INSTITUT FÜR INFORMATIK
Datenbanken und Informationssysteme

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT
DÜSSELDORF

Automatische Übersetzung von RDQL-Anfragen in SQL
(Automatic translation of RDQL queries into SQL)

Matthäus Leschek Zloch

Bachelorarbeit

Beginn der Arbeit:	28. Juli 2005
Abgabe der Arbeit:	28. Oktober 2005
Gutachter:	Prof. Dr. Stefan Conrad
	Prof. Dr. Michael Leuschel

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 28. Oktober 2005

Matthäus Leschek Zloch

Contents

1 Introduction

1.1 Motivation

Semantic Web is the vision giving semantically rich information on the web exact meaning in a way that machines can understand. This enables machines and finally Semantic Web applications to process and understand the semantics of the content of a document. This was the idea of Tim Berners-Lee in 1999, the man who also invented the World Wide Web [BLHL01].

RDF and OWL as describing languages constitute the basic technology behind this vision. They both simplify the ability to describe and represent information about resources and ontologies on the web. So the machines are able to understand the information they show to the users. Using Relational.OWL, it is meanwhile not a mystery being able to convert schema and data stored in a relational database into a Semantic Web capable document. Data stored as an RDF/OWL representation can be queried using semantic query languages like RDQL, which is discussed here.

Nevertheless, the place where data is stored are still relational databases. This bachelor thesis is about a wrapper system, called RDQuery. RDQuery enables the user to access and query data actually stored in relational databases by literally translating RDQL queries automatically into their relational query correspondence, which is SQL.

The next sections shortly introduce step by step the technologies this bachelor thesis deals with. Starting with the idea of the Semantic Web, going over RDF, a Semantic Web metadata model and the ontology language OWL, we descend to Relational.OWL and RDQL, the main foundation this thesis is based on. The introduction is followed by a detailed description of the problem. The latter end of this thesis is about possible solution approaches, including the implementation and a description of the most important classes. The thesis ends with an example of a possible implementation in the form of a graphical user interface.

1.2 The Semantic Web

”The semantic of something is the meaning of something” [W3].

The major idea of the Semantic Web is to extend the World Wide Web we currently know in appending exact meaning to web information, i.e. semantics, in a machine-readable form. Actually, the web we know is only about displaying information to humans, but it has no meaning for machines. The Semantic Web is about describing semantic rich web content in a way that different computers and independent applications can process and understand. Special syntax rules are used to enable independent applications to exchange information with each other. Independent Semantic Web applications may then collect, combine, and present web information from many different sources in a meaningful way [W3C].

The email of Matthäus Zloch is matthaeus.zloch@uni-duesseldorf.de.

The sentence above shows a statement with semantic rich content. A human being which is able to speak a language is able to deduce:

1. There is something called Matthäus Zloch.
2. This something has an email.
3. The value of this is matthaeus.zloch@uni-duesseldorf.de.

In the Semantic Web, information is stored in form of simple statements like the one above.

Similar to the current web, the Semantic Web will also have services like search engines. The difference lies in obtaining their results. Current search engines can actually answer only one question, e.g.:

Which pages contain the term 'database management system'?

The result is a collection of web pages that matched the query. The Semantic Web is about the relationship between things. Resources are described with the help of other resources and information can be deduced. Search engines are then able to answer a question more precisely. For example:

List all albums from The Beatles.

A Semantic Web search engine is then able to understand:

- What is the question about?
- What is an album?
- What is The Beatles?

Hence, associations with the term Semantic Web are:

- In the Semantic Web, web information has exact meaning.
- The Semantic Web is about relationships between things.
- The meaning of semantic rich web content can be processed and understood by computers.
- Computers can collect, combine, and present information to the user from many different sources in a meaningful way.

1.3 RDF - The Resource Description Framework

RDF [Fra04] is a description language for representing information on the web in a meaningful way. It achieves the idea that things on the web are identified using web identifiers and it is intended to describe information about web resources. Documents written in RDF enable computers to process information a document contains instead of just displaying it. Due to the fact that RDF provides a syntax for representing information, RDF documents can be exchanged by computers between independent applications. RDF can be used to describe resources in the Semantic Web. It is a part of the W3C Semantic Web activity [Fra04].

Resources

With RDF we are able to describe web resources. A resource is anything that can be identified by human beings. This may be a book, an image, a person, a building, a car, an HTML page, a database, or other real-life things. Web resources are related to strings, the so-called URI, i.e. the **U**niform **R**esource **I**dentifier, identifying the resource in the web [Tim].

An RDF URI can lead to any identifiable thing [Fra04]. The most RDF URIs are URLs, i.e. **U**niform **R**esource **L**ocators, which are a specialised kind of URIs and make it able to locate the resource by describing its primary access mechanism, for instance its network location [Tim]. It is not necessary that a described thing identified by an URI is accessible by a browser. Here are some examples:

1. http://www.w3.org/People/EM/contact#me identifies a person, Eric Miller [Fra04].
2. http://www.w3.org/2001/vcard-rdf/3.0# represents visiting card objects.
3. http://dbs.cs.uni-duesseldorf.de/RDF/relational.owl# describes an abstract concept for creating a database, a table object, etc.
4. uuid:04b749bf-3bb2-4dba-934c-c92c56b709df a unique identifier that is created by combining the time and the address of ones ethernet card [Aar].

RDF Metadata Model

RDF provides a simple way to describe information: building most simple statements with a subject, a predicate, and an object [Fra04]. The subject is the resource that is described. An example for that is ″Matthäus Zloch″ in the statement on page 3. The predicate is ″email″ and the object is ″matthaeus.zloch@uni-duesseldorf.de″, respectively. In analogy to RDF, these three basic components of a statement are called resource, property, and property value. Properties are also resources. They might be shown in the XML-based form of a namespace. Property values are represented as a string of characters, but they also can take other resources or datatypes such as integers, as their value [Fra04]. Because of these three parts, a statement is also called a triple.

RDF provides a model for data, the RDF graph. The graph shows resources as an ellipse. An arc labelled with the name of the property points to the corresponding property value. If this property value is a string, it is shown as a rectangle, otherwise as an ellipse. Figure 1 shows the statement on page 3 as an example for the corresponding RDF graph.

A more structured representation is needed, when a property, e.g. "hasAddress", consists of more than one value. The property value is then considered to be a new, virtual resource [Fra04]. In that case, new statements about that new resource can be made.

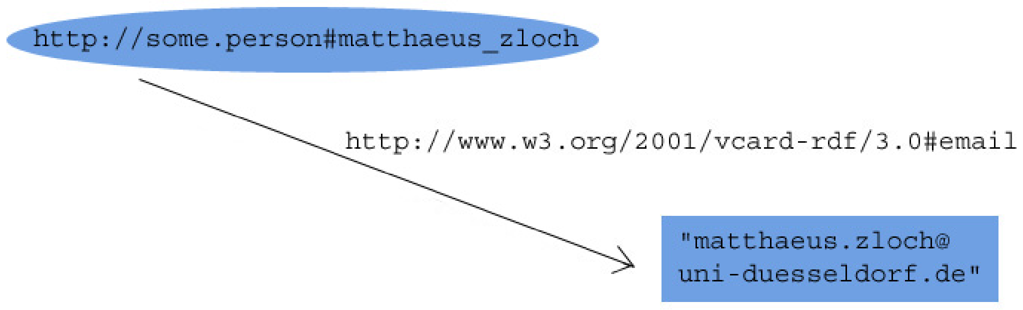

http://some.person#matthaeus_zloch
http://www.w3.org/2001/vcard-rdf/3.0#email
"matthaeus.zloch@
uni-duesseldorf.de"

Figure 1: Statement on page 3 shown as an RDF graph.

In order to record these graphs and enable computers to process the information presented in a graph, RDF documents can be written in XML. The syntax used by RDF is called RDF/XML. This enables the documents to be understood by different computers as well as exchanged through different applications in different operating systems [Fra04]. Each document written in RDF/XML is a legal XML document, but not vice versa.

The corresponding RDF/XML document for the RDF graph above is shown in Figure 2. `email` as a property of the resource `http://some.person#matthaeus_zloch` that is described here, does not need to be determined in a closer way, since the Semantic Web application is able to reason the meaning of it, using the given URI, i.e. `http://www.w3.org/2001/vcard-rdf/3.0#email`.

RDF Schema

RDF Schema (RDFS) is a vocabulary description language and a semantic extension of RDF [Dan04]. As already mentioned, RDF is about describing resources on the web. These resources have properties and their corresponding property values. An interpretation of properties is that they represent relationships between resources.

```
<?xml version="1.0"?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:vCard="http://www.w3.org/2001/vcard-rdf/3.0#">

 <rdf:Description rdf:about="http://some.person#matthaeus_zloch">
   <vCard:email>matthaeus.zloch@uni-duesseldorf.de</vCard:email>
 </rdf:Description>

</rdf:RDF>
```

Figure 2: RDF document for the RDF graph in Figure 1.

The function of RDF Schema is to semantically extend RDF and to provide a mechanism for describing RDF properties and relationships between them. With RDF Schema, the expert is able to establish groups of resources, so-called classes and consequently define resources to be instances that are members of these classes. Furthermore, classes can be defined as domains and ranges for properties [Dan04]. Domain properties limit the objects to which the property can be assigned to, whereas range properties indicate what objects the property can take as its value [MvH04].

RDFS can be treated like dealing with objects in a object-oriented programming language. Class inheritance is a possible alternative to define specialised types. Just like in Java, where each new class is automatically a subclass of the class Object, in RDF Schema each defined class is a subclass of the class rdfs:Resource [Dan04]. In other words, all other classes are subclasses and described resources are instances of the class rdfs:Resource.

Both, RDF and RDFS, give information on the interpretation of web content in a way that computers can process and understand. XML, XML DTD, and XML Schema do not meet this criteria of the Semantic Web [Dan04], since XML only structures information and XML DTD and XML Schema restrict a certain structure of an XML document.

Figure 3 shows a simple example how such class creation and class inheritance would look like.

```
<?xml version="1.0"?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

 <rdfs:Class rdf:ID="person" />

 <rdfs:Class rdf:ID="male">
  <rdfs:subClassOf rdf:resource="#person" />
 </rdfs:Class>

 <rdf:Description rdf:about="http://some.person#matthaeus_zloch">
  <rdf:type rdf:resource="#male" />
 </rdf:Description>
</rdf:RDF>
```

Figure 3: RDFS example. Creating a class and defining a subclass.

Résumé

- The Semantic Web uses RDF to describe web resources.
- RDF provides a syntax and a model for data.
- RDF attaches meaning to information by building. statements or triples with a SPO, i.e. the resource, property, and the property value.
- RDF Schema comes with a vocabulary enhancement to create classes and properties.

1.4 OWL - The Web Ontology Language

OWL is the last part of the growing stack of recommendations related to the Semantic Web [MvH04]. OWL stands for the Web Ontology Language and is built on top of RDF to be used in the Semantic Web. Like RDF, OWL is a description language which was invented with efforts to enable computers not only to display information, but to process their content. OWL entails a greater machine readability and interpretability than RDF Schema does. It provides additional vocabulary and formal semantics for representing semantic rich and interpretable web content [MvH04]. OWL is needed to create ontologies, to describe classes and properties and to represent information about them.

Understanding Definitions

Tom Gruber defined the term ontology in his paper [Gru93], as it is used in computer science: "*An ontology is a description of the concepts and relationships that can exist for an agent*". Referring to OWL, an ontology is the representation of an exact description of terms and their interrelationships. Agents are OWL applications and other automated processes [HPSvH03] that perform reasoning tasks to collect the information they search for. OWL has a greater vocabulary than RDF and RDFS, in order to achieve the vision of machines performing reasoning tasks on formally described terminology in documents [MvH04]. Ian Horrocks gives in [HPSvH03] some examples where ontologies might be used:

- in e-commerce, to simplify the communication between buying and selling OWL applications.
- in search-engines, for reasoning tasks to collect information about web pages that contains semantically similar but syntactically different words.
- in web and grid services, where they can help locating suitable devices.

One extension of OWL compared to RDFS is the logical combination of classes. They can be defined as intersections, unions, disjoint or complements of classes and enumerations of specified objects. Properties can be transitive, functional, or the inverse of other properties [MvH04]. The reader is referred to subchapter 1.5, where an ontology called Relational.OWL is discussed in detail.

The Three Sublanguages

OWL enables to describe not only resources, but also classes with a greater vocabulary than RDF Schema. Using OWL, programmers and ontology creators may select between three sublanguages [MvH04]: OWL Lite, OWL DL, and OWL Full. OWL Lite and OWL DL come with some limitations to facilitate designing ontologies. Since each of them are powerful and sophisticated languages, the following gives a short overview.

- **OWL Full**: Can be seen as an extension of RDF. It provides the maximum expressiveness and the syntactic freedom of RDF, modification of OWL and RDF vocabularies for instance. Reasoning tasks have no guarantee to be computable and perhaps will not finish in a finite time.
- **OWL DL**: Supports maximum expressiveness for building ontologies with a guarantee of computational completeness and decidability of reasoning tasks.
- **OWL Lite**: Is a subset of OWL DL. Provides classification hierarchy and simple constraints, for instance cardinality constraints only in values 0 and 1.

Since OWL Lite and OWL DL can be thought as extensions of restricted views of RDF, every OWL Lite, OWL DL, or OWL Full document is an RDF document as well [MvH04] whereas every RDF document is a OWL Full document. Hence, OWL documents written in RDF/XML syntax can be processed by RDF applications.

In the end of our discussion about the technologies related to the Semantic Web, Table 1 shows the stack of recommendations related to the Semantic Web with a short summarisation as shown in [MvH04].

XML	Provides the syntax for writing structured documents, but the meaning of semantic rich data is not clear.
XML Schema	Enables the programmer to restrict a structure of XML documents. XML Schema additionally defines datatypes.
RDF	First approach of bringing meaning into structured documents. Documents are written in an XML-based syntax. Resources were described by a simple mechanism, i.e. building statements with a resource, a property, and a property value (SPO).
RDF Schema	Vocabulary extension of RDF. Enables the programmer to describe resource properties and the relationship between other properties. A more object-oriented approach by creating own application-specific classes. Allows RDF resources to be defined as instances of classes. A generalisation-hierarchy, i.e. class inheritance is supported.
OWL	Vocabulary and semantic extension of RDF Schema. The programmer is able to create ontologies, i.e. the representation of the exact description of terms and their interrelationship OWL meets the expectations that machines may perform reasoning tasks.

Table 1: Stack of recommendations related to the Semantic Web [MvH04].

The following section introduces a technology which released in the recent years. It is created on top of OWL and goes beyond the stack of recommendations of the Semantic Web shown in Table 1. The technology enables Semantic Web applications to access data stored in relational databases with their own build-in functionality.

1.5 Relational.OWL - A Data and Schema Representation Format

Semantic Web applications are unable to query data stored in relational databases. Access to these data is only possible via SQL, what the majority of Semantic Web applications do not support.

The Representation Format

The idea of Relational.OWL [PC05b] is to extract the data stored in relational databases into OWL, a "widely accepted knowledge representation technique" [PC05b], in order to meet an exchangeable representation format. Additionally, once there is a RDF/OWL representation of data, it will be processable by a wide majority of Semantic Web applications.

Relational.OWL is a technique converting schema and data items of a relational database into an OWL Full ontology. This ontology contains class definitions which are the semantic representation of the corresponding relational model, i.e. databases, tables, columns, etc. Data items as are stored in a relational database were described as instances of this specific ontology describing the schema. Semantic Web applications are even able to query a representation of relational data items using semantic query languages like RDQL without having to use SQL. RDQL is discussed in the following subchapter.

Regarding up-to-date research fields such as Peer-to-Peer databases, a representation of a database with Relational.OWL also achieves this challenging task of a representation format which can be shared and collected by independent Semantic Web applications and different sources. Independent databases systems which are able to process OWL can easily integrate exchanged schema and data items [PC05b].

Concerning the amount of metadata information related to a database an ordinary database system provides, Relational.OWL is not an exact copy of a whole database system. It only conveys the most important information which are needed to store data items in a table [PC05b], i.e.

- Tables and Columns,
- Primary and Foreign Keys and,
- Data Types.

The Relational.OWL representation does not include indexes or triggers since they were not required for an ordinary representation of data.

textfiles/schemaowl.txt

Figure 4: Example of creating a new database class with two tables.

The example in Figure 4 illustrates a simple example using Relational.OWL in order to create an ontology describing one database containing one table with two columns. Line 4 initially defines an XML entity to abbreviate the URL `http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl` to `relowl`. A new class called `MyDatabase` is created in line 12-16. The next line constitutes a property for this class determining its type, i.e. Database which is defined in the related file relational.owl. The following two lines define two more properties for class `MyDatabase`. They call `hasTable` and their value is `#BOOKS` and `#PUBLISHERS`. The `#` signifies that the value of the property is not a resource, but a class which is described in that document. Another class `BOOKS` is created in line 18. The class is of type Table as the next line indicates. Line 20-26 defines the first column for table `BOOKS` by property `hasColumn`. This tag has an additional attribute `rdf:resource` indicating the name of the column that is `#BOOKS.BOOKID`. As mentioned in subchapter 1.3, in the case of a more structured value, as it is here, the property value again is a new "virtual" resource. Here, it has one property indicating the type, i.e. Column, and two more properties called domain and range. Property domain signifies that it only can be applied to instances of type `#BOOKS` whereas property range signify that it only can take instances of type `integer` as its value. Line 27 adds one more column to table `BOOKS`, i.e. `#BOOKS.BOOKNAME`.

Figure 5 shows how such a representation of data items in Relational.OWL would look like. Data items are represented as instances of classes described in the ontology representing the schema in Figure 4. Line 6-9 and 11-14 show two instances of class `BOOKS`. They contain all the information described in the above schema.

textfiles/dataowl.txt

Figure 5: Representation of data items which are instances of classes.

The following section introduces another significant technology for this thesis, i.e. RDQL, a query language for Semantic Web documents written in RDF/XML. RDQL enables Semantic Web applications to query data stored as an RDF graph, e.g. the representation of data items in Relational.OWL.

1.6 RDQL - RDF Data Query Language

RDF/OWL documents are not designed to be read by people [Fra04]. Yet the simple example in Figure 4 shows how sophisticated it is to retrieve the information hold in a graph by just reading the documents. The RDF Data Query Language (RDQL) [Sea04] is one of several languages to query data expressed in the RDF/XML representation format. RDQL operates on triple statements in an RDF graph.

Syntax and Functionality

As mentioned in section 1.3, RDF provides a model for data which is represented as a directed graph representing triples. These triples hold the information of created statements.

Table 2 indicates the SQL-based syntax of RDQL queries. They mainly consist of four parts that are the SELECT, FROM, WHERE, and USING part. The SELECT-clause

SELECT	Variables that should be returned
FROM	Sources to query
WHERE	Constraints for triple patterns
AND	Complex constraints for variables
USING	Namespace definitions

Table 2: RDQL query syntax.

signifies variables required to be in the result set. The FROM-clause indicates the source to be queried. The WHERE-clause respectively is the most important part of the RDQL query. It includes one or more triple patterns, separated by commas, which consist of named variables and RDF values, i.e. URIs or literals [Sea04]. Variables are identified by a preceding question mark ?. They were required to substitute elements of a triple and are primarily not bound to certain values. Triples in the WHERE-clause signify conditions to graph triples. Evaluating the result set, each triple statement of the RDF graph is verified whether it matches these conditions [Sea04]. RDQL treats structured property values, i.e. "virtual" resources also as possible matches.

The AND-clause is required to determine additional and much more complex restrictions for variables in triple patterns. They may be combined using the logical AND or OR-operator. This enables to bound a variable to more than one value. The USING-clause simplifies the usage of URIs with namespaces that will be used for RDF properties.

The example below shows a simple RDQL query:

```
SELECT ?x
WHERE  (?x, vcard:FN, "Max Muster")
USING  vcard for <http://www.w3.org/2001/vcard-rdf/3.0#>
```

Here, the SELECT-clause determines the subject variable, i.e. the resource identified by variable `?x`, to be in the answer set. The WHERE-clause consists of only one constraint. This triple pattern requires the statement in the graph to have predicate `<http://www.w3.org/2001/vcard-rdf/3.0\#FN>` and object `"Max Muster"`. Thus, the result set consists of possible values for resources which describe a person with full name `Max Muster`.

Since RDQL delivers only possible values for bounded variables in the triple pattern [Sea04], we do not obtain an Semantic Web document in form of an RDF/OWL syntax as a result set. Thus, as a conclusion to that, RDQL as a query language is not closed.

Section 3 discusses querying RDF/OWL documents in more detail.

2 Describing the Problem

Due to the fact that data stored in relational databases are unreachable for Semantic Web applications using their own build-in functionality, Relational.OWL achieves the idea to extract schema and data items of a relational database into a common knowledge representation format like OWL. Wanting the Semantic Web application to query relational data or similarly guarantee the up-to-dateness of a semantic representation of data, the first step before querying would be a reapplication of an extraction of schema and data. Otherwise, Semantic Web applications are expected to administrate their own mappings between the relational and the semantic representation, since relational databases are not dynamic in that aspect. D2R MAP by Bizer introduced in [Biz03] is a manual mapping solution between relational data and, among other representation formats, RDF. Unfortunately, these mappings are static and they have to be applied any time the schema changes. To avoid the required extraction progress, the motivation introduced in this thesis is, to simulate the OWL representation of data to the Semantic Web application and actually query the underlying relational representation using SQL. This would guarantee the up-to-dateness of data and we would obtain all features of a relational database system, like lower space requirements, transaction management, high performance, etc. In order to meet this challenge, also illustrated in Figure 6, it is necessary to translate RDQL queries into SQL. Semantic Web applications are then able to access and query data actually stored in relational databases using their own functionality, even without having an underlying OWL representation. It does not make

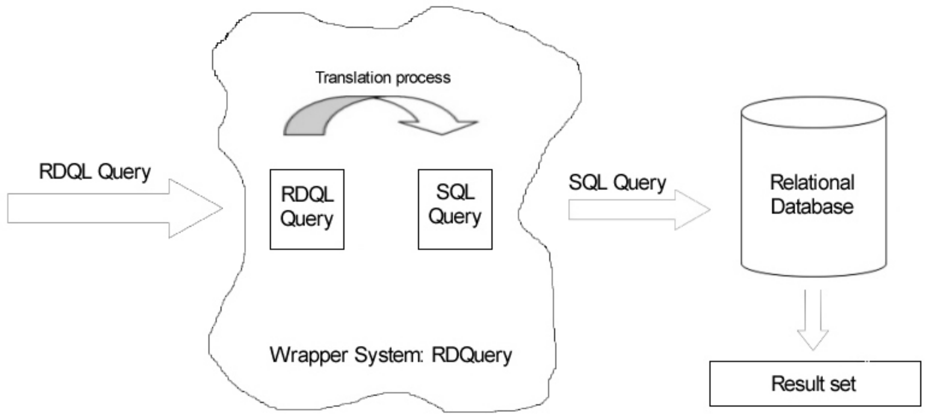


Figure 6: Overview of the functionality of RDQuery.

any difference when the underlying relational database varies, the translation process stays the same. In fact, like querying relational databases with SQL, the user is expected to know the local schema of data he wants to query. Evidently, queries have to be created with respect to the instances of the Relational.OWL ontology.

Based on a Relational.OWL representation of relational data, Pérez de Laborda and

Conrad introduces in [PC05a] the RDQL correspondent for completeness operations of relational query languages [EN04]. The first step is to translate these RDQL queries into SQL with the help of their characterisation, which constitutes the greatest part of the implementation, to guarantee the usage of basic query operations. Using techniques like JavaCC and the Jena Framework to implement that, this afterwards deals as a foundation for an example in form of a graphical user interface that is introduced in chapter 4.5.

3 Translating RDQL Queries into SQL

The following section introduces some possible approaches for translating RDQL queries into their SQL correspondent using their characteristics. Nevertheless, for a better comprehension the reader should already have some basic understanding in relational algebra and the relational query language SQL. All RDQL queries and their translations dealt in the following subsections are based on RDQL queries introduced in [PC05a]. Starting with subsection one and some considerations about characterising simple RDQL queries like a SELECTION or PROJECTION, the next subsections depict programming methods allowing programmers to potentiate the motivation and problem description.

As already mentioned, the first step in trying to translate an RDQL query into a valid SQL statement allowing Semantic Web applications to access relational data, is its characterisation to decide the translation process. To implement that, there have to be made exact distinctions between the five completeness operations shown as RDQL queries. The following description provides an insight into how this characterisations has been done, starting with a SELECTION. The existence of named relations and their related attributes is supposed.

3.1 Selection

In the relational algebra, a SELECTION selects all tuples from a relation matching the SELECTION-conditions [EN04]. Consider the following query:

```
SELECT ?x, ?y, ?z
WHERE  (?x, rdf:type, dbinst:ADDRESS)
       (?x, dbinst:ADDRESS.CITY, "Berlin")
       (?x, ?y, ?z)
USING  rdf for [...] dbinst for [...]
```

We concentrate on the second part of the RDQL query, i.e. the WHERE-clause enclosing three triple pattern conditions. As we see here, the first line is the triple condition

```
(?x, rdf:type, dbinst:ADDRESS).
```

We can conclude from that the type of the requested object, i.e. `dbinst:ADDRESS`. More precisely: the requested object is restricted to have property `rdf:type` with value `dbinst:ADDRESS`. This object conforms the ADDRESS relation in the database. This condition is the analogy to the FROM condition of SQL.
The second line in the WHERE-clause indicates yet another constraint that entails the actual selection identifiable by the concrete property value. Objects with property `ADDRESS.CITY`, again represented by the `?x` variable, have to have a property value of `"Berlin"`. Evidently, this has to be analogous to the WHERE condition clause of SQL. The third condition has no effect on the current translation. It states that the entire set of triples should be selected to match both conditions in this case. This will not be mentioned again in further discussions.
Thus, those objects are taken into the result set that have the requested two properties and their related values.
First conclusions in analysing this SELECTION are:

- `(?x, rdf:type, ex:TABLE)` is analogous to

  `FROM TABLE`

  in SQL.

- `(?x, dbinst:TABLE.COLUMN, "value")` is analogous to

  `WHERE TABLE.COLUMN = "value".`

- The requested objects are identified by the subject variable of the triple, i.e. here `?x`.

Restricting the objects to be of a special type, is a must for querying data stored as a representation of a relational database in Relational.OWL. However, the conditions shown above are typical for a SELECTION. In RDQL, every query having these symptoms can be characterised as at least SELECTIONs. From analysing this simple example, we may conclude that only the WHERE part of an RDQL query holds relevant information needed for a valid translation into an SQL statement.
Now the resulting SQL query can be written down. Since there are no other restrictions to an attribute of a table, we obtain the following SQL statement:

```
SELECT *
FROM   ADDRESS
WHERE  ADDRESS.CITY = "Berlin"
```

3.2 Projection

Another important operation of relational query languages is the ability to choose a subset of columns in a relation [EN04]. The example below signifies a PROJECTION shown in RDQL syntax based on the Relational.OWL representation of data.

```
SELECT ?x, ?y, ?z
WHERE  (?x, rdf:type, dbinst:ADDRESS)
       (?x, ?y, ?z)
AND    (?y EQ dbinst:ADDRESS.CITY) ||
       (?y EQ dbinst:ADDRESS.STREET)
USING  rdf for [...] dbinst for [...]
```

Similar to the SELECTION example in subchapter 3.1, the first condition in the WHERE-clause restricts the result objects to have property `rdf:type` with value `dbinst:ADDRESS`. In SQL, the SELECT-clause is responsible for choosing attributes which will appear in the result set. More than one projection on attributes have to be separated by a comma. In RDQL, this specification has to be done in the AND-clause of the query. As already mentioned, the AND-clause contains more complex constraints for variables. Logical operators like the OR-operator allows us to bind the property, here indicated by the `?y` variable, to more than one value. The example above restricts the object property to be `dbinst:ADDRESS.CITY` OR `dbinst:ADDRESS.STREET`. A conclusion to that:

- Binding the property of a triple to explicit values amounts to a PROJECTION of attributes of a relation. The SQL correspondent for `(?y EQ dbinst:TABLE.COLUMN)` is

  ```
  SELECT TABLE.COLUMN
  ```

Since there were no other restrictions, the translation amounts to

```
SELECT ADDRESS.CITY
FROM   ADDRESS
```

This example shows that the assumption in 3.1 is partly correct. The WHERE-clause holds relevant information needed for a translation as well as the AND-clause. This has to be taken into account while translating the next queries.

3.3 Set Union

In the relational algebra two relations can be unified if they are union-compatible [Dat82], i.e. they have the same number of attributes. The datatypes are irrelevant as long as the cardinality is equal. The resulting relation contains all tuples that either appear in the first, or the second relation, or both [EN04]. The correspondent UNION-operation based on the Relational.OWL representation of data items is shown below as an example.

```
SELECT ?x, ?y, ?z
WHERE  (?x, rdf:type, ?a)
       (?x, ?y, ?z)
AND    ((?y EQ dbinst:STUDENTS.ADDRESSID) ||
        (?y EQ dbinst:ADDRESS.ADDRESSID)) &&
       ((?a EQ dbinst:STUDENTS) || (?a EQ dbinst: ADDRESS))
USING  rdf for [...] dbinst for [...]
```

Here, the first triple pattern of the WHERE-clause `(?x,rdf:type,?a)` does not restrict the object property with name `rdf:type` to an explicit value but a variable. This is because there are two relations needed to execute an UNION-operation. This may only be denoted in the AND-clause where we can use logical operators. Thus, there are two main conditions to variables in the AND-clause. The first condition restricts the objects identified by variable `?x` to have property `dbinst:ADDRESS.CITY` or `dbinst:ADDRESS.POSTALCODE` AND the result set should further contain `dbinst:COUNTRY` and `dbinst:ADDRESS` objects.

Regarding only the Relational.OWL representation of data we definitely can say that two relations obviously exist if the objects are not bound to explicit values but a new variable. Thus, for the time being we may gather from this RDQL query:

- If the property value of a triple with property `rdf:type` is bound to a new variable, this probably indicates a UNION. The necessary table and column names in the AND-clause are bound to variables.

A direct translation of this RDQL query is:

```
(SELECT STUDENTS.ADDRESSID
 FROM   STUDENTS)
UNION
(SELECT ADDRESS.ADDRESSID
 FROM   ADDRESS)
```

3.4 Set Difference

Applying the SET DIFFERENCE-operation (also called MINUS) to two relations, creates a new relation including all tuples that are in the first but not the second relation [EN04]. Referring to the RDQL correspondent for a SET DIFFERENCE-operation shown below, it is again necessary to specify two relations here.

```
SELECT ?b
WHERE  (?a, dbinst:COUNTRY.COUNTRYID, ?b)
       (?a, rdf:type, dbinst:COUNTRY)
```

```
        (?x, dbinst:ADDRESS.COUNTRYID, ?y)
        (?x, rdf:type, dbinst:ADDRESS)
AND     !(?b EQ ?y)
USING   rdf for [...] dbinst for [...]
```

Unlike the UNION-query in subchapter 3.3, with only one object identified by one variable, this query specifies two different objects. Variable `?a` represents objects of type `dbinst:COUNTRY`. `dbinst:ADDRESS` objects are identified by variable `?x`. The WHERE-clause includes one more condition for each of them. The property values of property `dbinst:COUNTRY.COUNTRYID` and `dbinst:ADDRESS.ADDRESSID` are allocated with variables. The AND-clause restricts them as unequal, indicated by the logical NOT-operator `!`.

Characterisations are as follows:

- Two different subject variables represent two different objects.
- The property values allocated with a variable are determined to be unequal.

Like the UNION, the SET DIFFERENCE needs the property value of some triples to be a variable. This is not a disagreement with the characteristic in 3.3. Both regard the property values, but the UNION is more strict. It additionally requires the property to be `rdf:type`.

The translated SQL query would look like this:

```
(SELECT COUNTRY.COUNTRYID
 FROM COUNTRY)
MINUS
(SELECT ADDRESS.COUNTRYID
 FROM ADDRESS)
```

3.5 Cartesian Product

The CARTESIAN PRODUCT or CROSS PRODUCT operation applied on two relations, unifies the whole set of attributes from the first with the second relation. The resulting relation has one tuple for each combination of tuples [EN04]. Referring to RDF, applying the operation on two sets of objects, where the first set has m and the second has n triples, would create m*n new objects where each of them contains the properties of two objects, one of each set. Finding an adequate RDQL query for this operation is more sophisticated since RDQL queries do not return objects in result sets. Pérez de Laborda introduced in [PC05a] the CARTESIAN PRODUCT operation as follows, which is as close as possible to the CARTESIAN PRODUCT operation of the relational algebra.

```
SELECT ?x, ?y, ?z
WHERE  (?x, ?y, ?c)
       (?x, rdf:type, ?a)
AND    (?a EQ dbinst:STUDENTS) || (?a EQ dbinst:ADDRESS)
USING  rdf for [...] dbinst for [...]
```

This query is equal to the UNION query except for the specification of a certain column over which it is unified. The UNION-operation in 3.3 was characterised by a triple that has property `rdf:type` and is bound to a new variable. Hence, a UNION cannot be identified only by this feature. The recognition has to be done afterwards. If the objects represented by variable `?x` are supposed to have one more property such as a column specification for instance, this query turns out to be a UNION. Unfortunately, RDQL does not have a better capability to realise a UNION-operation that is based on Relational.OWL.

Instead of characterising the CARTESIAN PRODUCT, it is more appropriate to extend the UNION characteristics:

- In order to get a recognisable UNION, there have to be specifications for another property, i.e. the property with the column name. Otherwise this will be handled as a CARTESIAN PRODUCT

Finally, the translated SQL query results to:

```
SELECT *
FROM   STUDENTS, ADDRESS
```

3.6 Join

The JOIN is a special operation resulting from an application of a CARTESIAN PRODUCT and a following SELECTION-condition. Thus, the JOIN does not belong to the five completeness operations recommended by Elmasri and Navathe [EN04]. It is additionally discussed here because of the high relevance of the operation for relational databases [EN04].
The RDQL query below is not the exact analogy conforming with the relational algebra operation. To get the equivalent operation it is necessary to query the first result set of the CARTESIAN PRODUCT with a SELECTION-condition, which cannot be done since RDQL is not closed. The following query can express similar constrains.

```
SELECT ?a, ?d, ?e
WHERE  (?a, ?d, ?e)
       (?a, rdf:type, ?c)
       (?x, rdf:type, dbinst:COUNTRY)
```

```
        (?x, dbinst:COUNTRY.COUNTRYID, ?y)
        (?r, rdf:type, dbinst:ADDRESS)
        (?r, dbinst:ADDRESS.COUNTRYID, ?s)
AND     ((?c EQ dbinst:COUNTRY) || (?c EQ dbinst:ADDRESS)) &&
        (?y EQ ?s) &&
        ((?x EQ ?a) || (?r EQ ?a))
USING   rdf for [...] dbinst for [...]
```

As we see here, the WHERE-clause contains three different objects identified by the variables `?a`, `?x`, and `?r` respectively. `?x` and `?r` represent the two relations the JOIN-operation is applied to. `?a` represents the objects that receive the resulting tuples. The main feature of this query are the three objects representing the corresponding relations of the database. The AND-clause also encloses three additional conditions. The resulting object should be of type `dbinst:COUNTRY` or `dbinst:ADDRESS`. The property values of the other objects are indicated by two other variables which are restricted to be equal. Furthermore, the objects are constrained to be equal with the resulting object. Since this query is quite sophisticated, the characteristics of this seem to be:

- For this query it is sufficient to know that a JOIN-operation contains at least three objects and their related variables.
- One of those objects is the one corresponding to the other two objects, here `dbinst:COUNTRY` and `dbinst:ADDRESS`. The needed table names and attributes can be taken from their triple conditions.

Finally the resulting SQL statement with the additional SELECTION which is indicated by the WHERE-clause.

```
SELECT *
FROM   COUNTRY, ADDRESS
WHERE  COUNTRY.COUNTRYID = ADDRESS.COUNTRYID
```

While analysing these six queries the conclusion that was taken in 3.2 where the PROJECTION was discussed is confirmed. Relevant information for a adequate translation of RDQL queries into SQL are only included in the WHERE- and the AND-clause. The characterisation of each particular RDQL query should be sufficient to obtain the corresponding SQL statements. The next chapter introduces some programming methods that were applied to potentiate the gained approaches.

4 Implementation

RDQL queries have characteristics. It is necessary to recognise them to ensure the correctness of corresponding SQL statements. This chapter discusses among programming methods, the way RDQuery handles a translation. Starting with some programming technologies RDQuery deals with, the afterward discussion goes into detail and becomes more exact when the implementation level is achieved.

4.1 Programming Methods

The following section shortly discusses some applied frameworks and the way of their application in the implementation of RDQuery.

4.1.1 The Jena Framework

Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, including a rule-based inference engine [Hew].

Jena includes among other things an implementation of RDQL and an environment for reading and writing RDF in RDF/XML syntax. Precisely because Jena is pure Java and RDQL is one of its Semantic Web query languages, the implementation of RDQuery is also written in this object-oriented language.

Jena provides packages to facilitate working with Semantic Web documents. Instances of the class Query provided by the package com.hp.hpl.jena.rdql represent RDQL queries and defines helpful methods. It supports, among other things, its own RDQL syntax parser.

The access to the WHERE- and the AND-clause of an RDQL query is essential to filter the information contained in an RDQL query. This is guaranteed by instances of class Query invoking two methods:

- getTriplePatterns() returns a List instance that includes triple patterns from the WHERE-clause.
- getConstraints() returns a List instance with constraints that appear in the AND-clause.

Consider the following PROJECTION:

```
SELECT ?x, ?y, ?z
WHERE  (?x, rdf:type, dbinst:STUDENTS)
       (?x, ?y, ?z)
AND    (?y EQ dbinst:STUDENTS.SURNAME) ||
       (?y EQ dbinst:STUDENTS.LASTNAME)
USING  dbinst for <http://www.somewhere.com/RDF/schema.owl#>
```

The elements of the list returned by method getTriplePatterns() concatenated with these of getConstraints() result to the following string. For the sake of clearness, the end of each triple element holds a white-space and each triple is followed by a newline-command.

```
?x @rdf:type http://www.somewhere.com/RDF/schema.owl#STUDENTS
?x @?y ?z
((?y eq dbinst:STUDENTS.SURNAME)||(?y eq dbinst:STUDENTS.LASTNAME))
```

The first method returns triples from the WHERE condition without their brackets and commas between their elements. Predicates, i.e. properties have a preceding `@` symbol and each AND conditions is additionally enclosed by its own brackets. This is the sole job of Jena in RDQuery, to guarantee the access to the WHERE- and the AND-clause of an RDQL query. The next step is to filter the information containd in this concatenated string of both returns. In RDQuery this is done by JavaCC.

4.1.2 JavaCC - The Java Parser Generator

JavaCC [Jav] (stands for Java Compiler Compiler) is a java parser generator. It provides simple mechanisms to read grammar specifications and regular expressions to generate pure Java code out of it. A set of grammar rules used in computer science represent a certain language. A parser accepts matches to grammar specifications by determining the grammatical structure of an input string. Equally it proceeds with regular expressions that represent words, so-called tokens, in that language. Since JavaCC generates pure Java code, it can be easily integrated into classes and applications. Grammar rules were specified in a file with .jj extension in a special syntax. After compilation with JavaCC a .java file is generated. By using semantic actions [AP02] that is writing Java code between grammar specifications it is possible for programmers to react in an individually way to tokens that are recognised by the parser.

In RDQuery, the parser responsible for filtering the information needed to create a valid SQL query is written with JavaCC. As mentioned in the previous section, class Query provided by Jena supports access to the WHERE- and the AND-clause. After invoking the responsible methods the two returned list elements were concatenated to one string. Thus, the parser has to have a grammar specification matching this string. In other words, sentences with grammar in form of the string have to be part of the language the parser defines. The following subchapter introduces a grammar specification for the parser.

4.1.3 Parser Grammar

Figure 7 proposes a grammar specification with respect to the string that results from merging the methods return values, given in EBNF. References enclosed with <> define lexical token expressions which may be expressed as regular expressions.

```
read                     ::= (subject predicate object)+ (constraint)?
subject                  ::= <VAR>
predicate                ::= <AT> (<VAR> | url_or_rdfType)
object                   ::= <var> | <QUOT> (<NAME>)+ <QUOT>
                         | <URL> <HASH> (<NAME>)+
url_or_rdfType           ::= <URL> <HASH> (<NAME>)+ ("." (<NAME>)+)? | <RDFTYPE>
constraint               ::= <LPAR> negation_or_innerExpression <RPAR>
negation_or_innerExpr    ::= "!" <LPAR> innerExpression <RPAR>
                         | innerExpression
innerExpr                ::= <VAR> <EQ> (nestedExpression
                         | conjunc_or_disjuncExpression)
nestedExpr               ::= (<LPAR> conjunc_or_disjuncExpression <RPAR>)
                         | realValue
conjunc_or_disjuncExpr ::= <LPAR> innerExpression <RPAR> moreExpression
moreExpr                 ::= (<AND> <LPAR> innerExpression <RPAR>)+
                         | ( <OR> <LPAR> innerExpression <RPAR>)+
realValue                ::= (<NAME>)+ ":" (<NAME>)+ ("." (<NAME>)+)? | <VAR>

AT                             ::= "@"
EQ                             ::= "eq"
HASH                           ::= "\#"
AND                            ::= "&&"
OR                             ::= "||"
QUOT                           ::= "\""
LPAR                           ::= "("
RPAR                           ::= ")"




RDFTYPE                        ::= "rdf:type"
NAME                           ::= ["a"-"z", "A"-"Z", "0"-"9", "-"]
VAR                            ::= "?" (<name>)+
URL                            ::= "http://" <addr> (<hierarchy>)*
addr                           ::= (<name>)+ ("." (<name>)+)+
hierarchy                      ::= "/" (<name>)+ ("." (<name>)+)*
```

Figure 7: Grammar specification of the parser.

4.2 Implementation Structure

The programme is written with best efforts to unitise program parts as much as possible so that substitution, enhancements, and future work of certain programme parts is ensured. The programme is implemented with efforts to lean on the two basic principles of reusable object-oriented design [GHJV95] which are:

1. Programming to an interface, not an implementation and
2. Favour object composition over class inheritance.

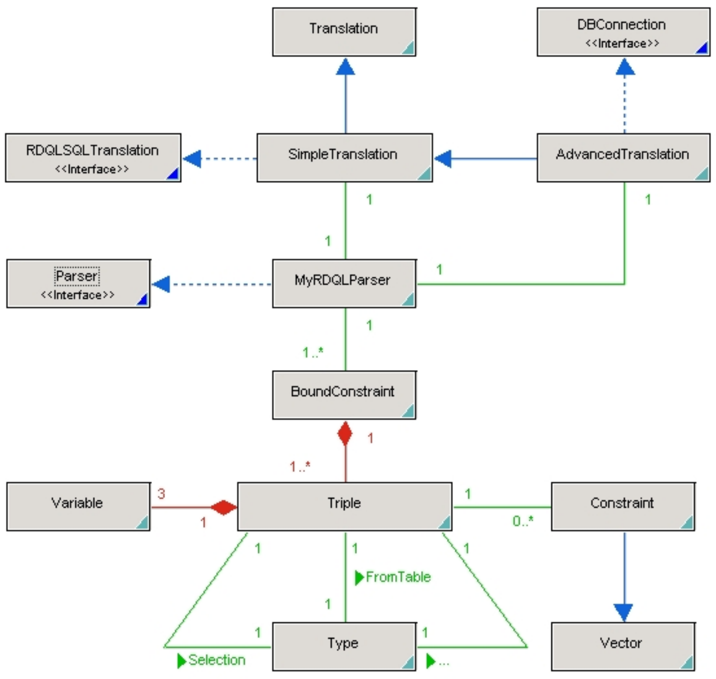


Figure 8: Associations, aggregations, and generalisations of classes in RDQuery.

In RDQuery, the graphical user interface is separated from the main implementation, which constitutes the parser recognising the characteristics of RDQL queries. Associations between the most important classes of RDQuery are shown in a class diagram in Figure 8. Directed arrows represent generalisations whereas links without an arrow represent associations. Associations depicts relationships between concepts of instances of classes. Links with a rhomb at the end represent aggregations defining an "is-a-part-of" relationship.

4.3 Classes and Interfaces of RDQuery

This section shortly describes the most important classes, interfaces, and approaches that RDQuery contains to enable an adequate translation of RDQL queries into SQL. They may be used as a foundation for creating a graphical user interface. The reader is referred to the Javadoc of the implementation of RDQuery for a full documentation. Classes are described in relation to their package. In order to avoid misunderstanding variables and method names are printed in **bold**. The next subchapter contains an example of how these classes and methods may be used.

4.3.1 Package <rdquery.translation>

RDQLSQLTranslation

This is an interface responsible for providing get-methods that enable the access to translation results. Classes implementing this interface have to implement these methods:

- void **translate(String rdqlQuery)**
- String **getSQLQuery()**
- String **getParserDetails()**
- String **getSQLResultsAsRDF()**

The motivation in defining this interface with these methods is in particular based on the idea to enable programmers to decide how they will implement method **translate(String rdqlQuery)**.

DBConnection

This is an interface offering methods for a database connection in order to query the database with the translated query. These methods are:

- void **connect(DBInformation data)**
- void **queryDatabase(String sqlQuery)**
- void **setSQLQuery(Object query)**

The argument of type DBInformation in the first method is a simple class defining variables representing information that are needed to connect a database such as username, password, server, etc.

Translation

There are two kinds of translations possible. The first one is a "simple translation" translating the RDQL query into its SQL correspondent. The second one is a "full" translation with a following database query. In order to distinguish between these kinds of translations, Translation is an abstract class that provides some useful variables and methods that may be used by both of them such as:

- String **getTranslationStatus()**
- boolean **isCorrect()**
- String **accessConstraintClauses(String query)**

The method **accessConstraint(String query)** is declared as static to be invoked without instantiating the class.

SimpleTranslation and AdvancedTranslation

These classes extend the abstract class Translation. SimpleTranslation and AdvancedTranslation both implement two different translation interfaces and Runnable. SimpleTranslation additionally implements RDQLSQLTranslation and AdvancedTranslation additionally implements DBConnection, respectively. Runnable enables instances of SimpleTranslation or AdvancedTranslation to be treated as separate processes.

4.3.2 Package <rdquery.translation.interpretation.rdqlQuery>

This package contains many classes that all have dealings with RDQL queries. Each of them represent a feature of an RDQL query.

In RDQuery, characteristics of certain RDQL queries as discussed in chapter 3, are handled as types. Each triple in the WHERE-clause contained in a RDQL query has a certain type.

```
1. (?x, dbinst:STUDENTS.NAME, "Billy Idle")
2. (?x, rdf:type, dbinst:STUDENTS)
3. (?x, ?y, ?z)
```

The first example is of type SELECTION. In RDQL, the coinciding operation to a PROJECTION is actually done by the AND-clause of the query, see chapter 3.2. For this reason every triple type except for the PROJECTION triple feature, has its own class which is defined in package <rdquery.translation.interpretation.rdqlQuery.types>. Triples that look like the third example above are of type Standard. Each of them has its own variables related to the operation. For instance, SELECTION has a variable called **whereConstraint** of type String which represents the whole argument for the WHERE-clause of an SQL statement.

Triple

The class Triple is self-explanatory. It represents an RDF triple and contains six variable declarations:

- Three variables for each triple element, i.e. **subject**, **predicate**, and **object**.
- One variable that indicates the triple type, for instance SELECTION, etc.
- One variable of type Constraint.

Constraint is a simple class representing a constraint in the AND-clause of an RDQL query. The variable name to which the constraint refers to is represented by the class variable called **variable**. Triple objects have a method called **checkVariableExistence(Variable variable)**. Invoking this method entails a verification whether the presented variable is one of the three elements of the triple.

BoundConstraint

An instance of class BoundConstraint stores triples which occur in the WHERE-clause of an RDQL query like the triples mentioned below which indicate a typical SELECTION:

```
(?x, rdf:type, db:ADDRESS)
(?x, ex:ADDRESS.CITY, "Berlin").
```

In particular, BoundConstraint objects are characterised by the **subject** variable which is the same for ALL triples. They also have a **type** which is determined by the most dominant triple type stored in a BoundConstraint. For instance, a BoundConstraint object containing these two triples above would be of type SELECTION.

4.3.3 Package <rdquery.translation.interpretation.parser>

Parser

This is an interface that may be implemented by any kind of parser class. It provides an object of type Vector called **boundConstraint** and two methods **getBoundConstraints()** and **getParserMessages()**, both returning a Vector. **boundConstraint** contains Bound-Constraints that were recognised by the parser. The purpose of **getBoundConstraints()** and **getParserMessages()** is self-explanatory.

FilteredArgument

This class is a representation of concatenated strings. They represent the arguments for SELECT-, FROM-, or WHERE-clauses. One complete string argument for the SELECT-clause for instance may look like this:

```
SELECT ADDRESS.street, ADDRESS.city
```

BoundConstraintInterpreter

In fact, this class can be handled as an example of filtering the parsed information contained in a BoundConstraint and creating an SQL statement of it. The method **interpret()** interprets the information stored in triples. Due to the fact that triples are stored in BoundConstraints, one constructor of BoundConstraintInterpreter accepts objects of type BoundConstraint. The interpretation of a vector containing BoundConstraints by the example of a SELECTION proceeds as follows: an if-statement verifies if there is one BoundConstraint stored in the vector that was presented to the constructor. If this is the case, it checks the variable **type** whether it is an instance of class SELECTION. If this is correct again, this instance of class SELECTION provides needed information stored in variable **whereConstraint**.

4.3.4 Package <rdquery.translation.interpretation.parser.rdqlParser>

This package provides the essential implementation part of RDQuery, the parser and its corresponding classes. The grammar introduced in chapter 4.1.3 in combination with semantic actions is implemented in a .jj file. Compiling it with JavaCC it generates a .java file that is a part of this package as well. The activity diagram in Figure 9 displays in a simplified way the behaviour of the parser used by RDQuery while the input, i.e. the concatenated string, is being interpreted. As an example the figure shows the triple that is equivalent to the FROM condition of an SQL statement.

The parser acts as follows: The first action is to create a new Triple instance. As the grammar specification in chapter 4.1.3 shows, the first token to be accepted is the subject variable. Semantic actions enable the programmer to save the recognised token before the next one is being analysed. Hence, the subject variable of the input will be saved in the current Triple object. After this, the token is absorbed by the parser and the recognition goes on until it notices the end of the first triple of the input. The Triple instance of the parser will now be added to the Vector containing BoundConstraints. A BoundConstraint is identified via the subject variable of its triples since it is the same for each of them. If the subject variable of the Triple differs from these of all BoundConstraints, a new instance of BoundConstraint is created. The Triple alternatively is added to that BoundConstraint to which it belongs. The recognition goes on starting

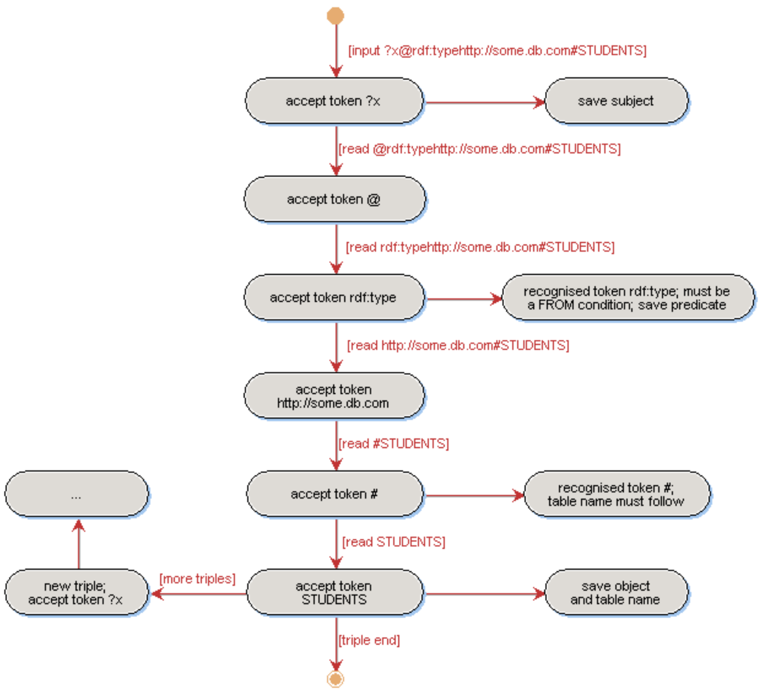

[input ?x@rdf:typehttp://some.db.com#STUDENTS]
accept token ?x
save subject
[read @rdf:typehttp://some.db.com#STUDENTS]
accept token @
[read rdf:typehttp://some.db.com#STUDENTS]
accept token rdf:type
recognised token rdf:type; must be a FROM condition; save predicate
[read http://some.db.com#STUDENTS]
accept token http://some.db.com
[read #STUDENTS]
...
accept token #
recognised token #; table name must follow
[read STUDENTS]
new triple; accept token ?x
[more triples]
accept token STUDENTS
save object and table name
[triple end]

Figure 9: Parser behaviour while reading the input.

with the subject variable and again a new Triple instance is created. The reader is referred to the Javadoc for a more detailed explanation of the parser.

4.4 Using Classes and Interfaces

This chapter exemplifies how classes and interfaces may be used to translate an RDQL query. This example is very simple. It is assumed that the RDQL query already has correct syntax and a simple translation without a following database query is asked. The first line in Figure 10 signifies that the class implements the interface

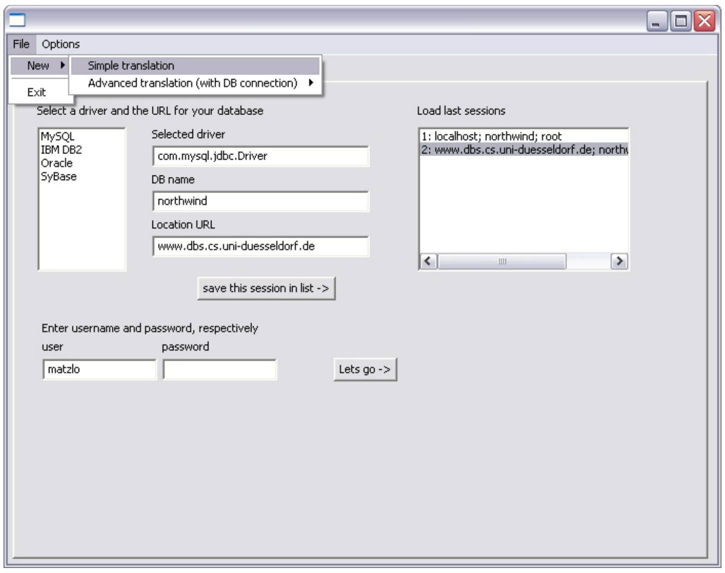
textfiles/TranslationExample.txt

Figure 10: Class that implements the interface RDQLSQLTranslation.

RDQLSQLTranslation, i.e. four methods **translate(String query)**, **getSQLString()**, **getParserDetails()**, and **getSQLResultsAsRDF()** havr to be implemented. The first one is **translate(String query)** in line 5 which is the most important method executing the parser. Line 6 invokes the static method **accessConstraintClauses(String query)** in order to receive a string that is a combination of the WHERE- and AND-clause of the RDQL query. With this string, line 8 creates a new StringReader instance that will be presented to the constructor of InterpretationParser in line 9. Starting the parser is done by invoking method **read()** in the next line which is enclosed by a try-catch-block if the parsing fails. The method in line 14 is responsible for returning the translated SQL statement. When presenting an instance of type Parser to an BoundConstraintInterpreter, the constructor automatically gets the important attributes from it. He calls up the method **interpret()** to filter the information. The SQL string is returned invoking method **getSQLString()**.

4.5 Example - Graphical User Interface

RDQuery has a graphical user interface that deals as an example to the implementation. It partly implements and particulary uses the classes which are discussed in the previous section. The GUI is implemented with SWT [SWT], the Standard Widget Toolkit that provides *efficient, portable access to the user-interface facilities of the operating systems on which it is implemented* [SWT]. It distinguishes between two kinds of translations. The user may choose whether he intends to translate the query with an afterward database connection or not.


File
Options
New
Simple translation
Advanced translation (with DB connection)
Exit
save this session in list ->
Lets go ->

Select a driver and the URL for your database

MySQL
IBM DB2
Oracle
SyBase

Selected driver

com.mysql.jdbc.Driver

DB name

northwind

Location URL

www.dbs.cs.uni-duesseldorf.de

Load last sessions

1: localhost; northwind; root
2: www.dbs.cs.uni-duesseldorf.de; northw

Enter username and password, respectively

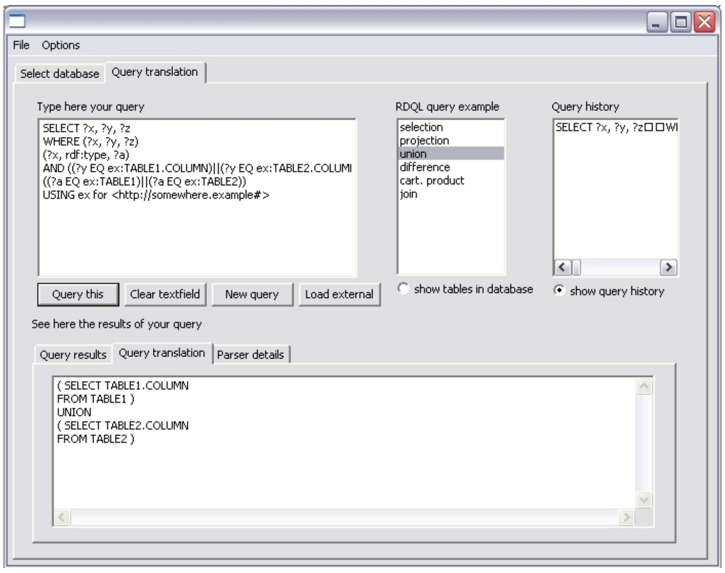
user

matzlo

password

Figure 11: Graphical user interface of RDQuery. Specifying database information.

Figure 11 shows the first screen view of RDQuery where database information such as the database driver, database name, URL, etc. may be entered. Information about databases that are often queried may be saved in the list of sessions located on the right side of the



Type here your query

```
SELECT ?x, ?y, ?z
WHERE (?x, ?y, ?z)
(?x, rdf:type, ?a)
AND ((?y EQ ex:TABLE1.COLUMN)||(?y EQ ex:TABLE2.COLUMI
((?a EQ ex:TABLE1)||(?a EQ ex:TABLE2))
USING ex for <http://somewhere.example#>
```

RDQL query example

selection
projection
union
difference
cart. product
join

Query history

SELECT ?x, ?y, ?z□□WI

See here the results of your query

```
( SELECT TABLE1.COLUMN
FROM TABLE1 )
UNION
( SELECT TABLE2.COLUMN
FROM TABLE2 )
```

Figure 12: Graphical user interface of RDQuery. The main screen for translating queries.

screen.

Figure 12 shows the main screen of RDQuery, the translation tab, consisting in fact of two parts. The upper part represents RDQL-related information and options. The left window is intended for the RDQL query input. In order to simplify creating RDQL queries the middle list contains six frequent operations expressed in RDQL. The user may choose via two radio buttons whether the box on the right side lists either tables of the database or queries already executed. The lower part is intended for the output of the result sets. The first tab lists the result set the database returns after querying it with the translated query. Figure 12 shows the second tab as activated. It indicates the translated SQL query. Statements the parser made while filtering the RDQL information are listed in tab three. Figure 13 shows an activity diagram that illustrates the event beginning

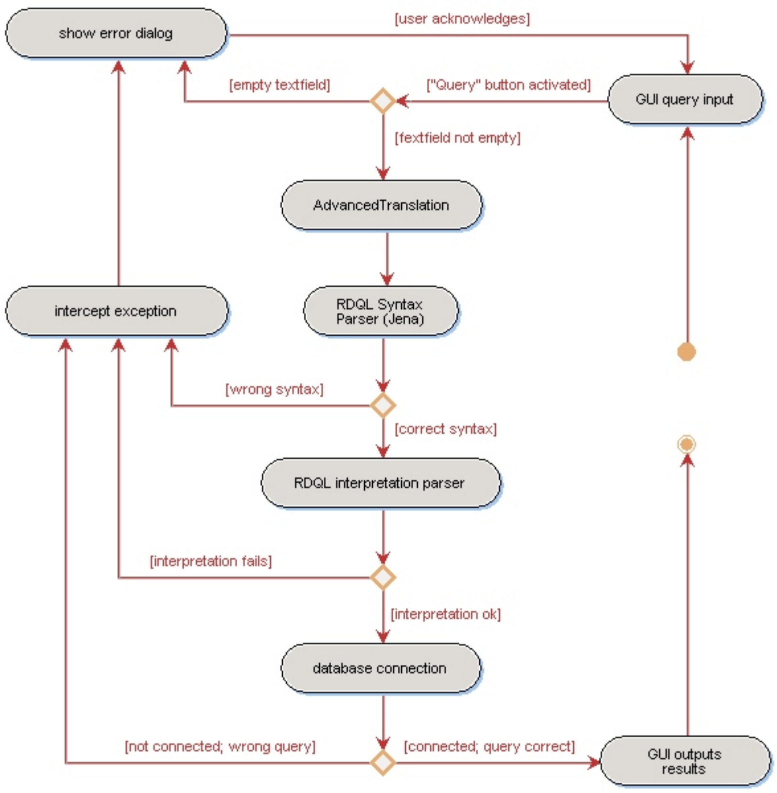


Figure 13: Activity diagram showing the full translation process.

with the query input until it reaches the final result set from the database. Diamonds with yellow edges represent so-called decision knots.

5 Summary and Future Work

This bachelor thesis is about a wrapper system called RDQuery. RDQuery takes RDQL queries that are expressed with respect to the Relational.OWL ontology and translate them into an adequate SQL statement. The motivation is to enable Semantic Web applications to get access and query relational data stored in relational databases with their own build-in functionality. The main advantage of this procedure is, Semantic Web applications are not necessary to have an underlying OWL document that deals as a representation of relational schema and data items to query relational data. Furthermore, Semantic Web applications are not required to administrate their own mappings between a relational representation and the semantic representation of data to ensure the up-to-dateness of the semantic representation for instance.

With the completeness operations of relational query languages expressed as RDQL queries as a groundwork, the major challenge in the beginning of the implementation was to characterise them to ensure the correct translation of basic query operations. After finding their features that were sufficient to gain an SQL statement obtaining the same results as its RDQL correspondent, we conclude that only the WHERE- and the AND-clause of an RDQL query hold essential information for a translation. Class Query provided by Jena provides methods which enable us to get access to these parts of a query. After that, the result sets of this methods were concatenated to one string. The greatest part of the implementation is the filtering of relevant information out of this string. JavaCC enables us to analyse the grammatical structure of a sequence of characters specifying a grammar. It generates a parser that reads the string, analyses its structure, and accepts matches to its grammar. Hence, the correct recognition of query characteristics and the retrieval of relevant information is guaranteed. Afterwards a graphical user interface was created that deals as an example to the implementation.

RDQuery is definitively on its way to be expanded with other semantic query languages, like SPARQL for instance. SPARQL provides a SQL based syntax as well and is more powerful in its expressiveness, regarding the difference between the UNION and the CROSS PRODUCT operation for instance. Simultaneously, some considerations are required to improve the recognition of variable constraints in the AND-clause. RDQuery is currently not able to recognise mathematical inequations like greater than (>) or less then (<). Further on, for the time being RDQuery does not support the transformation of SQL result sets into a Semantic Web representation of data, i.e. RDF/OWL. This would promote RDQuery to a powerful alternative for accessing relational data using Semantic Web methods.

References

[Aar] Aaron Swartz. The Semantic Web (for web developers). Available at http://logicerror.com/semanticWeb-webdev (last visit: 25. October 2005).

[AP02] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, USA, 2002.

[Biz03] Christian Bizer. D2R MAP - A Database to RDF Mapping Language. In *The Twelfth International World Wide Web Conference*, Budapest, May 2003.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.

[Dan04] Dan Brickley and Ramanathan V. Guha. Rdf schema, February 2004. http://www.w3.org/TR/2000/CR-rdf-schema-20000327 (last visit: 11. October 2005).

[Dat82] Christopher J. Date. A formal definition of the relational model. *SIGMOD Record*, 13(1):18–29, September 1982.

[EN04] Ramez A. Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 4th edition edition, 2004.

[Fra04] Frank Manola and Eric Miller. The resource description framework (rdf), February 2004. Available at http://www.w3.org/TR/2004/REC-rdf-primer-20040210/ (last visit: 11. October 2005).

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[Hew] Hewlett-Packard Development Company. Jena a semantic web framework for java. Available at http://jena.sourceforge.net (last visit: 13.October 2005).

[HPSvH03] I. Horrocks, P. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[Jav] Javacc - java compiler compiler. Available at https://javacc.dev.java.net (last visit: 13. October 2005).

[MvH04] Deborah L. McGuinness and Frank van Harmelen. Owl web ontology language overview, February 2004. Available at http://www.w3.org/TR/2004/REC-owl-features-20040210/ (last visit: 11. October 2005).

[PC05a] Cristian Pérez de Laborda and Stefan Conrad. Querying relational databases with RDQL. In *Berliner XML Tage 2005*, pages 161–172, Berlin, Germany, September 2005.

[PC05b] Cristian Pérez de Laborda and Stefan Conrad. Relational.OWL - A Data and Schema Representation Format Based on OWL. In *Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005)*, volume 43 of *CRPIT*, pages 89–96, Newcastle, Australia, 2005. ACS.

[Sea04] Andy Seaborne. RDQL: A query language for RDF, January 2004. Available at http://www.w3.org/Submission/2004/SUBM-RDQL-20040109 (last visit: 12. October 2005).

[SWT] Swt: The standard widget toolkit. http://www.eclipse.org/swt/.

[Tim] Tim Berners-Lee and Network Working Group. Uniform resource identifier (uri). Available at http://www.ietf.org/rfc/rfc3986.txt (last visit: 24. October 2005).

[W3] W3 Schools. Semantic Web Tutorial. http://www.w3schools.com/semweb/default.asp (last visit: 11. October 2005).

[W3C] W3C. W3C Semantic Web Activity. Available at http://www.w3.org/2001/sw (last visit: 11. October 2005).

List of Figures

List of Tables