, die Peter *oder* Mary kennen. Dazu ist es notwendig, die beiden entsprechenden Muster separat auszuwerten und das Ergebnis zu vereinigen.

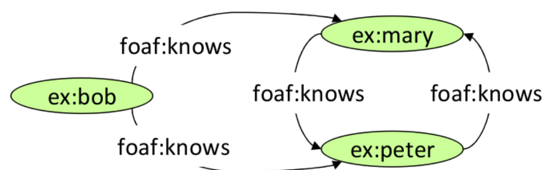

Abbildung 3: Datengraph zur Illustration von alternativen Mustern

Das Schlüsselwort `UNION` setzt dies um:

```
SELECT ?person
WHERE {
  ?person rdf:type foaf:Person .
  ?person foaf:name ?name .
  { ?person foaf:knows ex:peter . }
  UNION
  { ?person foaf:knows ex:mary . }
}
```

person
<http://example.org/peter>
<http://example.org/mary>
<http://example.org/bob>
<http://example.org/bob>

Hierbei ist zu beachten, dass Personen, die Peter *und* Mary kennen, zweimal auftauchen. Duplikate werden also nicht automatisch entfernt, da es sich bekanntlich um eine teure Operation handelt. Zur Elimination wird `DISTINCT` verwendet:

```
SELECT DISTINCT ?person ?name
WHERE {
  ?person rdf:type foaf:Person .
  ?person foaf:name ?name .
  { ?person foaf:knows ex:peter . }
  UNION
  { ?person foaf:knows ex:mary . }
}
```

`DISTINCT` bezieht sich auf die direkt folgende Variable, hier also auf `person`. Sinnvollerweise wird hier somit die URI, also die Identität der Person und nicht etwa ihr Name zur Elimination von Duplikaten verwendet.

Restriktionen

Eine Frage nach den Namen aller Personen, die älter sind als 30 Jahre lässt sich nicht mit einem Graphmuster allein umsetzen. Es ist notwendig Restriktionen für die Variablen festzulegen. Hierzu dient das Schlüsselwort `FILTER`:

```
SELECT ?name
WHERE {
  ?person rdf:type foaf:Person .
  ?person foaf:name ?name .
  ?person ex:age ?age .
  FILTER ( ?age > 30 )
}
```

Neben den aus Programmiersprachen bekannten Vergleichs-, Arithmetik- und Wahrheitsoperatoren stehen auch RDF- bzw. SPARQL-spezifische Funktionen zur Verfügung. Tab. 1 zeigt eine Auswahl solcher Funktionen.

`bound(`*var*`)`	liefert `true` falls die Variable gebunden wurde
`isURI(`*var*`)`	Prüfen auf Knotentyp
`isBlank(`*var*`)`	
`isLiteral(`*var*`)`	
`regex(`*var*, *exp*`)`	liefert `true` falls der reguläre Ausdruck in der Variable gefunden wird
`datatype(`*var*`)`	liefert den Datentyp einer Variable als URI

Tabelle 1: Eine Auswahl von SPARQL-spezifischen Funktion, die in `FILTER`-Ausdrücken verwendet werden können.

Folgendes Beispiel demonstriert die Anwendung von `datatype()` und `regex()`. Die Anfrage beschränkt den Datentyp der Variable `name` auf `xsd:string` und lässt nur Zeichenketten zu, die mit `Peter` beginnen.

```
SELECT ?person ?name
WHERE {
  ?person rdf:type foaf:Person .
  ?person foaf:name ?name .
  FILTER ( datatype(?name) == xsd:string )
  FILTER ( regex(?name, "^Peter") )
}
```

Ein passendes Literal wäre in diesem Fall also `"Peter Parker"^^xsd:string`.

Ergebnis als Graph

Bisher wurden nur `SELECT`-Anfragen betrachtet. Das Ergebnis war eine Menge von Bindungen, die sich besonders für eine sequentielle Verarbeitung eignet. Es fällt allerdings schwer aus dieser Form des Ergebnisses die Struktur der Objekte und die Beziehungen unter ihnen zu rekonstruieren. SPARQL bietet daher die Möglichkeit aus den Lösungen des Graphmusters wiederum einen Graphen zu konstruieren. An die Stelle des `SELECT` tritt dazu die `CONSTRUCT`-Klausel, die eine Schablone für den neuen Graphen beschreibt. Die Syntax entspricht der des Graphmusters.
Folgendes Beispiel findet zunächst alle Personen und legt dann einen neuen Graphen an, der für jede Person das Tripel

Person `rdfs:label` *Name* .

enthält.

```
CONSTRUCT {
    ?person rdfs:label ?name .
} WHERE {
    ?person rdf:type  s:Person .
    ?person foaf:name ?name .
}
```

Die Abfrage überführt also einen Teil des Datengraphen in einen anderen Namensraum.

Die CONSTRUCT-Anfrage stellt den wohl mächtigsten Bestandteil der Sprache dar. Um die Möglichkeiten zu demonstrieren, soll kurz gezeigt werden, wie man mit CONSTRUCT prinzipiell zwischen Ontologien vermitteln kann. [Knu06] Personen, die ein Auto fahren seien entsprechend Abb. 4 modelliert.

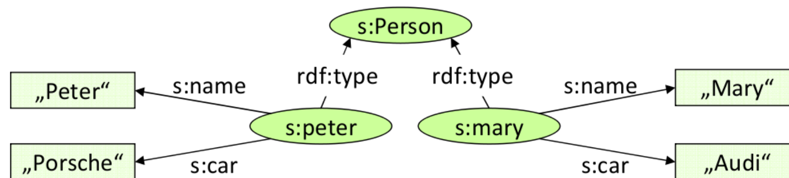

Abbildung 4: Ursprünglicher Datengraph

Es fällt sofort auf, dass das Auto nicht besonders günstig modelliert wurde. Statt als „Attribut“der Person sollte das Auto als Instanz eines Typs repräsentiert werden, um Redundanzen und Inkonsistenzen zu vermeiden. Die Überführung von diesem in ein normalisiertes Modell lässt sich mit einer CONSTRUCT-Anfrage realisieren.

```
CONSTRUCT {
    _:p rdf:type   t:Person .
    _:p t:name     ?personName .
    _:c rdf:type   t:Car .
    _:c rdfs:label ?carName .
    _:p t:drives   _:c .
} WHERE {
    ?person rdf:type s:Person .
    ?person s:name   ?personName .
    ?person s:car    ?carName .
}
```

Das Graphmuster findet zunächst die gezeigte Struktur und bindet die Namen der Personen und Autos. Im neuen Graphen wird nun ein leerer Knoten _:p für die gefundene Person erzeugt und ihr ein Typ und ihr Name zugewiesen. Für das Auto der Person wird wiederum ein leerer Knoten angelegt. Auch ihm wird ein Typ und seine Bezeichnung zugewiesen. Schließlich wird noch ein Tripel eingefügt, das ausdrückt, dass die Person das Auto fährt.

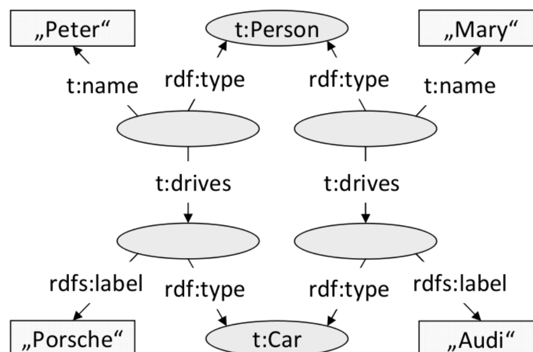

Abbildung 5: Normalisiertes Modell als Ergebnis der `CONSTRUCT`-Anfrage

Das Ergebnis ist in Abb. 5 zu sehen. Es handelt sich um die selbe Aussage in normalisierter Form.

Andere Anfrageformen

ASK

Eine `ASK`-Anfrage stellt fest ob das angegebene Muster zu mindestens einem Ergebnis führt und liefert entsprechend `true` oder `false` als Antwort. So lässt sich zum Beispiel erfragen ob weibliche Personen bekannt sind:

```
ASK ?person
WHERE {
  ?person rdf:type foaf:Person .
  ?person foaf:gender "female" .
}
```

DESCRIBE

Eine `DESCRIBE`-Anfrage beschreibt Ressourcen mit dem Vokabular der Datenquelle. Die Anfrage

```
DESCRIBE <http://example.org/peter>
```

bedeutet also „beschreibe Peter mit deinen Worten“. Dabei ist nicht vorgegeben, in welchem Format die Antwort vorliegen muss.[2] Im Gegensatz zu anderen Anfrageformen muss der Client also den Aufbau des Datengraphen nicht kennen. Somit stellt die `DESCRIBE`-Anfrage einen interessanten Schritt in Richtung Wissensentdeckung durch die Maschine dar. [MS08]

[2] Die Implementierung *Joseki* liefert z.B. alle Tripel, die die Ressource enthalten im N3-Format.

Negation in SPARQL

Die Negation stellt ein konzeptionelles Problem dar, da für die Interpretation von RDF-Daten die *open world assumption* gilt. Dass eine Aussage nicht im Graphen enthalten ist, heißt also nicht, dass sie falsch ist. Aus diesem Grund existiert in SPARQL kein NOT-Operator, mit dem man ein Teilmuster negieren könnte. Um eine RDF-Datenquelle aber als Datenbankersatz nutzen zu können, ist eine Negation unverzichtbar und so schlägt die Spezifikation die Negation durch *negation as failure* vor. Die Aussage $\neg p$ wird also abgeleitet aus dem Fehlschlag p abzuleiten.
Im folgenden Beispiel sollen die Namen aller Personen erfragt werden, zu denen *keine* E-Mail-Adresse bekannt ist:

```
SELECT ?name
WHERE {
  ?person rdf:type foaf:Person .
  ?person foaf:name ?name .
  OPTIONAL { ?person foaf:mbox ?mail . }
  FILTER ( !bound(?mail) )
}
```

Es werden also zunächst alle Personen - mit und ohne Mail-Adresse - gefunden und anschließend diejenigen verworfen, bei denen die Variable `mail` gebunden wurde. Das Ergebnis sind alle Personen, für die das Tripel `(?person, foaf:mbox, ?mail)` nicht im Graphen vorkommt.

Ausblick: Aktuelle Vorschläge zu Erweiterungen

SPARQL/Update

SPARQL ist zunächst eine Nur-Lese-Sprache. Aus der Satzung der Arbeitsgruppe geht hervor, dass sie an einer entsprechenden Erweiterung momentan nicht interessiert ist. [Pru07] Der Grund ist wohl, dass ein schreibender Zugriff eine Zugriffskontrolle notwendig macht. Hierzu müsste also auch die Spezifikation des Zugriffsprotokolls angepasst werden. Zumindest für den sprachlichen Teil gibt es aber bereits Vorschläge wie *SPARQL/Update*. [SM08] Die Erweiterung ist sehr intuitiv und so soll SPARQL/Update Anfragen der Form

```
INSERT INTO <http://example.org/myStore> { ... }
WHERE { ... }

DELETE FROM <http://example.org/myStore> { ... }
WHERE { ... }
```

erlauben.

Pfadbasierte Ausdrucksmittel

Mit SPARQL lassen sich Abfragen die sich auf Muster in den Pfaden des Graphen beziehen nur schwer oder überhaupt nicht umsetzen. Es existieren daher mehrere Ansätze SPARQL

um pfadbasierte Ausdrucksmittel zu erweitern. Einer davon ist *PSPARQL* [Alk07]. Die Idee besteht darin, reguläre Ausdrücke als Prädikat in Graphmustern zu verwenden.
Sind in einem Graphen Orte und die Verkehrsmittel zwischen ihnen modelliert, dann liefert folgende Abfrage in PSPARQL alle direkten und *indirekten* Verkehrsmittel zwischen Berlin und München:

```
SELECT ?transport
WHERE {
  ex:berlin +?transport ex:munich .
}
```

Der Operator + deutet dabei an, dass dieser Teil des Pfades auch wiederholt auftreten kann.

Aggregate

SPARQL enthält keinerlei Aggregatfunktionen wir `COUNT`, `MIN` und `MAX`. Aber auch diese sind für den Gebrauch als Datenbank unerlässlich und so werden sie zum einen für die Spezifikation vorgeschlagen und zum anderen von nahezu allen Implementierungen bereits umgesetzt. Die Bedenken der Arbeitsgruppe sind allerdings nachvollziehbar. Es fragt sich ob es Sinn macht, die Personen im Datengraphen zu zählen, wenn man als Folge der *open world assumption* die Annahme machen muss, nie alle zu kennen.

Literatur

[Alk07] Alkhateeb, Faisal: *PSPARQL Query Language* . http://psparql.inrialpes.fr/, 2007.

[Dod05] Dodds, Leigh: *Introducing SPARQL: Querying the Semantic Web*. http://www.xml.com/pub/a/2005/11/16/introducing-sparql-querying-semantic-web-tutorial.html, 2005.

[Hera] Herman, Ivan: *SPARQL is a Candidate Recommendation*. http://www.w3.org/blog/SW/2007/06/15/sparql_is_a_candidate_recommendation.

[Herb] Herman, Ivan: *SPARQL is a Recommendation*. http://www.w3.org/blog/SW/2008/01/15/sparql_is_a_recommendation.

[Knu06] Knublauch, Holger: *Ontology Mapping with SPARQL CONSTRUCT*. http://composing-the-semantic-web.blogspot.com/2006/09/ontology-mapping-with-sparql-construct.html, 2006.

[McC05] McCarthy, Philip: *Search RDF data with SPARQL*. http://www.ibm.com/developerworks/xml/library/j-sparql/, 2005.

[MS08] Mintert, Stefan und Bastian Spanneberg: *SPARQL - Anfragesprache für RDF-Daten*. iX, 1:134 – 137, Januar 2008.

[Pru07] Prud'hommeaux, Eric: *RDF Data Access WG Charter*, 2007.

[PSa] Prud'hommeaux, Eric und Andy Seaborne: *SPARQL Query Language for RDF*. http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/.

[PSb] Prud'hommeaux, Eric und Andy Seaborne: *SPARQL Query Language for RDF*. http://www.w3.org/TR/2006/WD-rdf-sparql-query-20061004/.

[PSc] Prud'hommeaux, Eric und Andy Seaborne: *SPARQL Query Language for RDF*. http://www.w3.org/TR/rdf-sparql-query/.

[SM08] Seaborne, Andy und Geetha Manjunath: *SPARQL/Update – A language for updating RDF graphs*. http://jena.hpl.hp.com/~afs/SPARQL-Update.html, 2008.