Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BSC-2010-16

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Query Processing over Distributed Web Sources under Limited Access Capabilities using Semantic Constraints

Benjamin Dake

Arbeitsgruppe für

Datenbanken & Informationssysteme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

12. November 2010

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel. +49 (5 51) 39-17 42010
Fax +49 (5 51) 39-1 44 15
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 12. November 2010

Bachelor Thesis

Query Processing over Distributed Web Sources under Limited Access Capabilities using Semantic Constraints

Benjamin Dake

November 12, 2010

Supervised by Prof. Dr. Wolfgang May
Databases and Information Systems Group
Georg-August-Universität Göttingen

Abstract

A lot of the data sources that are available in the Web can only be accessed through limited access patterns. By entering some input values, appropriate output values are returned in a table-like schema. On the one hand, there are human-oriented sources that return data in form of dynamically generated Web pages by filling out Web forms (Deep Web). On the other hand, there are only machine accessible Web Services. This thesis implements the prototype of a Query Broker that uses these sources to answer declarative queries. For this, semantic annotations are assigned to the sources. This enables the query broker to select suitable sources for the query answering and to query them in an appropriate order by taking the output values of queried sources as the input for other sources.

Contents

1 Introduction **1**

2 Foundations **3**

2.1 Semantic Web . . . 3
2.2 RDF . . . 3
2.2.1 Graphical Representation . . . 4
2.2.2 Turtle Syntax . . . 4
2.3 RDFS . . . 6
2.4 SPARQL . . . 6

3 Web Data Sources **8**

3.1 Sample Sources . . . 8
3.2 Characteristics . . . 10
3.3 Technical Handling . . . 10
3.4 Source Annotation . . . 11
3.4.1 Signature Level . . . 11
3.4.2 Semantic Level . . . 12

4 Query Processing **16**

4.1 Overview . . . 16
4.2 Query Graph . . . 18
4.3 Web Data Source Matching . . . 19
4.4 Maximal Structural Overlappings . . . 21
4.4.1 Edge Inversion . . . 21
4.4.2 Splitting . . . 22
4.4.3 Separated Checks . . . 23

4.4.4 Graph Pruning . . . 24
4.5 Query Plan . . . 28
4.5.1 Query Order . . . 28
4.5.2 Query Plan . . . 29

5 Implementation **32**
5.1 General . . . 32
5.1.1 Jena Framework . . . 32
5.1.2 MARS Framework . . . 32
5.2 Class Architecture . . . 33
5.2.1 Broker . . . 33
5.2.2 WebDataSource . . . 33
5.2.3 View . . . 34
5.2.4 Query . . . 34
5.2.5 StructuralOverlapping . . . 34
5.2.6 QueryGraph . . . 34
5.2.7 Node . . . 35
5.2.8 Edge . . . 35

6 Conclusion **36**

Bibliography **37**

List of Figures

1.1 Schematic Representation of the Query Broker 2

2.1 RDF Graph of the Sample Triple 4

3.1 Sample Data of the Travel Domain 10
3.2 Mapping from Source A to the Travel Ontology 13

4.1 Comparison of the Query with Source A 17
4.2 Comparison of the Query with Source C 18
4.3 Initial Query Graph 19
4.4 Query Graph with Substituted Literals 20
4.5 Edge Inversion Example 21
4.6 Query Graph before and after the Splitting. 22
4.7 Query Graph after the Separated Checks 24
4.8 Sample Query and the Web Data Source 25
4.9 Structural Overlappings of the Source and the Query 27

5.1 Class Diagram for the Implemented Classes 33
5.2 Sample QueryGraph Instance 35

1 Introduction

This thesis implements the approach described in the paper "Semantic Annotations and Querying of Web Data Sources" [5].

A lot of the data sources that are available in the Web can only be accessed through restricted access patterns. For example, consider a flight portal that can be queried by entering a departure airport, a destination airport and a date to get connections between these airports by flight code, departure time, arrival time, price and the operating airline. We daily use such sources to gather information. In order to answer a question, often the use of different data sources is necessary, whereupon the results have to be combined manually. If we, for example, want to find the places where we can fly from Munich on 1.10.2010 for less than 50€, we can not use the above-mentioned source directly, since no destination airport is known. But there may be another data source that can be queried by entering a departure airport to get a list of reachable destination airports. It can be used to get the destination airports that are reachable from Munich. With the result, the first source can be queried.

Many data sources that are available in the Web require some input values and return corresponding output values in a table like-schema. They are called Web Data Sources. On the one hand, there are human-oriented sources that return data in form of dynamically generated Web pages by filling out Web forms (Deep Web). On the other hand, there are only machine accessible Web Services.

The goal of the thesis is to implement a Query Broker that automatically answers declarative queries by the use of distributed Web Data Sources. Since the semantics of the information provided by the different data sources is usually not explicitly given, an automatic combination of the information is a problem. For this purpose, the Web Data Sources are associated with the semantics of an application domain. If now a query in terms of the domain ontology is sent to the broker, appropriate Web Data Sources for the query an-

swering can be selected and queried, taking the input-output-characteristics into account. A schematic representation of the Query Broker is shown in Figure 1.1.

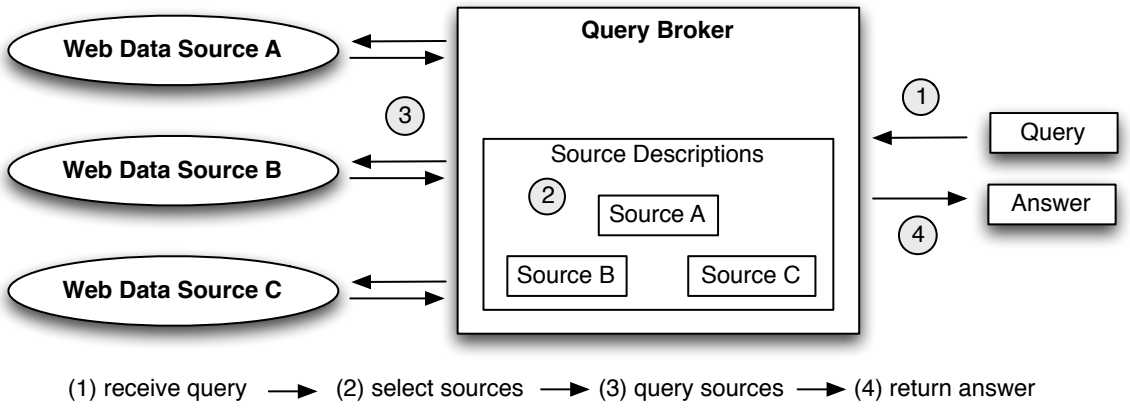


Figure 1.1: Schematic Representation of the Query Broker

This thesis deals not with the technical aspects of how to access the data sources. For this purpose, generic wrapper languages are available, which are used to encapsulate the sources. The thesis starts above this level and is concerned with the selection, query order and result combination of the wrapped sources.

Structure of the Thesis

The thesis is divided into 6 chapters. In Chapter 2 the basic theoretical foundations of the thesis are introduced. Chapter 3 covers the semantic annotation of Web Data Sources. In the subsequent Chapter 4, the query processing is described. Chapter 5 describes the implementation of the Query Broker. The thesis is concluded with a short summary and perspective.

2 Foundations

This thesis is based on concepts of the Semantic Web, that are introduced in this chapter. First, the idea of the Semantic Web is presented. In the next section, the Resource Description Framework (RDF), which is the data model underlying the Semantic Web, is introduced. The subsequent section introduces the RDF vocabulary definition language RDF Schema. This chapter ends with an overview of the RDF query language SPARQL.

2.1 Semantic Web

The World Wide Web (Web) is organized in interlinked documents, designed to provide content for human users. Thus, the Web is meant to be read and understood by humans, and not machines. The Semantic Web extends the World Wide Web with the purpose of assigning machine processable semantics to the data. This enables computers to automatically process information in a reliable way. For example, information from different sources can be automatically integrated and unknown connections can be discovered.

An important term concerning the Semantic Web is the ontology. A domain ontology formally specifies a domain. It defines concepts, properties and relationships of a domain. Besides, logical rules that allow reasoning are part of ontologies. [12]

2.2 RDF

The Resource Description Framework (RDF) [9] is a data model for representing information in the Semantic Web. RDF represents information by making statements about resources. A resource may be a Web page, a person, a book or any other object. A resource is identified by a Uniform Resource Identifier (URI). Thus, different sources can describe the same resource using its URI. To give an example, the city Berlin is a resource, identified

by its URI *http://example.org/cities/berlin*. A statement about a resource is called triple and consists of the following three parts:

- a subject, which is the resource (represented by its URI) the statement is made about,
- a predicate, which is the property (represented by its URI) of the resource that is described,
- and an object, which is the value of the property and can either be a URI or a literal.

Consider the sentence: "Berlin has a population of 3442675".

- Subject (URI): *http://example.org/cities/berlin*
- Predicate (URI) : *http://example.org/population*
- Object (literal): *3442675*

2.2.1 Graphical Representation

An RDF model consists of a set of triples. These triples form a labeled directed graph, where subjects and objects are nodes and predicates are directed edges. For the graphical representation, ovals are used for nodes. Literal nodes are drawn as rectangles. The example sentence is shown in Figure 2.1.

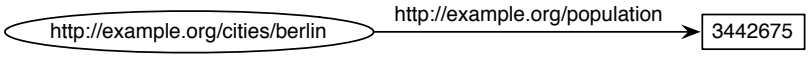


Figure 2.1: RDF Graph of the Sample Triple

2.2.2 Turtle Syntax

As a textual syntax for RDF data, the Terse RDF Triple Language (Turtle) [7] is used in this thesis. A triple has the format

```
subject predicate object.
```

URIs are enclosed in < and >. The above example statement is written as follows.

```
<http://example.org/cities/berlin> <http://example.org/population> 3442675.
```

Combining statements

Triples are separated by full stops. Two or more statements about the same subject can be combined using a semicolon as a separator.

```
subject predicate1 object1; predicate2 object2.
```

If additionally the predicates are the same, statements can be written with comma-separated objects.

```
subject predicate object1, object2.
```

Namespace Prefixes

Namespace prefixes are used to abbreviate URIs. The prefix definition

```
@prefix city: <http://example.org/cities/>.
```

declares *city* as a short prefix for <*http://example.org/cities/*> and allows to use *city:berlin* as a shorthand for <*http://example.org/cities/berlin*>.

The following prefix definition defines a default namespace.

```
@prefix : <http://example.org/>.
```

Blank Nodes

A blank node is a node that is neither a URI nor a literal. It indicates an anonymous resource. To serialize a blank node, an id is assigned to it. The scope of the id is limited to the serialization. Thus, the same blank node may have different ids in two documents. In Turtle, blank nodes are written as "*_:id*".

The sentence "Peter knows someone who lives in Berlin" might look like:

```
<http://example.org/persons/peter> :knows _:p.
_:p :livesIn <http://example.org/cities/berlin>.
```

In this example, *_:p* is a blank node. Alternatively, anonymous nodes can be defined by square brackets.

```
<http://example.org/persons/peter> :knows
        [ :livesIn <http://example.org/cities/berlin> ] .
```

2.3 RDFS

RDF Schema (RDFS) [10] is an extension of the Resource Description Framework, used to define vocabularies. A vocabulary declares classes and properties to describe data of a specific domain. The RDFS vocabulary descriptions themselves are written in RDF.

To specify a vocabulary, classes can be defined and arranged in a hierarchy. In addition, relations between classes can be specified by properties, including domain and range. The following may be a fragment of a simple ontology.

```
ex:City rdf:type rdfs:Class .
ex:Person rdf:type rdfs:Class .
ex:livesIn rdfs:domain ex:Person; rdfs:range ex:City.
```

The two classes *ex:City* and *ex:Person* are defined. A person can have the property *ex:livesIn* with a city as range. To express that a resource is an instance of a class, the predicate *rdf:type* (or its short form "*a*") is used. The example

```
<http://example.org/cities/berlin> rdf:type ex:City.
```

declares Berlin as a city.

2.4 SPARQL

The SPARQL Protocol and RDF Query Language (SPARQL) [3] is a query language for RDF data. A SPARQL query specifies a graph pattern, which is matched against an RDF graph and returns answers by binding variables.

Consider a query that asks for the cities where at least 100,000 people live.

```
PREFIX ex: <http://example.org/>
SELECT ?city
WHERE { ?city a ex:City .
        ?city ex:population ?population .
        FILTER ( ?population >= 100000 ) }
```

The SPARQL query basically consists of three parts: prefix, select and where.

Prefix The prefix defines namespaces similar to the Turtle syntax.

Where The graph pattern that is matched against the RDF data is specified by the where clause. The where clause consists of triple patterns, written in turtle syntax, where positions can be replaces by variables. A variable is indicated either by "*?*" or "*$*", followed by its name, and acts as a placeholder for URIs, literals or blank nodes. A variable appearing more than once acts as a join variable. For a solution, all triple patterns must match. Additionally, results can be restricted by filters. The filter condition "*?population >= 100000*" restricts the solutions to those where the value bound by *?population* is greater than or equal to 100000.

Select The select clause identifies the variables to be returned. The select clause can be replaced by ASK. In that case, a boolean value that indicates whether the query has a solution or not is returned.

In a basic graph pattern, as given above, all triple patterns must match. An optional graph pattern contains triple patterns that may match. For example

```
PREFIX ex: <http://example.org/>
SELECT ?person, ?email
WHERE { ?person a ex:Person.
        OPTIONAL { ?person ex:email ?email. } }
```

returns all persons, even if they don't have an email address. Hence, variables in optional parts may extend the solution.

Alternative graph patterns can be combined using UNION. Consider a query to find persons that were born **or** live in Berlin (or both).

```
PREFIX ex: <http://example.org/>
SELECT ?person
WHERE { ?person a ex:Person.
        { ?person ex:livesIn <http://example.org/cities/berlin>. }
        UNION
        { ?person ex:bornIn <http://example.org/cities/berlin>. } }
```

For a solution it is sufficient if one of the alternative graph patterns matches. Without the union, the result would be restricted to persons that were born **and** live in Berlin.

Further Reading. In [8] the topics of this chapter are introduced in detail.

3 Web Data Sources

A lot of the data sources that are available in the Web are internally based on databases that can't be accessed directly from the outside, but through limited access patterns. By entering some input values, appropriate output values are returned in a table-like schema. Such sources are called Web Data Sources and can be classified in two groups. On the one hand, there are human-oriented sources that return data in form of dynamically generated Web pages by filling out Web forms (Deep Web). On the other hand, there are sources that are only machine accessible by technologies like REST [1] and SOAP [11] (Web Services).

This chapter describes how Web Data Sources can be annotated such that the Query Broker can use them to answer queries.

3.1 Sample Sources

The thesis is illustrated by querying for flight connections. For this, the following three sample Web Data Sources are introduced.

(A) Source A is a flight portal that can be queried with a departure airport, a destination airport and a date to get connections between these airports by flight code, departure time, arrival time, price and the operating airline.

(B) By entering a departure airport, a destination airport and a date, possible connections with departure time, arrival time, price and flight code operated by Lufthansa are returned by http://lufthansa.de.

(C) Source C requires a departure airport to return a list of reachable destination airports in conjunction with the airlines that offer flights between these airports.

Travel Ontology

In addition to the sample sources, the travel ontology is introduced. The sample sources are later associated with it to specify the semantics of the sources. The following classes and properties are part of the travel ontology.

- Airports *travel:Airport* have the property IATA code *travel:code*.
- Airlines *travel:Airline* have a name *travel:name*.
- *travel:Flight* represents flight connections with the following properties. A flight connection has a flight code *travel:fcode*, departs from an airport *travel:from* at time *travel:deptTime* and arrives a destination airport *travel:to* at time *travel:arrTime*. A flight connection is related to an airline by *travel:operatedBy*.
- While *travel:Flight* represents an abstract flight connection, bookable connections *travel:BookableConn* are the actual instances of flight connections *travel:instanceOfConn*. A bookable connection is on a given date *travel:date* and has a price *travel:price*.

For example, consider a connection with the flight number "LH4728" from Frankfurt (FRA) to London (LHR). The connection is operated by Lufthansa and daily departs from Frankfurt (FRA) at 12:50 and arrives in London (LHR) at 13:25. A bookable connection is, for example, given on 01.10.2010 at the price of 439 €. The corresponding RDF data is:

```
@prefix t: <http://www.semwebtech.org/domains/2006/travel#> .
@prefix airport: <http://www.semwebtech.org/domains/2006/travel/airports/> .
@prefix airline: <http://www.semwebtech.org/domains/2006/travel/airlines/> .
@prefix flight: <http://www.semwebtech.org/domains/2006/travel/flights/>.

airline:lh a t:Airline; t:name "Lufthansa".
airport:fra a t:Airport; t:code "FRA".
airport:lhr a t:Airport; t:code "LHR".
flight:lh4728 a t:Flight; t:flightCode "LH4728";
   t:from airport:fra;  t:to airport:lhr;
   t:deptTime "12:50"; t:arrTime "13:25".
flight:lh4728-01-10-2010 a t:BookableConn; t:instanceOfConn flight:4728;
   t:date "01-10-2010"; t:price 439.
```

The graphical representation of the sample data is shown in Figure 3.1

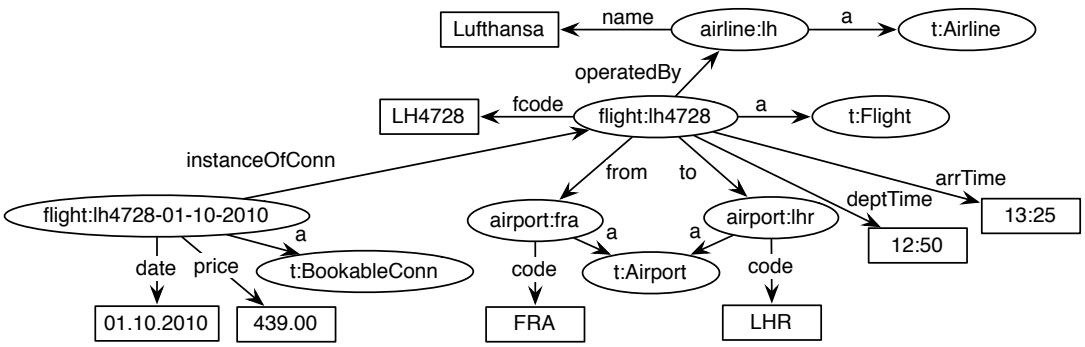


Figure 3.1: Sample Data of the Travel Domain

3.2 Characteristics

Web Data Sources range from Deep Web sources to Web Services. All these sources have in common that they return data in form of a table like schema. It can be seen as a predicate $q(\overline{x}) = q(x_1, ..., x_n)$ over variables $\{x_1, ..., x_n\}$, called the characteristic predicate of a Web Data Source. The characteristic predicate can only be queried through limited access patterns, which require input values to be given to return suitable output values. Thus, a Web Data Source provides predefined views on its data. A view v has a set of input arguments $\overline{qin} = \{xin_1, ..., xin_k\} \subseteq \{x1, ..., xn\}$ and returns corresponding output arguments $\overline{qout} = \{xout_1, ..., xout_m\} \subseteq \{x1, ..., xn\} \backslash \overline{qin}$. It is called the signature of a view, written as $\overline{qout} \leftarrow v(\overline{qin})$.

3.3 Technical Handling

Each type of Web Data Sources requires a specific technical handling. The actual technical handling of the different types of Web Data Sources is not part of this thesis. For this purpose, wrapper languages that provide a uniform set oriented interface to the outside are available. Web forms are queried using the Deep Web Query Language (DWQL). REST-based and SOAP-based Web services are wrapped with the Web Service Query Language (WSQL). Consider a wrapped view $\overline{qout} \leftarrow v(\overline{qin})$. If a set r of tuples of input variable bindings over $\overline{qin}$ is sent to the wrapper, the tuples are joint with the tuples q that the characteristic predicate is holding and the resulting tuples $\pi[\overline{qin} \cup \overline{qout}](q \bowtie r)$ are returned.

3.4 Source Annotation

To make Web Data Sources available for the Query Broker, they are described in the Web Data Source Description Language (WDSDL) [5]. By the use of the WDSDL vocabulary, a Web Data Source can be described in RDF. The notation of a source is done in two steps. The signature level specifies the characteristic predicate of the Web Data Source and the views provided upon this. Then, the signature is related to the domain ontology by the semantic level.

3.4.1 Signature Level

The signature level specifies the characteristic predicate and the views of a Web Data Source. For this, the variables of the characteristic predicate are labeled with tags. The sample sources could be tagged as:

Source A: (from, to, date, airline, flightCode, deptTime, arrTime, price).
(airline, flightCode, deptTime, arrTime, price)← viewA(from, to, date).

Source B connection(from, to, date, flightCode, deptTime, arrTime, price).
(flightCode, deptTime, arrTime, price) ← viewB(from, to, date).

Source C flight(from, to, airline).
(to, airline) ← viewC(from).

Note that the tag names can be freely chosen and may differ at different sources. The purpose of the tagging is not to assign a semantics to the sources. This is done later on the semantic level. The current level is only concerned with the signature description.

WDSDL specifies the following vocabulary for describing the signature:

```
@prefix <http://www.semwebtech.org/languages/2008/wdsdl#> .
:WebDataSource a owl:Class.
:baseURL rdfs:domain :WebDataSource; rdfs:range xsd:anyURI.
:DeepWebSource rdfs:subClassOf :WebDataSource.
:WebServiceSource rdfs:subClassOf :WebDataSource.
:View a owl:Class.
:providesView rdfs:domain :WebDataSource; rdfs:range :View.
:Tag a owl:Class.
:name rdfs:domain :Tag; rdfs:range rdfs:Literal.
```

```
:hasTag rdfs:domain :WebDataSource; rdfs:range :Tag.
:hasInputVariable rdfs:domain :View; rdfs:range :Tag.
:hasOutputVariable rdfs:domain :View; rdfs:range :Tag.
```

The description is done in RDF. Thus, the Web Data Source and its views are resources, represented by URIs. The tags are anonymous resources, described by name and data type. The tags of the characteristic predicate are related to the source by *:hasTag*. The views provided by the source are described by their input and output tags.

The description of the signature level of source A is:

```
@prefix travel:  <http://www.semwebtech.org/domains/2006/travel#> .
@prefix :        <http://www.semwebtech.org/languages/2008/wdsdl#> .

<bla://views/travel/sourceA> a :WebDataSource;
  :baseURL <http://www.sourceA.com/>;
  :providesView <bla://views/travel/sourceA/viewA>;
  :hasTag _:from, _:to, _:date, _:airline, _:fcode, _:depT, _:arrT, _:price.

_:from a :Tag; :name "from"; :datatype xsd:string.
_:to a :Tag; :name "to"; :datatype xsd:string.
_:date a :Tag; :name "date"; :datatype xsd:date; :format "dd.MM.yyyy".
_:airline a :Tag; :name "airline"; :datatype xsd:string.
_:fcode a :Tag; :name "flightCode"; :datatype xsd:string.
_:depT a :Tag; :name "deptTime"; :datatype xsd:time; :format "HH:mm".
_:arrT a :Tag; :name "arrTime"; :datatype xsd:time; :format "HH:mm".
_:price a :Tag; :name "price"; :datatype xsd:decimal.

<bla://views/travel/sourceA/viewA> a :View;
  :hasInputVariable _:from, _:to, _:date;
  :hasOutputVariable _:airline, _:fcode, _:depT, _:arrT, _:price.
```

3.4.2 Semantic Level

So far, the signature of the Web Data Source has been described. Now, the semantics of the signature is specified by correlating the signature with the domain ontology. For this, the tags of the characteristic predicate are mapped to the domain ontology. As exemplarily demonstrated in Figure 3.2 for Source A, each tag is associated with a node of the ontology. So, the relation between the tags is expressed in terms of the domain ontology.

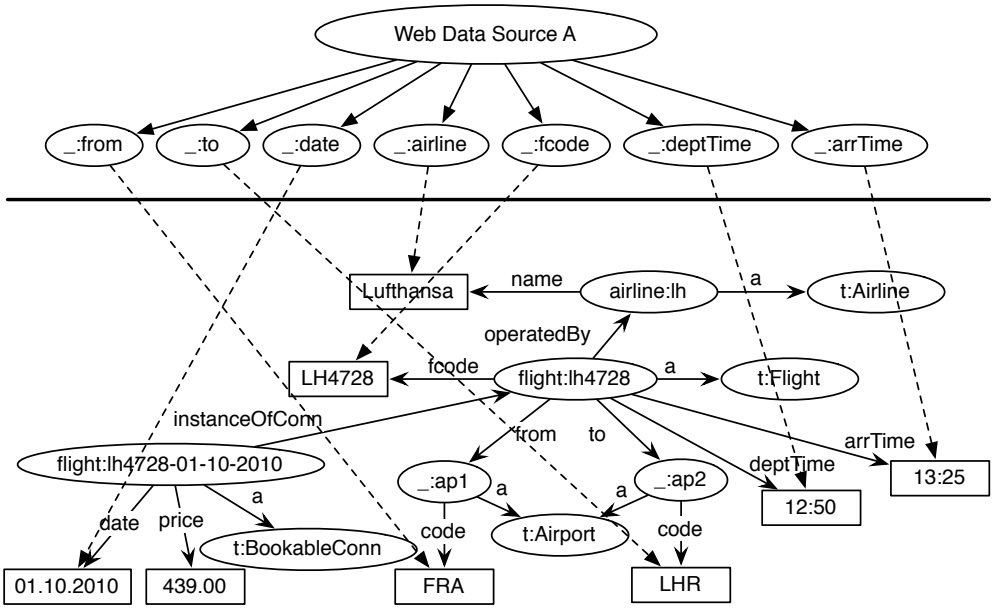


Figure 3.2: Mapping from Source A to the Travel Ontology

The approach of the WDSDL is to represent the correlation between the tags as RDF triples. Consider the following RDF triples. The tags are printed in bold.

```
_:flightV a travel:Flight;  travel:flightCode "fcode";
        travel:from [a travel:Airport; "from"];
        travel:to [a travel:Airport; "to"];
        travel:deptTime "deptTime"; travel:arrTime "arrTime".
        travel:operatedBy [a travel:Airline travel:name "airline"];
_:bconnV a travel:BookableConn; travel:instanceOfConn _:flightV;
        travel:date "date"; travel:price "price".
```

Instead of real data, the tag names are used. It can be seen as a pattern for the tuples returned by the source, where the tags act as placeholders for the actual values. But if such a source description would be available in the Web, it could be wrongly assumed as real data. Due to this, WDSDL uses source annotation statements to describe the graph pattern. The annotation statements are based on a reification vocabulary that allows to describe RDF statements using RDF. Consider the statement below.

```
_:flightV rdf:type travel:Flight.
```

An annotation statement for the statment looks like:

```
_:astmt a AnnotationStatement.
_:astmt rdf:subject _:flightV; rdf:predicate rdf:type; rdf:object  travel:Flight.
```

The semantic level of the WDSDL ontology works according to this principle. The graph pattern, which specifies the relation between the tags, is described using the source annotation statements. If required, the annotation statements can be materialized back into real RDF triples, resulting in the above-mentioned triples.

The second part of the WDSDL ontology is given below.

```
@prefix : <http://www.semwebtech.org/languages/2008/wdsdl#> .
:Annotation a owl:Class.
:hasAnnotation rdfs:domain [ owl:unionOf (:WebDataSource :View)] ;
        rdfs:range :Annotation.
:AnnotationStatement a owl:Class. :AnnotationConstraint a owl:Class.
:hasAnnotationStatement rdfs:domain :Annotation; rdfs:range :AnnotationStatement.
:hasAnnotationConstraint rdfs:domain :Annotation; rdfs:range :AnnotationConstraint.
```

The semantic level of Source A is annotated as follows. It is stored together with the signature level description and uses the blank nodes of the tags introduced in the signature level.

```
@prefix travel:  <http://www.semwebtech.org/domains/2006/travel#> .
@prefix :        <http://www.semwebtech.org/languages/2008/wdsdl#> .

<bla://views/travel/sourceA>  :hasAnnotation [
:localVar _:flightV, _:airlineV, _:airp1V, _:airp2V, _:bconnV ;
:hasAnnotationStatement
[ rdf:subject _:flightV; rdf:predicate rdf:type; rdf:object travel:Flight],
[ rdf:subject _:flightV; rdf:predicate travel:operatedBy; rdf:object _:airlineV ],
[ rdf:subject _:airlineV; rdf:predicate rdf:type; rdf:object travel:Airline],
[ rdf:subject _:airlineV; rdf:predicate travel:name; rdf:object _:airline ],
[ rdf:subject _:flightV; rdf:predicate travel:from; rdf:object _:airp1V],
[ rdf:subject _:flightV; rdf:predicate travel:to; rdf:object _:airp2V],
[ rdf:subject _:flightV; rdf:predicate travel:flightCode; rdf:object _:fcode ],
[ rdf:subject _:airp1V; rdf:predicate travel:code; rdf:object _:from ],
[ rdf:subject _:airp1V; rdf:predicate rdf:type; rdf:object travel:Airport ],
[ rdf:subject _:airp2V; rdf:predicate travel:code; rdf:object _:to ],
[ rdf:subject _:airp2V; rdf:predicate rdf:type; rdf:object travel:Airport ],
[ rdf:subject _:flightV; rdf:predicate travel:deptTime; rdf:object _:depT ],
```

```
[ rdf:subject _:flightV; rdf:predicate travel:arrTime; rdf:object _:arrT ],
[ rdf:subject _:bconnV; rdf:predicate rdf:type; rdf:object travel:BookableConn],
[ rdf:subject _:bconnV; rdf:predicate travel:instanceOfConn; rdf:object _:flightV ],
[ rdf:subject _:bconnV; rdf:predicate travel:date; rdf:object _:date ],
[ rdf:subject _:bconnV; rdf:predicate travel:price; rdf:object _:price ]].
```

WDSDL describes the characteristics of the the Web Data Sources in terms of RDF. Resources in RDF are identified by URIs. Anyway, the sources just return literal tuples. Since no URIs are returned by the sources, it is not possible to merge the results of different web data sources using URIs. So, the literal properties of the resources (like *travel:code* for airports) have to be used to join the results. A more correct way would be to specify which of the literal properties actually are key attributes. This may be added in a further version.

4 Query Processing

The Query Broker answers queries by means of distributed Web Data Sources. Available Web Data Sources are related to the domain ontology with WDSDL. To be able to identify usable sources, the queries must be posed in terms of the domain ontology. The Query Broker can handle simple SPARQL select queries with a basic graph pattern. This especially means, that no optional and alternative graph patterns are supported. Furthermore, the query is supposed to be completely specified in terms of the ontology using the rdf:type property, maybe done by some preprocessing.

To get the idea of the query handling, at first an overview of the query processing is given. Then the query processing is shown in detail.

4.1 Overview

When the Query Broker receives a query, its job is to answer the query by means of the known Web Data Sources. The query processing starts with the identification of useful Web Data Sources. For this, the graph pattern specified by the where clause of the query is compared to the WDSDL annotations of the sources, in order to find structural overlappings.

Consider the query "where can we go from MUC on 1.10.2010 for less than 50€?". The corresponding SPARQL query is:

```
PREFIX t: <http://www.semwebtech.org/domains/2006/travel#>
SELECT ?DEST ?P
WHERE { ?C a t:Flight;  t:from [a t:Airport; t:code "MUC"];
           t:to [a t:Airport; t:code ?DEST] .
        ?BC a t:BookableConn; t:instanceOfConn ?C; t:date "01.10.2010"; t:price ?P .
        FILTER (?P < 50 ) }
```

The graph pattern defined by the sample query is depicted in the upper part of Figure 4.1. For source A, consider the lower part of Figure 4.1. Source A completely covers the query, as well as Source B. Source C only covers a part of the query, as illustrated in Figure 4.2. Even if the sources A and B completely cover the query graph, they can't be used

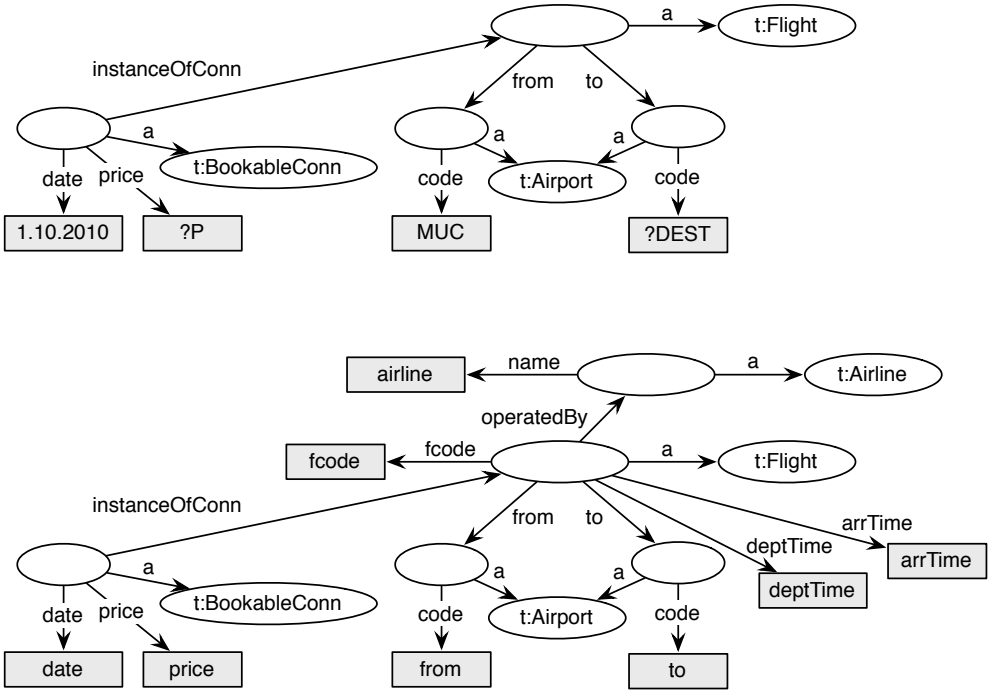


Figure 4.1: Comparison of the query with source A. Source A (lower part) covers the complete query (upper part).

directly, since the views provided by these sources need the destination airport as an input parameter. Now reconsider the view (to, airline) ← viewC(from) provided by Source C. It is applicable, since the only required input parameter, from, is given by the query. After querying Source C, a list of destination airports reachable from Munich is known. With that, sources A and B can be asked. Finally, the filter condition "$?P < 50$" is applied to the results and the final answer returned.

The query processing is structured in three main tasks. At first, the query graph is created according to the graph pattern specified by the query. In the next step, structural overlappings between the query graph and the Web Data Sources are ascertained. Once the useful Web Data Sources have been identified, a query plan is created under consider-

ation of the access limitations. It specifies the order in which the sources are queried and how the results are combined.

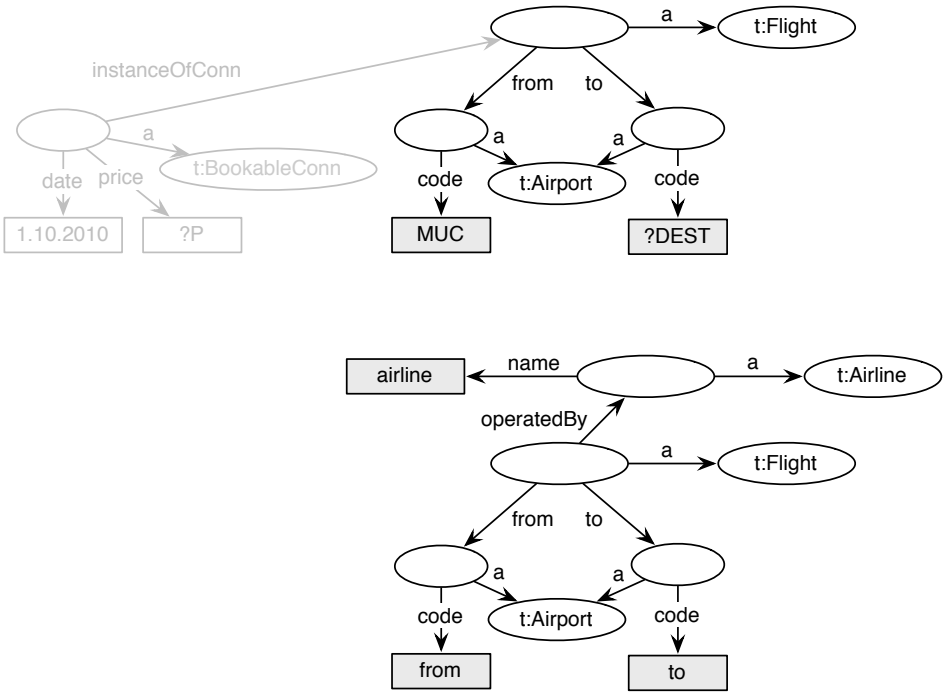


Figure 4.2: Comparison of the query with source C. Source C (lower part) covers just a fragment of the query (upper part). Uncovered parts are faded.

4.2 Query Graph

The where clause of a SPARQL query consists of a set of triple patterns. These triple patterns specify a graph pattern, similar to graphs specified by RDF data. To determine what a Web Data Source can contribute to the query, at first a graph representing the query, the query graph, is built.

The query graph is created according to the graph pattern specified by the query. A node is created for each subject and object occurring in the graph pattern. The directed edges are created according to the triple patterns and labeled with the predicates.

Reconsider the sample query

```
SELECT ?DEST ?P
WHERE { ?C a t:Flight; t:from [a t:Airport; t:code "MUC"];
           t:to [a t:Airport; t:code ?DEST].
        ?BC a t:BookableConn; t:instanceOfConn ?C; t:date "01.10.2010"; t:price ?P.
        FILTER (?P < 50 ) }
```

For the triple pattern "*?C a t:Flight*" a node representing the subject *?C*, and a node for the object *t:Flight* is created. Then, the nodes are linked with an arc from *?C* to *t:Flight*, labeled with the predicate *a*. This is done for each triple pattern and results in the graph shown in Figure 4.3.

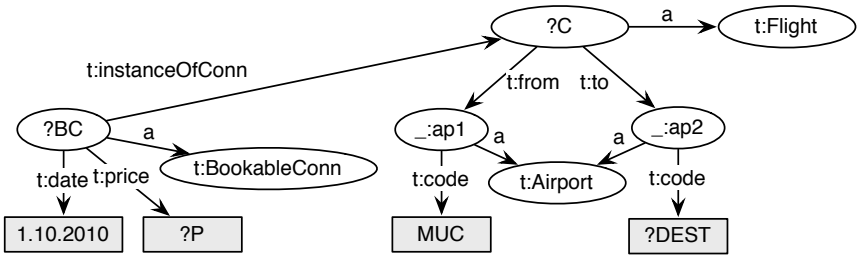


Figure 4.3: Initial Query Graph

4.3 Web Data Source Matching

The basic idea of how to compare the query graph with a Web Data Source is the following. The query graph is reduced until the Web Data Source covers the remaining parts of the graph. For this, a test if the Web Data Source covers the query graph is required. This is done by a SPARQL query. Since the query graph represents the graph pattern of a SPARQL query, it can simply be turned back into a SPARQL query. The query is then posed against the materialized WDSDL source annotation statements (see Section 3.4.2). Thereby, the variables are exploited to bind tags instead of real data.

The SPARQL query technically requires variables to bind tags. Thus, the literals occurring in the query graph have to be replaced by variables. Otherwise an airport with the code "*MUC*" would be searched in the source description, though a tag is to be found. For this reason, literals occurring in the query graph are substituted by variables. The sample

query graph with replaced URIs and literals is shown in Figure 4.4. The literals *"MUC"* and *"01.10.2010"* have been replaced by the variables *?FROM* and *?DATE*. Note that intuitive variable names are used in the running example. However, the variable names can be arbitrarily chosen.

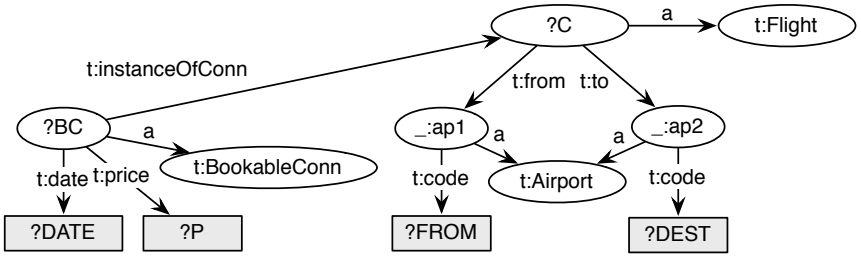


Figure 4.4: Query Graph with Substituted Literals

For the sample query graph the following query is created:

```
PREFIX t: <http://www.semwebtech.org/domains/2006/travel#>
ASK WHERE {
  ?C a t:Flight.
  ?C t:from _:ap1.
  ?C t:to _:ap2.
  _:ap1 a t:Airport.
  _:ap1 t:code ?FROM.
  _:ap2 a t:Airport.
  _:ap2 t:code ?DEST .
  ?BC a t:BookableConn.
  ?BC t:date ?DATE.
  ?BC t:price ?P}
```

Reconsider the materialized source description of Source A given in section 3.4.2. If the query is posed against it, the query returns true with the following variable bindings:

?C	?FROM	?DEST	?BC	?DATE	?P	_:ap1	_:ap2
_:flightV	**from**	**to**	_:bconnV	**date**	**price**	_:airp1V	_:airp2V

In the following, the expression "a source covers a query graph" signifies that the query graph posed against the materialized source description of the source returns true.

4.4 Maximal Structural Overlappings

So far, the Web Data Source matching has been introduced. Since a Web Data Source maybe just covers a part of the query graph, the partial structural overlappings of the query graph by the Web Data Sources have to be ascertained. This is discussed in this section.

4.4.1 Edge Inversion

The algorithm that will be presented in the subsequent Section 4.4.4 requires the query graph (without the ontology parts) to be a tree. Consider the query graph in the left part of Figure 4.5. The node *?P* has two incoming edges. Thus, the graph isn't a tree. If the

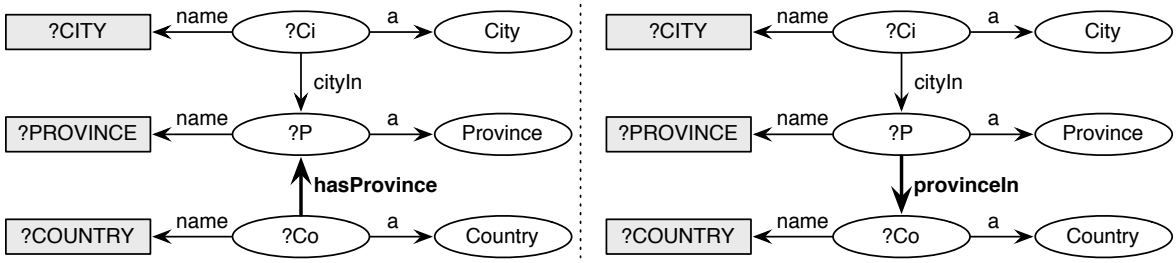


Figure 4.5: Edge Inversion Example

ontology would describe the relation *hasProvince* by its inverse *provinceIn*, as demonstrated in the right part of Figure 4.5, the graph would be a tree. Based on this idea, it is tried to convert the query graph into a tree by inverting edges. If the graph has a cycle after the inversion of an edge, the inversion is removed and the node has to be split. This is described in the next subsection.

The edge inversion is done to transform the query graph into a directed tree. As already mentioned above, this is required by the algorithm that will be presented in Section 4.4.4. In contrast, the matching queries use the edges in their original direction as defined in the ontology and used in the source annotations, otherwise the queries wouldn't match. Thus, there are different views on the query graph. On the one hand, the graph is considered as the query it represents with the edges in their original direction (query view). On the other hand, the graph is considered with the inverse edges (graph view).

4.4.2 Splitting

For instance, consider a Web Data Source that provides information about persons, including the native city of a person and the city the person currently lives in. The materialized source description looks like the following (tags are printed bold).

```
_:p a :Person; :name "person";
  :bornIn [a City; :name "bornIn"];
  :livesIn [a City; :name "livesIn"].}
```

A query looking for persons that live in their native city is diagrammed in the left part of Figure 4.6. The source obviously provides all the required information to answer the query. To determine if the the query graph is covered by the source, the following query is created.

```
ASK WHERE {
  ?PERSON a :Person; :name ?NAME; bornIn ?C; livesIn ?C.
  ?C a :City; name ?CITY.}
```

The SPARQL query posed against the materialized WDSDL description of the source returns false. This is because the city acts as a join condition. The variables *?C* and *?CITY* can either bind the tag *bornIn* or the the tag *livesIn*, but not both at once. The tags may return the same cities, but the source description just contains the tag names. Thus, the triple patterns "*?PERSON :bornIn ?C*" and "*?PERSON :livesIn ?C*" can't be bound at the same time. To avoid this problem, the node *city* is split in two nodes. The query graph with split nodes is shown in the right part of Figure 4.6.

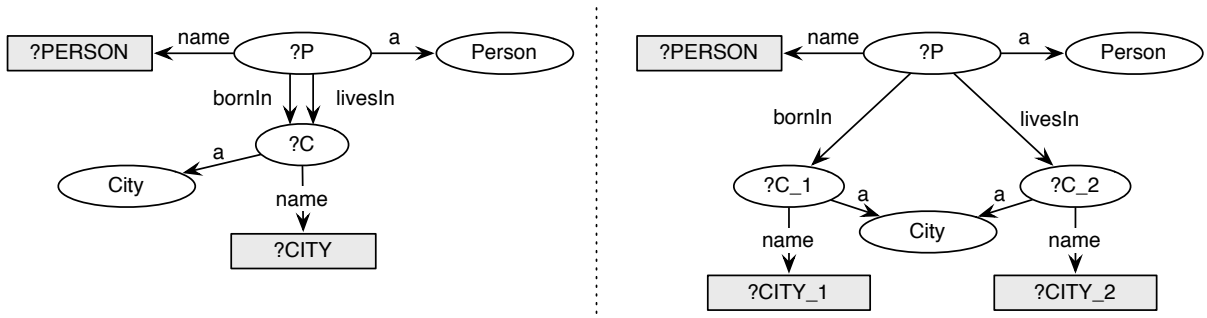


Figure 4.6: Query Graph before and after the Splitting.

The query has the following form after the splitting.

```
ASK WHERE {
  ?PERSON a :Person; :name ?NAME;
    bornIn ?C_1; livesIn ?C_2.
   ?C_1 a :City; name ?CITYNAME_1.
   ?C_2 a :City; name ?CITYNAME_2.}
```

Now it matches with the following solution.

?Person	?NAME	?C_1	?CITYNAME_1	?C_2	? CITYNAME_2
_:p	**person**	_:c1	**bornIn**	_:c2	**livesIn**

The problem demonstrated above is a general problem when a variable node has multiple incoming edges. The different tags may return the same values, but the source description just contains the tag names instead of real data. So, variable nodes with more than two incoming edges are split into one node for each incoming edge. The fragment that is connected to the split node by outgoing edges is copied to each new node. At this point, it is important that the graph is cycle free, otherwise this wouldn't work. Due to this reason, cycles in the graph pattern are forbidden. The equality of the values that are returned for the different split parts is checked later with the help of the results. A side effect of the splitting is that the query qraph (in the graph view), if it is considered without the ontology nodes like "*City*", now is a tree, because each node can only have one incoming edge and cyles are forbidden.

4.4.3 Separated Checks

Now, each triple pattern represented by the query graph in its query view is checked separately whether or not it is covered by the Web Data Source. The idea is the following. If an isolated triple pattern isn't covered by the Web Data Source, it especially isn't covered in combination with other triple patterns. Each check is done by a SPARQL query like

```
ASK WHERE { S P O. S a TypeOfS. O a TypeOfO. }
```

where S is the subject, P the predicate and O the object of the triple pattern. If a type of the subject or object is available, it is added to the check. The query is posed against the materialized source annotation statements. If it returns false, then the triple pattern is removed from the query graph.

The following separated checks are done for the sample query graph (Figure 4.4)

```
1) ASK WHERE { ?BC t:date ?DATE. ?BC a t:BookableConn.}
2) ASK WHERE { ?BC t:price ?P. ?BC a t:BookableConn.}
3) ASK WHERE { ?BC t:instance t:instanceOfConn ?C. ?BC a t:BookableConn. ?C a t:Flight.}
4) ASK WHERE { ?C t:from _:ap1. ?C a t:Flight . _:ap1 a t:Airport.}
5) ASK WHERE { _:ap1 t:code ?FROM  _:ap1 a t:Airport .}
6) ASK WHERE { ?C t:to _:ap2. ?C a t:Flight . _:ap2 a t:Airport.}
7) ASK WHERE { _:ap2 t:code ?DEST. _:ap2 a t:Airport.}
```

For source C, the queries 1, 2 and 3 return false. Hence, the corresponding triples are removed from the query graph. The remaining query graph for source C is shown in Figure 4.7.

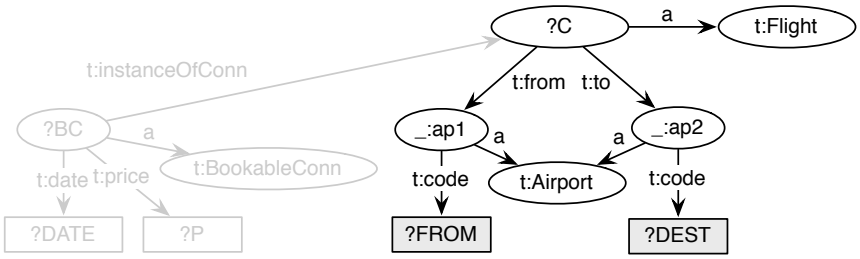


Figure 4.7: Query Graph after the Separated Checks

4.4.4 Graph Pruning

The Web Data Source provides information for the triple patterns that survived the separated checks, but this does not necessarily mean that the Web Data Source also covers these triples in the structural combination required by the query. So the remaining query graph as a whole is checked against the source. This test returns true for Source C and the remaining query graph (Figure 4.7).

But consider another Web Data Source that provides information about the parents of a person as illustrated in the right part of Figure 4.8. The graph of a query asking for the mother and the maternal grandfather of a child is diagrammed in the left part of Figure 4.8. The separated checks return true for each triple, but the source doesn't cover the triples in combination, since the structure of the Web Data Source is different from the query graph. To determine the maximal structural overlappings with the Web Data Source, the query

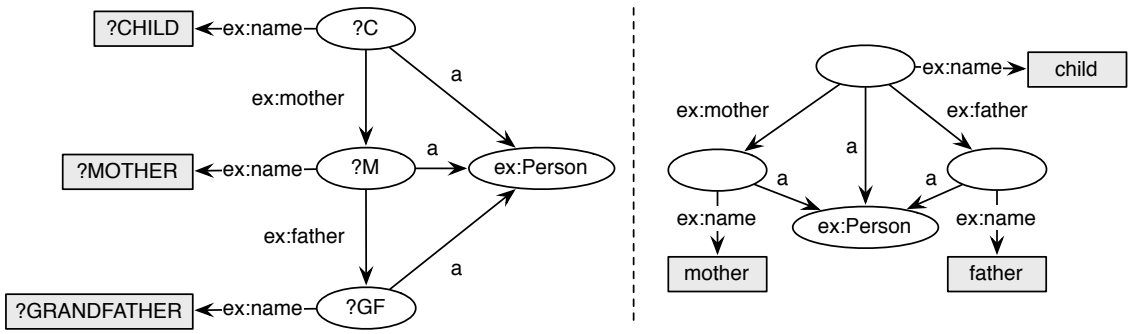


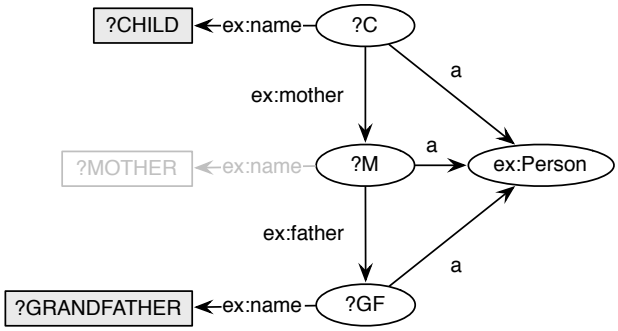
Figure 4.8: Sample Query (left part) and the Web Data Source (right part)

graph is reduced until the Web Data Source covers the remaining graph. This is done by the following algorithm: While the Web Data Source doesn't cover the query graph, a leaf triple pattern is selected and removed. A leaf triple pattern is a node (object) that has no outgoing edge, together with its incoming edge (predicate) and the corresponding node (subject). It can be removed without splitting the graph in unconnected components.

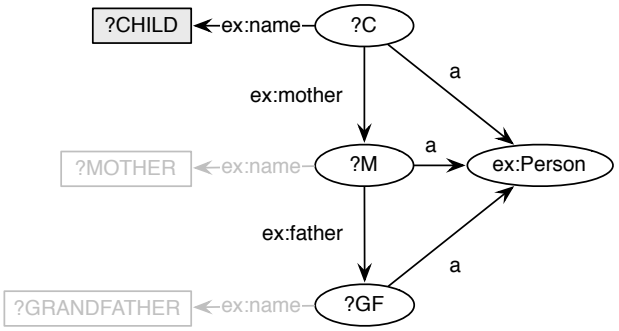
Because the ontology parts (triples of the form "node rdf:type type") are important for the meaning of the variables, they are not removed by the algorithm. This algorithm uses the graph view. Thus, the removable parts of the graph form a tree, because each node can only have one incoming edge. Hence, a leaf triple is always available.

The algorithm applied to the example may remove the triples in the following order.

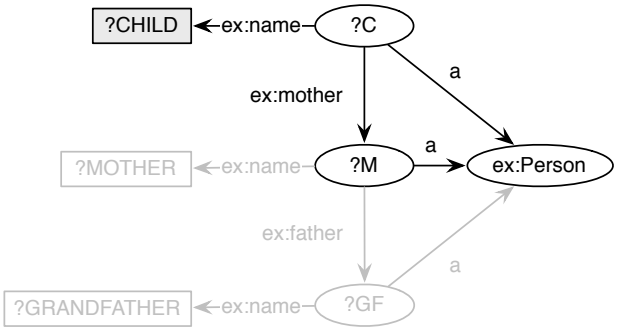
At first, "*?M ex:name ?MOTHER*" is removed.



Then, "*?GF ex:name ?GRANDFATHER*" is removed.



Next, the triple "*?M ex:father ?GF*" is removed. Now the source covers the remaining graph.



The triple pattern "*?M ex:father ?GF*" isn't covered in combination with the remaining query graph, because the query graph is covered after the removal of "*?M ex:father ?GF*". The triple pattern "*?GF ex:name ?GRANDFATHER*" was connected to the query graph by "*?M ex:father ?GF*", thus it also isn't covered in combination with the triples of the remaining query graph. But for the other removed triple pattern, "*?M ex:name ?MOTHER*" , this is unknown. As the maximal structural overlapping is searched, it is added to the query graph. The query graph is still covered. So, the query graph and the source have a maximal structural overlapping as given in the upper part of Figure 4.9.

Nevertheless, the removed triple patterns survived the separated test. Thus, the Web Data Source is also appropriate for the removed triple patterns, but has to be queried separately for those. Consider Figure 4.9. The Web Data Source has two structural overlappings with the query graph. The upper part contains the remaining query graph and the cover by the source. The lower part contains the two removed triple patterns. The Web Data Source covers both parts, but not in combination. Hence, the Web Data Source has to be queried separately for both parts. Please note that the triple pattern "*?M ex:name ?MOTHER*" has also been added to the lower overlapping, because it is the identifying attribute of *?M*.

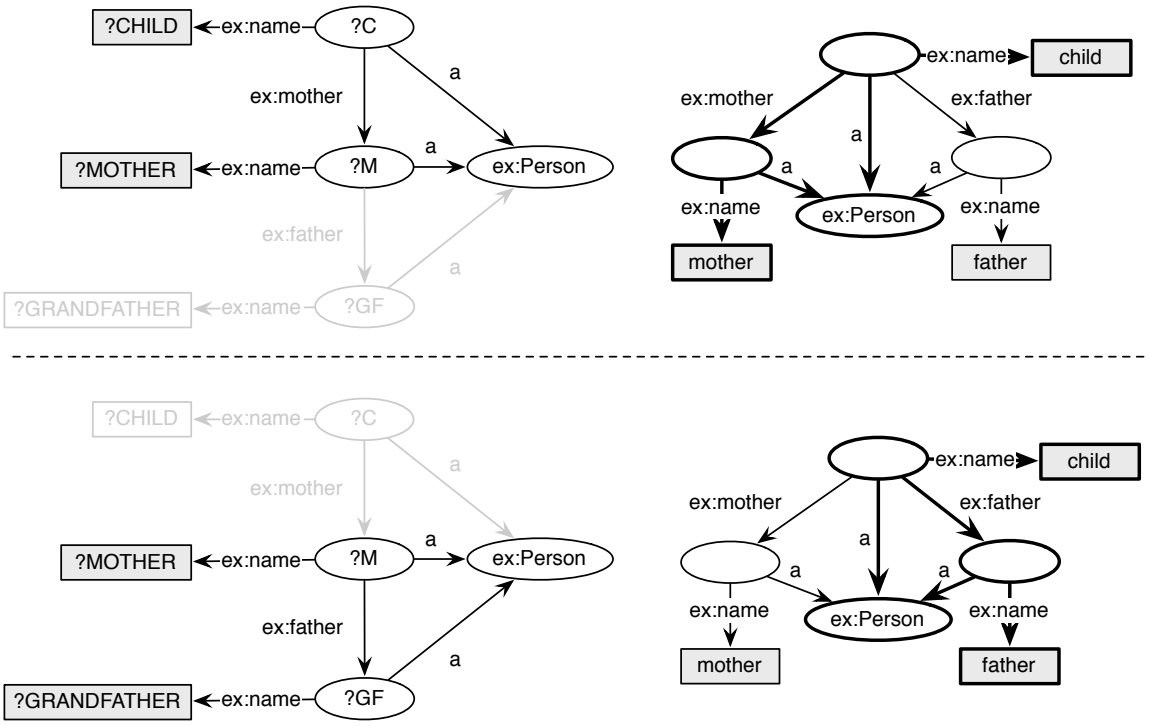


Figure 4.9: Structural Overlappings of the Source and the Query

Summarized, the algorithm works as follows: Leaf triples are removed until the Web Data Source covers the query graph. Then the removed triple patterns are added in reverse order, since the maximal structural overlapping is searched. If the graph isn't covered anymore after a triple pattern has been added, the triple pattern and its children (triple patterns connected by outgoing edges of the object) are finally removed from the query graph. The result is the remaining query graph, which is a maximal structural overlapping between the query graph and the web data source. Additionally, the removed triple patterns are also structural overlappings with the source.

4.5 Query Plan

Once a list of appropriate Web Data Sources, or rather structural overlappings, has been created, they have to be accessed in a suitable order. Due to the input-output-charactersitics, the sources can not be queried in any order.

4.5.1 Query Order

A Web Data Source can only be queried by its views. In consideration of the input-output-characteristics, a view can only be accessed if values for all input tags are known. The querying starts with the structural overlappings, where all input tags are known. The results of these overlappings expand the set of known variables. Then, the remaining overlappings are checked if they can be queried. If an overlapping can be queried and contribute not yet covered parts of the query graph, it is added to the query plan. A query order is determined by iterating this until the complete query graph is covered by the queried overlappings. Note that even if sources for all triples of the query are available, it is possible that the query can not be answered by reason of the input-output-characteristics of the sources.

Reconsider the flight example.

```
PREFIX t: <http://www.semwebtech.org/domains/2006/travel#>
SELECT ?DEST ?P
WHERE { ?C a t:Flight;  t:from [a t:Airport; t:code "MUC"];
           t:to [a t:Airport; t:code ?DEST] .
        ?BC a t:BookableConn; t:instanceOfConn ?C; t:date "01.10.2010"; t:price ?P .
        FILTER (?P < 50 ) }
```

The corresponding query graph has been given in Figure 4.4.

The sample sources A, B and C have structural overlappings with the query graph. The following views are available to access the sources.

Source A: (airline, flightCode, deptTime, arrTime, price) ← viewA(from, to, date).

Source B: (flightCode, deptTime, arrTime, price) ← viewB(from, to, date).

Source C: (to, airline) ← viewC(from).

The mapping from the query variables to the tags is ascertained by a select query, at which the variables are used to bind tags, like mentioned in Section 4.3.

The literals "MUC" and "01.10.2010" have been replaced with the variables *?FROM* and *?DATE*. So these variables are known. Only viewC can be queried with these variables. Thus, it is added to the query plan. Source C binds the variable *?DEST*. Now the sources A and B are applicable. The source A is added to the query plan. Now the complete query is covered. Once the query plan has been executed, the filter conditions are applied to the result and the variables of the select clause are returned.

4.5.2 Query Plan

The MARS Framework is used for the actual query execution. The query order is transformed into a MARS CCS sequence of queries, at which not required output values are removed in intermediate steps, and that ends with a test of the filter conditions.

An example of such a query plan in XML [13] Markup is given next.

```
<execute>
  <ccs:Sequence xmlns:ccs="http://www.semwebtech.org/languages/2006/ccs#"
                xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#"
                xmlns:dwql="http://www.semwebtech.org/languages/2008/dwql#">
    <ccs:Query eca:bind-to-variable="DATE">
      <eca:Opaque eca:language="http://www.w3.org/XPath">"01-10-2010"</eca:Opaque>
    </ccs:Query>
    <ccs:Query eca:bind-to-variable="FROM">
      <eca:Opaque eca:language="http://www.w3.org/XPath">"MUC"</eca:Opaque>
    </ccs:Query>
    <ccs:Query>
      <dwql:Query>
        <dwql:view dwql:resource="bla://views/travel/sourceC/viewC"/>
        <dwql:inputVariable dwql:name="FROM" dwql:use="from" />
        <dwql:outputVariable dwql:name="DEST" dwql:use="to" />
        <dwql:outputVariable dwql:name="tmp0" dwql:use="airline" />
      </dwql:Query>
    </ccs:Query>
    <ccs:Projection>
      <ccs:remove-variable name="tmp0" />
    </ccs:Projection>
    <ccs:Query>
      <dwql:Query>
        <dwql:view dwql:resource="bla://views/travel/sourceA/viewA"/>
        <dwql:inputVariable dwql:name="FROM" dwql:use="from" />
```

```
26          <dwql:inputVariable dwql:name="DEST" dwql:use="to" />
27          <dwql:inputVariable dwql:name="DATE" dwql:use="date" />
28          <dwql:outputVariable dwql:name="P" dwql:use="price" />
29          <dwql:outputVariable dwql:name="tmp0" dwql:use="arrTime" />
30          <dwql:outputVariable dwql:name="tmp1" dwql:use="deptTime" />
31          <dwql:outputVariable dwql:name="tmp2" dwql:use="flightCode" />
32          <dwql:outputVariable dwql:name="tmp3" dwql:use="airline" />
33        </dwql:Query>
34      </ccs:Query>
35      <ccs:Projection>
36        <ccs:remove-variable name="tmp0" />
37        <ccs:remove-variable name="tmp1" />
38        <ccs:remove-variable name="tmp2" />
39        <ccs:remove-variable name="tmp3" />
40      </ccs:Projection>
41      <reldb:Store>
42        <ccs:has-input-variable name="*" />
43        <reldb:tablename useValue="TEMP" />
44      </reldb:Store>
45      <ccs:Query>
46        <reldb:QueryFakeView reldb:from="TEMP">
47          <ccs:has-output-variable name="DEST" />
48          <ccs:has-output-variable name="P" />
49          <reldb:where>( "P" < 50 )</reldb:where>
50        </reldb:QueryFakeView>
51      </ccs:Query>
52      <reldb:Store>
53        <ccs:has-input-variable name="DEST" />
54        <ccs:has-input-variable name="P" />
55        <reldb:tablename useValue="RESULT" />
56      </reldb:Store>
57    </ccs:Sequence>
58  </execute>
```

The data flow in the MARS Framework is based on sets of tuples of variable bindings. An example of a tuple is *(FROM/"MUC", DATE/"01-10-2010")*, at which the variable *FROM* is bound to *"MUC"* and the variable *DATE* is bound to *"01-10-2010"*. A set of such tuples can be seen as a relation with the attributes *FROM* and *DATE*.

Line 5-10 The known variables are initialized. The values given by the query are bound.

Line 11-18 viewC is queried. The input and output variables are specified. Row 15, for example, specifies the variable *FROM* as the input for the "*from*" tag. The bindings of the specified input variable (*FROM*) are sent to the query engine. Tuples containing bindings for the new variables *DEST* and *tmp0* are returned. The current tuples are joined with the returned tuples using *FROM* as a join variable: *(DATE, FROM)* $\bowtie$ *(FROM, DEST, tmp0)*. So, the tuples now have the form *(DATE, FROM, DEST, tmp0)*.

Line 19-21 viewC returned values for the output tag airline (*tmp0*). But since it is not needed for the query answering, it is removed.

Line 22-34 viewA is queried. It adds the price (variable *P*).

Line 41-55 The tuples that satisfy the filter condition $p < 50$ are selected. The final result is stored in the table "*RESULT*".

Like demonstrated above, each queried view contributes its structural overlapping to the result. The combined result is the intersection of all queried Web Data sources.

5 Implementation

This chapter gives an overview of the Query Broker implementation. The used tools are introduced and the implemented classes are presented. A detailed description of the classes and their methods is not given, but can be found in the source code. Instead, the significant design decisions of the implementation are pointed out.

5.1 General

The goal of the implementation is to test the described query processing approach in practice. The implementation is a prototype of a Query Broker that implements the basic functionality. The implementation has been done in the programming language Java [6], using the Jena Framework and the MARS Framework.

5.1.1 Jena Framework

Jena [2] is an open source framework for building Semantic Web applications. It is written in Java and provides the functionality to handle RDF data. The Jena API can be used to to read, write and manipulate RDF data. SPARQL queries are supported by ARQ, a query engine for Jena.

5.1.2 MARS Framework

The MARS (Modular Active Rules for the Semantic Web) [4] Framework is developed by the DBIS group of Göttingen University. It is a framework for specifying and implementing active rules (ECA rules) in the Semantic Web. Among others, the MARS framework supports queries against Deep Web sources and Web Services by the Deep Web Query Language (DWQL) and the Web Service Query Language (WSQL).

5.2 Class Architecture

Now, the interaction of the different classes is presented. Then, a short description is given for each class. (Class names are italicized)

The *Broker* is the main class of the prototype implementation. During the start-up, it creates a *WebDataSource* object for each known source. After this, the *Broker* is ready to process queries. When the *Broker* receives a query, it creates a new *Query* object. The *Query* object parses the query using the Jena Framework and builds a *QueryGraph* for the query. Once the *Query* object has been initialized, the *Broker* passes a list of the available *WebDataSources* to the *Query*. Then, the *Query* starts with the identification of useful sources, arranges them in a suitable order, and creates a query plan.

The prototype implementation of the Query Broker consists of the classes shown in Figure 5.1.

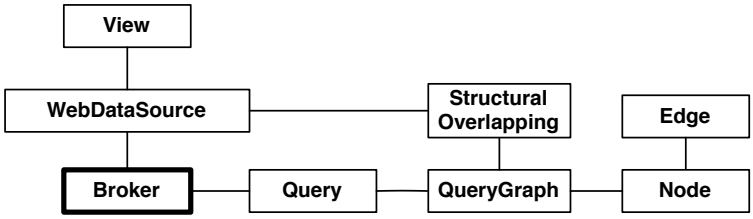


Figure 5.1: Class Diagram for the Implemented Classes

5.2.1 Broker

The static class *Broker* is the main class. It maintains the available web data sources and receives the queries.

5.2.2 WebDataSource

The *WebDataSource* class is the internal data structure of a Web Data Source. A *WebDataSource* object is created for each available Web Data Source. The constructor takes the URL of the WDSDL source description file and creates a new object according to the source description. A *WebDataSource* object stores an RDF model with the WDSDL source description, an RDF model with the materialized source annotation statements, a list of its

tags and a list of its views. The *WebDataSource* class provides methods to access those information.

5.2.3 View

The *View* class is the internal data structure of a view. Among others, it stores the input and output tags of the view. The attributes are accessible through getter and setter methods.

5.2.4 Query

The *Query* class is responsible for the query processing. It parses the query, builds the query graph, compares the query graph with the sources and creates the query plan.

5.2.5 StructuralOverlapping

StructuralOverlapping is a simple data class. An instance is created for each determined structural overlapping. It stores the overlapped triples of the query graph and the mapping from the tags to the query variables.

5.2.6 QueryGraph

The query graph requires a data structure that can store labeled nodes and labeled directed edges. The following data structure has been chosen. A query graph is represented by a *QueryGraph* object. It keeps a list of the nodes. A *Node* has a label and a list of its outgoing and incoming *Edge*s. An *Edge* has a label and stores its source *Node* (from) and its target *Node* (to).

Consider a short example.

```
?city ex:population ?population.
```

The example is stored as shown in Figure 5.2

The *QueryGraph* stores the *Node*s ":node1" and ":node2". The *Edge* ":edge1" stores its source *Node* of the edge (":node1"), and its target *Node* (:node2).

Please note that it would be sufficient to just store the outgoing edges of a node and the target nodes of the edges, but due to performance reasons the nodes also keep a list of their incoming edges, and the edges store their source nodes .

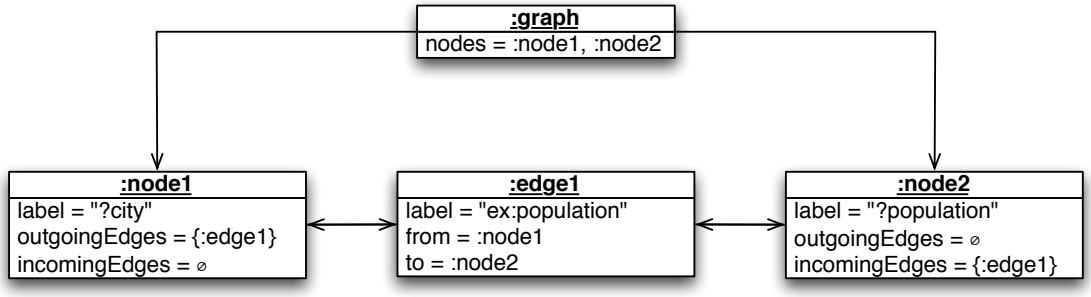


Figure 5.2: Sample QueryGraph Instance

5.2.7 Node

As already mentioned, a *Node* has a label and a list of its incoming and outgoing *Edge*s. The rdf type declarations (node rdf:type type) are not stored as edges in the graph. Instead, the *Node* class has an attribute to store its type. This is done for two reasons. First, it impoves the performance. Second, it simplifies the algorithms that work on the graph, because without these edges the graph is a tree.

5.2.8 Edge

The *Edge* class has attributes for storing the source and target *Node*, the label and an enabled flag. The enabled flag indicates if an *Edge* is enabled or disabled. If an *Edge* is disabled, the algorithms consider the graph without this *Edge*.

6 Conclusion

The prototype version of a Query Broker that uses distributed Web Data Sources to answer SPARQL queries has been developed in this thesis. A way to assign semantic annotations to the Web Data Sources by dint of the Web Data Source Desription Language has been described. For the identification of appropriate sources that can be used to answer a query, a graph-based matching algorithm that compares the query with the source annotations has been designed. The identified sources are arranged in an applicable query order, which is executed using the MARS Framework.

Further Work

The current version of the Query Broker can be extended in a variety of ways.

Supported Queries. The current version of the Query Broker only supports simple SPARQL queries with a basic graph pattern. A support for complex graph patterns, such as alternative and optional graph patterns, could be added.

Source Selection. The algorithm that creates the query order only pays attention to the sources that have structural overlappings with the query. A further version could also take the other sources into account. Even if they can't directly contribute to the query answer, their outputs may help to make other views accessible.

If different Web Data Sources have the same structural overlappings with the query, the query broker only queries one of those. But since they may return different results, all of these sources could be queried.

Result Combination. The literal properties of the resources are used to join the results. A more correct way would be to specify which of the literal properties actually are key attributes.

Bibliography

[1] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000.

[2] Jena Semantic Web Framework for Java. http://openjena.org/.

[3] SPARQL Query Language for RDF. http://www.w3.org/tr/rdf-sparql-query/. 2008.

[4] MARS Framework. http://www.dbis.informatik.uni-goettingen.de/mars/.

[5] T. Hornung and W. May. Semantic annotations and querying of web data sources. In *On the Move to Meaningful Internet Systems: OTM 2009*, volume 5870 of *Lecture Notes in Computer Science*, pages 112–129. Springer, 2009.

[6] Java. http://www.oracle.com/technetwork/java/index.html.

[7] Turtle Terse RDF Triple Language. http://www.w3.org/teamsubmission/turtle/. 2008.

[8] S. Rudolph P. Hitzler, M. Krötzsch and Y.Sure. *Semantic Web: Grundlagen*. Springer, 2008.

[9] Resource Description Framework (RDF). http://www.w3.org/rdf/. 2004.

[10] RDF Schema (RDFS). http://www.w3.org/tr/rdf-schema/. 2004.

[11] SOAP. http://www.w3.org/tr/soap12-part1/. 2007.

[12] J. Hendler T. Berners-Lee and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[13] Extensible Markup Language (XML). http://www.w3.org/xml/. 2008.