Uppsala Student Thesis
Computing Science No. 289
2005-01-07
ISSN 1100-1836

Semantic Web Queries to a Mediator System

Master´s thesis by

Josef Schindler

Advisor and Supervisor: Tore Risch

January 2005

Uppsala Database Laboratory
Uppsala University
P.O. Box 513
S-751 20 Uppsala
Sweden

Abstract: *This report describes how to extend the functional mediator system Amos II (Active Mediator Object System) to allow queries expressed in the query language RDQL be processed over RDF-data (Resource Description Framework) through Amos II. A new embedding to Amos II is proposed, which translates RDQL-statements to AmosQL query strings (the Amos II query language), which, when executed, evaluates the queries over RDF-data accessible through Amos II. The parser itself is written by using JavaCC, a Java based parser generator.*

Contents

1 Introduction

This work is done to allow queries expressed in *RDQL* (Resource Description framework data Query Language sec. 2.4) to be processed over *RDF*-data (Resource Description Framework sec. 2.3) provided through *Amos II* (Active Mediator Object System [4][5]).

> It is getting increasingly difficult to get the desired information in modern free-text search methods as provided by, e.g., GOOGLE. One needs to access many different kinds of data to get the useful information. Especially if one wants to combine web resources with other kinds of data stored outside the web like enterprise databases this data retrieval problem gets even worse [1].

The *semantic web initiative* [2] provides Internet-wide standards for semantically describing web data. Any web resource can be annotated with *properties* describing its structure using the standards RDF [3][10] and RDF-Schema [9]. This facilitates guided search of web resources in terms of these properties, which are represented as sets of RDF *statements*. These are triples containing a web resource, a property and a value (subject, verb, object).
HP-Labs has developed a Java based package, Jena2 [7], for writing semantic web applications. Different kinds of back-ends can be connected to Jena2 for transparently storing RDF-data persistently given Jena´s Java-API. RDQL is an RDF-based query language and a Jena2 API for RDQL has been defined [6].

The extensible functional multi-database system Amos II allows to execute object-oriented and functional queries over federations of databases distributed over the Internet. There are interfaces between Amos II and Java documented in [11]. Either a Java program calls Amos II through the *callin* interface, or foreign AmosQL functions are defined by Java methods through the *callout* interface. This permits the development of access modules to external data sources, called *wrappers*[8]. This functionality allows transparent queries to views over combinations of different kinds of back-end data sources, so called *mediators*. The mediators combine the underlying data sources in the needed way to offer a high-level abstraction. It simplifies accessing heterogeneous data sources at the application level. For example, wrappers for ODBC based relational databases, CAD systems, Internet search engines, XML documents or MIDI files have been implemented[8].
AmosQL is the query language based on Amos II. It is a relationally complete object-oriented language.

This report will describe the development of a translator and the embedding into Amos II, which allows to translate RDQL queries into AmosQL and execute them. The now existing *RDQLFront system* has been developed with a Java based parser generator, called JavaCC (Java Compiler Compiler). Exactly the same RDQL grammar [12], which is used in the Jena2 package, is also used in this project to make sure that every RDQL query is recognized

by the parser. After recognizing the RDQL query, it is translated into the appropriate AmosQL query, which is then executed in Amos II.
Now, Amos II supports RDQL access to RDF sources in general.

This report is introduced by an overview of background knowledge of related technologies from databases in general to the special Java framework for writing Semantic Web applications — Jena2. Question like: 'What is *Semantic Web* and *RDF*?', are detailed explained in this part of the report. Section 3 explains the present work, its architecture, implementation, and evaluation, followed by a short summary. Detailed information about the RDQL grammar and all test queries to verify the RDQL–front system are in appendix A and B.

2 Background

This chapter gives an overview of the related technologies whose knowledge is necessary for the project. There might be some few other things which are not explained in detail here but I assume to have this knowledge background.

2.1 Databases, query languages, SQL

> A *database* is nothing more than a collection of information that exists over a long period of time [21].

In most cases, information is structured in tables, and queries manipulate and combine tables. The database schema is a group of table definitions possibly containing information. A *Database Management System (DBMS)* provides an interface to these tables and implements mathematical theory for manipulating the tables.
The query language is necessary to support the declarative specification of both data manipulations. It allows manipulation and retrieval of data from a database. Figure 1 shows the concept of a database system.

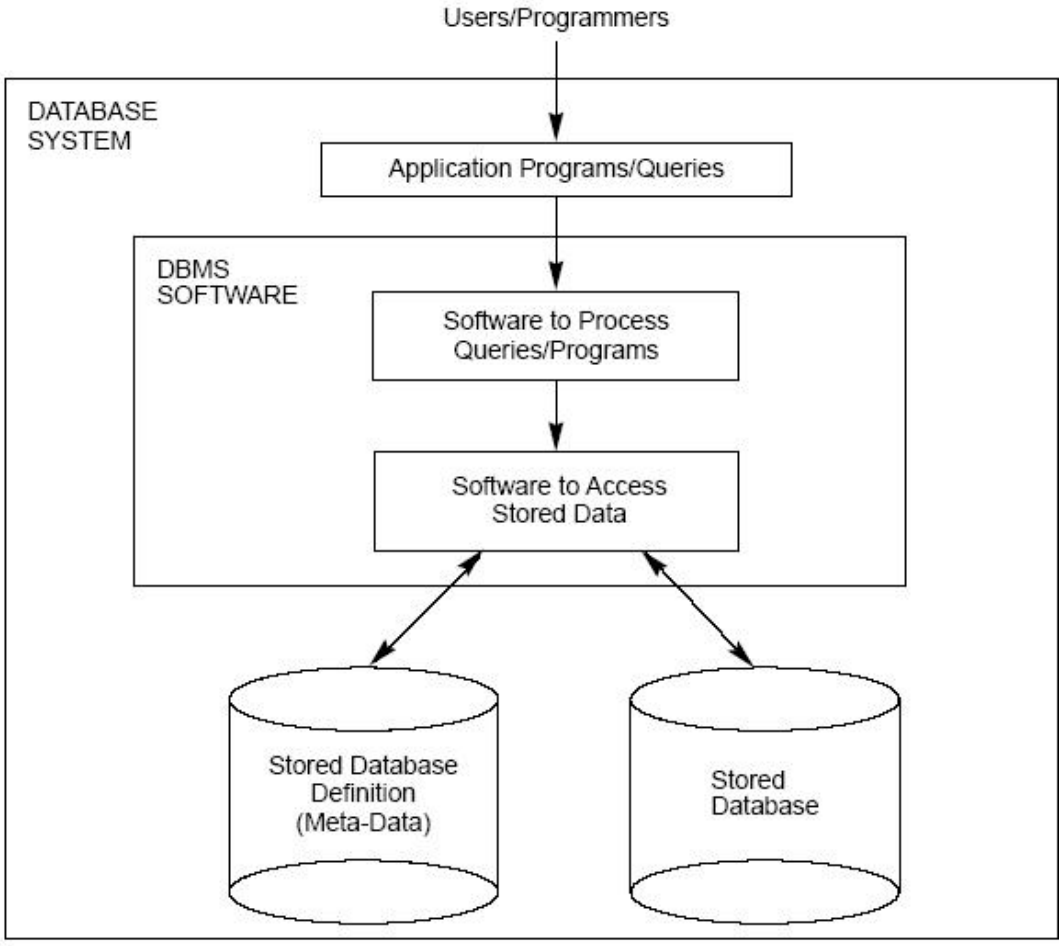

Figure 1: source: [22]

SQL (Structured Query Language) is an ANSI (American National Standards Institute) standard computer language. There are many different versions of the SQL language, but to be in compliance with the ANSI standard, they must support the same major keywords in a similar manner (such as *SELECT, FROM, WHERE, UPDATE, DELETE, INSERT*).

2.2 Semantic Web

The *Semantic Web* is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation. It is based on the idea of having data on the Web defined an linked such that it can be used for more effective discovery, integration, automation and reuse across various applications.
The Semantic Web provides an infrastructure that enables not just web pages, but databases, services, programs, sensors, personal devices, and even household appliances to both consume and produce data on the web. Software agents can use this information to search, filter and prepare information on the web [13].
Also the Semantic Web consists of resources and links. However, now the resources and links can have types which define concepts that tell a bit more to the machines.
The rules *Semantic Web* is based on, called Semantic Web Main Principles as described in [14], follow now:

1. Everything can be identified by URI´s
2. Resources and links can have types
3. Partial information is tolerated
4. There is no need for absolute truth
5. Evolution is supported
6. Minimalist design

The Semantic Web principles are implemented in the layers of Web technologies and standards. The layers are presented in Figure 2.

The Unicode and URI layers make sure that we use international characters sets and provide means for identifying the objects Semantic Web. The XML layer with namespace and schema definitions make sure that the definitions with the other XML based standards can be integrated in the Semantic Web. With RDF and RDFSchema (RDFS) (see sec. 2.3) it is possible to make statements about objects with URI´s and define vocabularies that can be referred to by URI´s. This is the layer where types can be given to resources

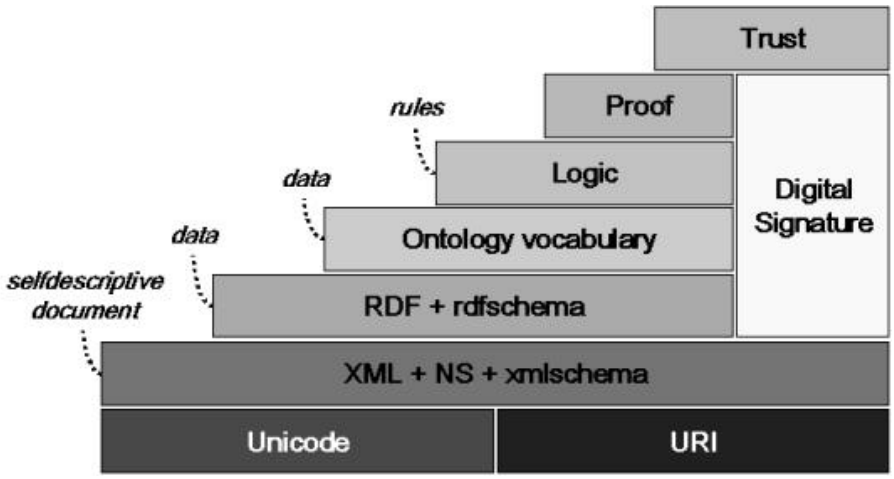

Figure 2: The Semantic Web layers: The Logic layer enables the writing of rules while the Proof layer executes the rules and evaluates together with the Trust layer mechanism for applications whether to trust the given proof or not. *Source: [14]*

and links. The Ontology layer supports the evolution of vocabularies as it can define relations between the different concepts. The Digital Signature layer is used to detect alterations to documents.

The Semantic Web provides a common framework that allows data to be shared an reused across application, enterprise and community boundaries. It is based on the Resource Description Framework (RDF) (sec. 2.3), which integrates a variety of applications using XML for syntax and URIs for naming.
The present work is only based on RDF, not on RDFSchema.

2.3 Resource Description Framework (RDF)

> The Resource Description Framework (RDF) is a general-purpose language for representing information in the Web [15].

It is intended for representing metadata about Web resources., such as the title, author, and modification data of a Web page, or the availability schedule for some shared resource. RDF can e.g. be used to represent information about things that can be identified on shopping facilities (e.g., information about specifications, prices, and availability), or the description of a Web user´s preferences for information delivery.

RDF provides a common framework for expressing information so it can be exchanged between applications without loss of meaning. The ability to exchange information between different applications means that the information may be made available to applications other than those for which it was originally created.

The idea of RDF is to identify things using URIs, and describe resources in terms of simple properties and property values. This enables RDF o represent simple *statements* about resources as a graph of nodes and arcs representing the resources, and their properties and values. Figure 3 should explain this clearer[1].

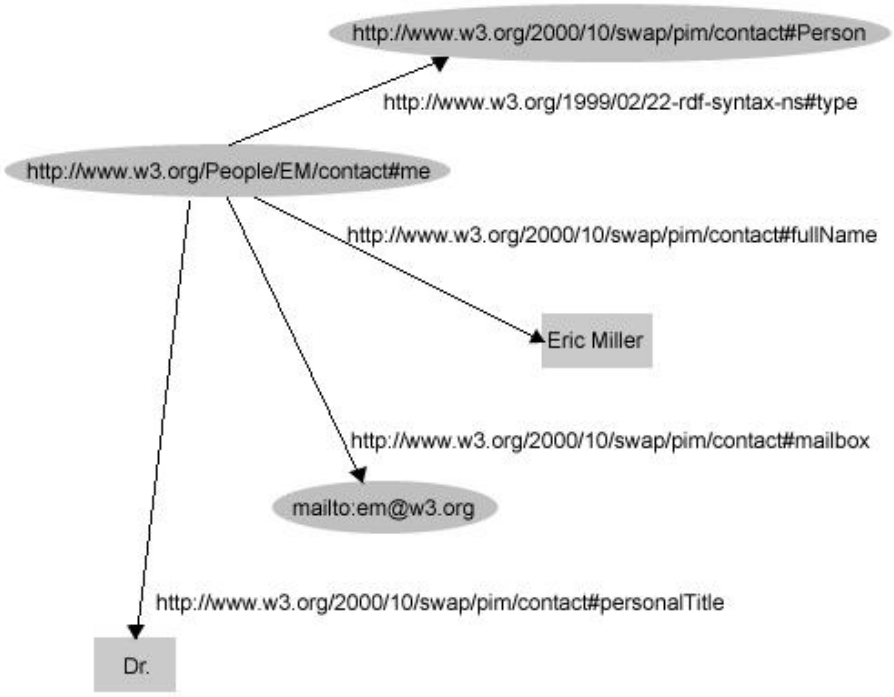

Figure 3: RDF graph: There is a person identified by http://www.w3.org/People/EM/contact#me, whose name is Eric Miller, whose email address is em@w3.org, and whose title is Dr.

- individuals, e.g., Eric Miller, identified by http://www.w3.org/People/EM/contact#me
- kinds of things, e.g., Person, identified by http://www.w3.org/2000/10/swap/pim/contact#Person
- properties of those things, e.g., mailbox, identified by http://www.w3.org/2000/10/swap/pim/contact#mailbox
- values of those properties, e.g. mailto:em@w3.org as the value of the mailbox property (RDF also uses character strings such as "Eric Miller", and values from other datatypes such as integers and dates, as the values of properties)

[1]Source: [16]

RDF uses a particular terminology for talking about the various parts of statements. The part that identifies the thing the statement is about is called the *subject*. The part that identifies the property or characteristic of the subject that the statement specifies is called the *predicate*, and the part that identifies the value of that property is called the *object*. Subject, predicate, object (s,p,o) are together the so called RDF-triples.

RDF-Schema [15] is a framework defined in terms of RDF that allows properties to be standardized for different application domains. It is RDF´s vocabulary description language. Various RDF-standards for different kinds of web documents have been developed, the most well known is *Dublin Core* (http://dublincore.org/).
Amos II includes the possibility to access and query such RDF and RDF-Schema based meta-data descriptions [18].

2.4 RDF Data Query Language (RDQL)

RDQL is a query language for RDF based on SquishQL[2]. It extracts information from RDF graphs. This section gives a quick description of the key elements of the query language with some examples. The grammar of RDQL is in appendix A.
As in section 2.3 explained is an RDF model a graph, often expressed as a set of triples. An RDQL query consists of a graph pattern, expressed as a list of triple patterns. Each triple pattern is comprised of named variables and RDF values (URIs and literals). An RDQL query can additionally have a set of constraints on the values of those variables, and a list of the variables required in the answer set.
RDQL queries are in the form:

SELECT *vars*
FROM *document*
WHERE *Expressions*
AND *Filters*
USING *Namespace declarations*

This translator is restricted to only one document in the FROM-clause.

Example 1:
SELECT *?x,?y*
FROM *<http://example.com/sample.rdf>*
WHERE *(?x,<dc:name>,?y)*USING *dc for <http://www.dc.com#>*

This would return all the ?x-?y tuples indicating the resource name and the value of the dc:name property for each resource [19].

[2]SquishQL is a simple, SQL-like, triples-based query language for RDF, which is designed to be human readable ('SQL-ish').

Example 2:
SELECT *?x*
FROM *<http://example.com/sample.rdf>*
WHERE *(?x, <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,*
<http://example.com/someType>)

This triple pattern matches all statements in the graph that have predicate http://www.w3.org/1999/02/22-rdf-syntax-ns#type and object http://example.com/someType. The variable "?x" will be bound to the label of the subject resource. All such "x" are returned [6].
The example query above had just one triple pattern forming a single edge in the graph pattern. More complicated graph patterns are made by writing all the edges in the query.

Example 3:
SELECT *?family , ?given*
FROM *<http://example.com/sample.rdf>*
WHERE *(?vcard vcard:FN "John Smith")*
(?vcard vcard:N ?name)
(?name vcard:Family ?family)
(?name vcard:Given ?given)
USING *vcard FOR <http://www.w3.org/2001/vcard-rdf/3.0#>*

This query finds the family name and given name from any vcards with formatted name (FN) "John Smith". Here we see also the use of the USING clause. The prefix "vcard" is used to abbreviate the URI or URIref [6]. Writing the full URI or writing the abbreviated form is the same query as RDF only deals with full URIrefs.

RDQL was first released in Jena 1.2.0 (see subsection 2.5). Appendix A shows the grammar which is derived from the Jena implementation of RDQL[3]. This grammar is exactly as is implemented in the translator in this project, so every possible variation of RDQL query can be recognized by the parser.

2.5 Jena2

Jena is a Java framework for writing Semantic Web applications. Different kinds of back-ends can be connected to Jena2 for transparently storing RDF-data persistently given Jena´s Java-API. Jena2 uses ARP2 as RDF parser.
It features (source: [20]):

- statement centric methods for manipulating an RDF model as a set of RDF triples
- resource centric methods for manipulating an RDF model as a set of resources with properties

[3]source: [17]

- cascading method calls for more convenient programming
- built in support for RDF containers - bag, alt and seq
- enhanced resources - the application can extend the behaviour of resources
- integrated parsers and writers for RDF/XML (ARP), N3 and N-TRIPLES
- support for typed literals

The basic Jena API did not have any query language interface. It was a basic store-retrieve object interface. But then the RDF based query language RDQL was proposed and a Jena2 API for RDQL has been defined. The implementation in Jena2 is coupled to relational database storage so that optimized query is performed over data held in an internal Jena relational persistent store.
The present project provides RDQL to AmosQL translation which enables Amos II to wrap RDF sources through ARP2, the RDF parser included in the Jena2 package, through RDQL.

2.6 Mediator / wrappers

An Amos II database system can integrate data of different type into its own object-oriented database using *wrappers*. The wrappers process data from different external sources to make them searchable with a query language, in our case AmosQL. For example, wrappers for ODBC based relational databases, CAD systems, Internet search engines, XML documents or MIDI files have been implemented[8]. This results in a common data model and a query language for heterogeneous data.

The wrappers functionality allows transparent queries to views over combinations of different kinds of back-end data sources, so called *mediators*. Amos II is such a distributed mediator system. Each mediator peer provides a number of transparent functional views of data reconciled from other mediator peers or wrapped data sources. Some wrapper functionality in Amos II is completely data source independent while other functionality is specific for a particular kind of data source. Amos II contains a hierarchy of wrappers to share wrapper functionality.
Amos II wraps RDF sources through ARP2, the RDF parser which is also included in the Jena2 package from HP-Labs. In [7] one can find a list of wrappers, that have been implemented in Amos II.

The mediator/wrapper approach is used for integrating heterogeneous data. Most mediator systems integrate data through a central mediator server accessing one or several data sources through a number of wrapper interfaces that translate data to a common data model (CDM) and know how to process queries to the source [4].
The present work is an embedding, not a wrapper from Amos II point of view.

It does not integrate data into Amos II´s own database like a wrapper would do. An embedding is an embedding of the Amos II system in other systems, while wrappers make other system searchable through Amos II.
This system makes it possible to access Amos II from RDQL. It will provide wrapping of SQL-sources and other sources Amos II can wrap.

3 The RDQL–Front System

This section describes the work and results of the project included the RDQL–front system.

3.1 Translation from RDQL to AmosQL

At first, I give some examples of RDQL queries and the appropriate AmosQL queries. All examples have been tested and verified. The RDQL queries are taken from HP RDQL tests in April 2003 [23].

RDQL:

```
SELECT ?x
FROM <vc-db-1.rdf>
WHERE (?x, <http://www.w3.org/2001/vcard-rdf/3.0#FN>, "John Smith")
```

AmosQL:

```
SELECT x
FROM charstring x
WHERE rdf_triples('vc-db-1.rdf') =
<x, 'http://www.w3.org/2001/vcard-rdf/3.0#FN', "John Smith">;
```

Figure 4: RDQL - AmosQL Example 1

To make it simple, the system allows namespace to be defined before running a query on Amos II and the resources in the FROM-clause are prefixed with the specified namespace. In In this case the namespace is: *http://swordfish.rdfweb.org/rdfquery/tests/old/tests/rdql-tests-2003-04-10/inputs/*. If the full namespace had been specified in the FROM-clause, the source would be *<http://swordfish.rdfweb.org/rdfquery/tests/old/tests/rdql-tests-2003-04-10/inputs/vc-db-1.rdf>* instead of only *<vc-db-1.rdf>*. The feature of the translator which allows to set the namespace is described in subsection 3.3.2.
In RDQL each variable is introduced by a question mark (?) and can be named using alphanumerical characters combined with an underscore (_). Multiple variables are separated by a space and/or an optional comma (,), see the following example:
The corresponding AmosQL has a similar SELECT-clause as RDQL just

```
SELECT ?name ?email, ?age,?tel_number
```

without the question mark in front of the variables. There is one special difference:
In case that all query variables are to be specified, one can use in RDQL the SQL-like shortcut in form of a star (*). In that case the parser has to scan the whole RDQL query for variables, first. Then all found variables are used in the AmosQL SELECT-clause.
The generation of the AmosQL FROM clause is a bit tricky to handle. All

variables included in the SELECT-clause *and* in the triple patterns have to be inserted here (in the FROM-clause of the generated AmosQL statement). As you can see in all examples, they are introduced by the word 'charstring'. In example 3 you can see that the whole query must be scanned for variables not only the SELECT-clause. Otherwise one would not find the ?y in the triple pattern and the translation would loose 'charstring y'. The RDF-source-file, in that case 'vc-db-1.rdf' is not used in the AmosQL FROM-clause but in the WHERE-clause. The WHERE-clause: In RDQL is a list of triple patterns

RDQL:

```
SELECT ?x, ?fname
FROM <vc-db-1.rdf>
WHERE (?x, <http://www.w3.org/2001/vcard-rdf/3.0#FN>, ?fname)
```

AmosQL:

```
SELECT x, fname
FROM charstring x, charstring fname
WHERE rdf_triples('vc-db-1.rdf') =
<x,'http://www.w3.org/2001/vcard-rdf/3.0#FN', fname>;
```

Figure 5: RDQL - AmosQL Example 2

indicated which have to be matched by each valid query result set. All patterns representing an RDF statement have the form *(subject, predicate, object)* where subject, predicate, and object can either be a *<URIref>* or a *?variable*.
Note, that the triple patterns in AmosQL are embedded in '<...>' but in RDQL they are embedded in '(...)' because '<...>' marks URIrefs in RDQL wherefore in AmosQL regular string delimiters ('...') are used. However, the main difference is the used foreign function *'rdf_triples'*, which is explained in detail in subsection 3.3.3. This function gets the RDF-source-file from the RDQL-FROM-clause as argument, and then its result is to set equal with the RDF triple patterns which were indicated in the RDQL query. If there is more than one triple pattern in the RDQL query, each triple pattern must have its own rdf_triple foreign function and be connected together with the keyword *'AND'*.
Example 3 shows this.
To make the query easier to read and write for humans, RDQL provides a way to shorten the length of URIs used in the FROM-, WHERE- and AND-clauses by defining a string prefix. Every prefix is defined in the USING-clause as demonstrated below:

```
WHERE (?resource, <info:age>, ?age)
USING info FOR <http://somewhere/peopleInfo#>
```

The last example (4) shows, how the USING-clause must be handled in the translation and how conditional expressions can be handled in the WHERE-clause. The matched variables in the query (here: info) and the following colon

RDQL:

```
SELECT ?givenName
FROM <vc-db-1.rdf>
WHERE (?y, <http://www.w3.org/2001/vcard-rdf/3.0#Family>, "Smith"),
(?y, <http://www.w3.org/2001/vcard-rdf/3.0#Given>, ?givenName)
```

AmosQL:

```
SELECT givenName
FROM charstring givenname, charstring y
WHERE rdf_triples('vc-db-1.rdf') =
<y, "http://www.w3.org/2001/vcard-rdf/3.0#Family", "Smith">
AND
rdf_triples('vc-db-1.rdf') =
<y, 'http://www.w3.org/2001/vcard-rdf/3.0#Given', givenName>;
```

Figure 6: RDQL - AmosQL Example 3

RDQL:

```
SELECT ?resource
FROM <vc-db-2.rdf>
WHERE (?resource, <info:age>, ?age)
AND ?age >= 24
USING info FOR <http://somewhere/peopleInfo#>
```

AmosQL:

```
SELECT resource
FROM charstring resource, charstring age
WHERE rdf_triples('vc-db-2.rdf')
= <resource, 'http://somewhere/peopleInfo#age', age>
AND age>='24';
```

Figure 7: RDQL - AmosQL Example 4

(:) are replaced by the prefix. The conditional expression (here: age>=24) is copied as is to AmosQL. Only the parameter (not variable or operator) (here: 24) must be embedded in "'...'", so that AmosQL deals it as charstring.
Finally, RDQL does not have a special character for ending queries but every AmosQL statement ends with a semicolon (;).

3.1.1 Translation rules

This is a brief summary of the steps to perform the translation:
For the *SELECT-clause* do the following:
If it is SELECT *

- insert the keyword 'SELECT'
- look for all variables in the whole query
- remove the question marks (?)
- insert all found variables in the SELECT-clause separated by commas

If it is SELECT ?Var (, ?Var)*

- insert the keyword 'SELECT'
- remove the question marks (?)
- insert the variables in the SELECT-clause separated by commas

For the *FROM-clause* do the following:

- insert the keyword 'FROM'
- save the RDF-source-filename
- insert each variable from the SELECT-clause, WHERE-clause and AND-clause introduced by 'charstring' separated by commas

For the *WHERE-clause* do the following:

- insert the keyword 'WHERE'
- replace <...> around URIrefs with '...'
- replace the brackets '(...)' around the triple patterns with '<...>'
- remove all question marks in front of the variables

Do the following 4 steps for each triple pattern

- insert "rdf_triples('"

- insert the RDF-source-filename from the FROM-clause and "')"
- insert equal ('=')
- insert the handled triple pattern
- if another triple pattern follows insert 'AND'

FOR the *AND-clause* do the following: If it has a triple pattern included:

- treat it like a WHERE-clause

Otherwise:

- insert the keyword 'AND'
- remove all question marks in front of the variables
- surround all parameters (not variables or operators) with '...'

FOR the *USING-clause* do the following:

- scan the whole query for the word in front of 'FOR'
- if the word followed by a colon is found, replace it with the prefix (the string after FOR)

And finally:

- insert a semicolon (';') at the end

3.1.2 Restrictions

There are a few RDQL statements, which can not be translated by this translator. It does not support to enter more than one source URL in the FROM-clause. I found two different grammars of RDQL on the web. One supports multiple source URLs [17] the other one does not [12]. Anyway, there was no example in the test cases (appendix B) to verify multiple sources, so therefore it is not implemented in this parser.
There were also no examples in the test cases to verify all possible operators in the AND-clause. All possible operators are listed in appendix A, and they are not altered by the translator. Only '=', '>' and '<' have been tested and approved, so there might be some operator which is not accepted in AmosQL.

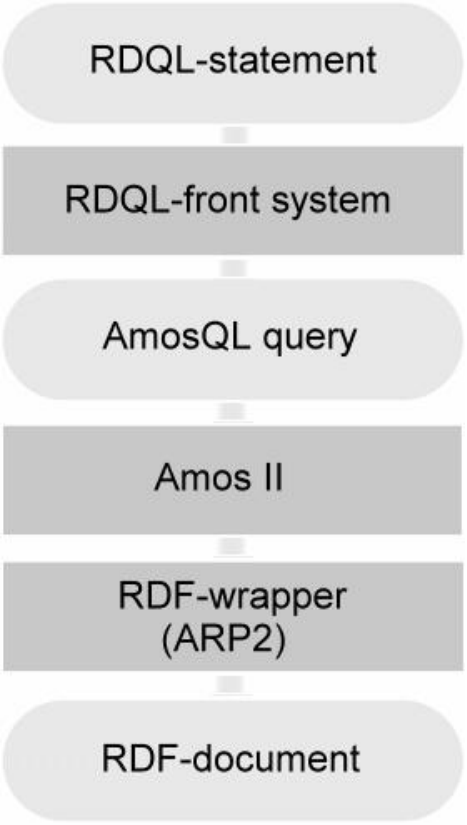

Figure 8: System layers around RDQL-front system

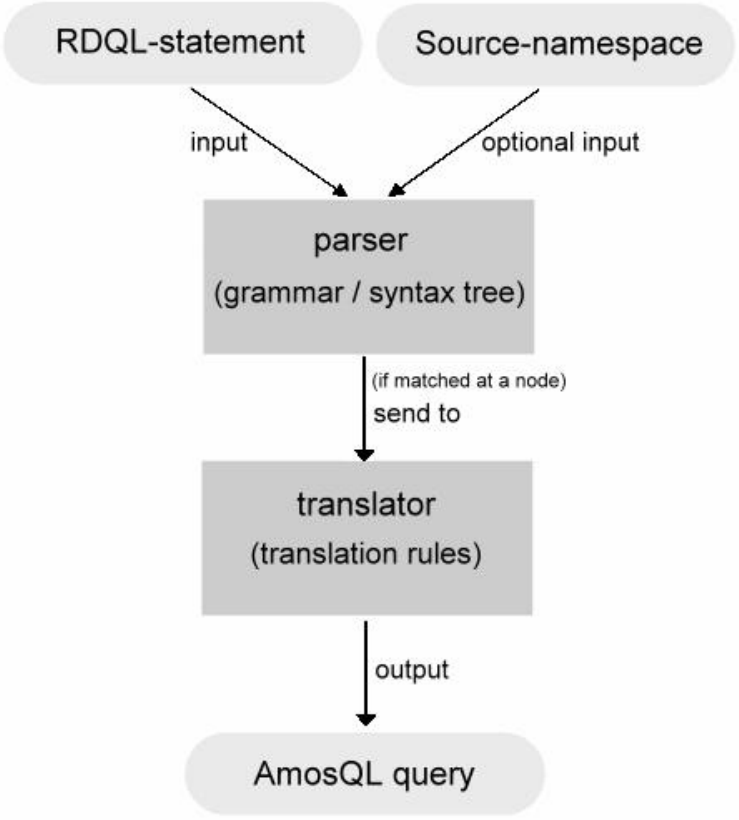

Figure 9: Modules and interfaces of the *RDQL-front system*

3.2 Architecture

Diagram 8 shows the different system layers and modules affected in this work. The RDQL-statement on top is given to the parser using a foreign function in Amos II, which calls the parser start method of the *RDQL-front system*. The parser recognizes the query and translates it into the corresponding AmosQL query, which is then executed in Amos II. Amos II uses its wrapper for RDF-sources (ARP2) to wrap the data in the RDF documents.
The present work was to implement the RDQL-front system. The architecture of that layer is demonstrated in figure 9.

It is optional to define a source-namespace (prefix) by a foreing Amos II function, which is then automatically inserted into the AmosQL output query. This feature is explained in subsection 3.3.2. After the parser gets an RDQL query from Amos II included as argument in a foreign function, it recognizes matches to the grammar, which is built up as a syntax tree. If a match is approved, that part of the query is sent to the translator, which handles it on the basis of its translation rules and outputs the translated query to Amos II. Finally the query is executed in Amos II.

3.3 Implementation

This section is an overview, how the translator is implemented and what features it has.

3.3.1 The translator

The parser itself is written by using JavaCC (Java Compiler Compiler), which is the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a program, in this case a Java program, that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides tree building via a tool called JJTree, which is also used in this project. JJTree is a preprocessor for JavaCC that inserts parse tree building actions at various places in the JavaCC source. The output of JJTree is run through JavaCC to create the parser.
In section 2.4 you can find the RDQL grammar included in the Jena package, which is exactly as is also used in this project. After recognizing a match the translator has to do the steps explained in 3.1.1.
A syntax tree is build from non-terminals in the grammar and finally it ends with terminals at its leaves. Every time the parsers syntax tree matches a terminal at one of his leaves, it has do some changes in the final AmosQL output query, which is a realized as an array called, *wholeQuery*. Sometimes it is necessary to know, which branches were taken in the syntax tree, what is easily realized by integer flag variables.

Figure 10 shows the code for the SELECT-clause, which is one of the easiest to handle but similar to other clauses. All terminals are surrounded by

```
void SelectClause() :
{}
{
LOOKAHEAD(2)
<SELECT> Var(0) (CommaOpt(0) Var(0))*
      {
wholeQuery[0]="select";
      }
|
<SELECT> "*"
      {
wholeQuery[0]="select";
      }
}
```

Figure 10: Handling of the SELECT-clause

"<...>" in the grammar (here for example <SELECT>). The integers (here (0)) shows the parser in the following node which node matched before. Either *<SELECT> Var() (CommaOpt() Var())** or *<SELECT> "*"* matches. Then the Java code beneath it will be processed, which inserts "select" into array wholeQuery in that case, which is the final AmosQL output query.
This array is built bit by bit during the parsing operation. There are several

Field	**Format of array wholeQuery**
field 0:	"select"
field 1:	variables for SELECT-clause separated by commas
field 2:	"from"
field 3:	all variables of the query introduced with "charstring" and separated by commas
field 4:	"where"
field 5:	"rdf_triples('" + source URL + "')"
field 6:	"="
field 7:	the triple pattern (for (#triple–patterns > 1) : <s,v,o> + field 5 + "=")
field 8:	constraint clause
field 9:	";"

vectors in which the parser writes something (imported strings from the original RDQL query or new defined strings) when something matches at an end node. The required modifications are handled then in these vectors.
Finally, when the array wholeQuery is built, the translator saves it into a string variable, called parsedQuery, which is also the return Amos II gets back for the foreign function 'rdql(*%RDQL-query%*', which is using the in [11] explained Java interfaces. This return AmosQL query is then executed in Amos II.

Vectors	
Vector selectVariables	handles variables after SELECT
Vector sources	(just one source) handles source URL
Vector rdfTripleVariables	handles variables in the triple patterns
Vector rdfTriples	handles triple patterns
Vector prefixvec	handles prefixes
Vector constraintvec	handles conditional expressions

3.3.2 Namespace

Setting a namespace to a certain prefix makes the query easier to read and write for humans.
Included in the created foreign functions for this project there is one function, called SourceNameSpace. This is a feature included in this system, which can be used under Amos II by typing: SourceNameSpace(*%URL prefix%*). It sets the namespace of the source URL to the entered one as long as it is not changed by the user or Amos II is quitted.
You can find the initializing code for the function in subsection 3.3.3.

3.3.3 Used foreign functions

Multi-directional foreign function mechanism gives transparent access from AmosQL to special purpose data structures such as internal Amos II metadata representations or user defined storage structures. The mechanism allows the programmer to implement query language operators in an external language (Java, C or Lisp)[4].
The code in this section contains the foreign functions which have to be loaded into Amos II before the RDQL–front system is disposed to work:

- *rdf_ triple_ cache*: holds the triple pattern in cache
- *rdf_ triples*: gets the source URL as argument and stores all triple patterns of the RDF source file in the cache if it wasn´t there before. The predefined foreign AmosQL function *parserdf* uses ARP2 to parse an RDF-file and emits the result as a stream.
- *rdql*: gets an RDQL query as argument and starts the translation and execute of the query
- *SourceNameSpace*: gets the prefix of the source as argument and holds it until a new prefix is inserted

The functions *rdf_ triples* and *rdf_ triple_ cache* implement a statement caching which is trivial to implement in Amos II as stored procedure. This effects correct behavior of the system. Otherwise, a purely streamed implementation of *rdf_ triples* in terms of *parserdf* without statement caching would cause reentrant calls to ARP2 which makes the system crash or give unexpected results.

rdf_ triple_ cache:

```
create function rdf_triple_cache(Charstring src) -> Bag of
<Charstring, Charstring, Charstring> as stored;
```

rdf_ triples:

```
create function rdf_triples(Charstring src) ->
<Charstring s, Charstring p, Charstring o> as
begin
    if notany(rdf_triple_cache(src)) then
        for each Charstring s, Charstring p, Charstring o, Integer i
            where <s,p,o,i> = parserdf(src)
            add rdf_triple_cache(src) = <s,p,o>;
        for each Charstring s, Charstring p, Charstring o
            where <s,p,o> = rdf_triple_cache(src)
            result <s,p,o>;
end;
```

rdql:

```
create function rdql(charstring query) -> object as foreign "JAVA:Start/rdql";
```

SourceNameSpace:

```
create function SourceNameSpace(Charstring nm) -> Charstring as foreign
"JAVA:Start/sourcenamespace";
```

3.4 Evaluation

To evaluate the translator, several queries have been tested and verified. There is a standard benchmark source for RDQL queries from HP used for implementing RDQL into their Jena package [23]. It includes source input files, queries, and the documented results. Only the queries having an RDF-file as a source are used in this evaluation.
Every query is translated by the parser and all results are correct.
You can find the queries, translations, and results in appendix B.

When we started to run the tests, we realized that an RDQL query with more than one triple pattern in its WHERE-clause did not run. We looked at the problem and it turns out that the ARP2 parser we are using is not designed for being recursively callable (i.e. it is not reentrant). A purely streamed implementation of *rdf_ triples* in terms of *parserdf* without statement caching causes reentrant calls to ARP2, which makes it crash or give unexpected results. Thus for correct behavior of the system, statement caching is necessary.

4 Summary and future work

The now created RDQL–front system allows queries expressed in RDQL to be processed over RDF-data provided through Amos II. This is implemented as an embedding to Amos II which recognizes, translates, and executes RDQL queries.

One of the most important things to get the RDQL–front system run, was the implementation of the statement caching 3.4. The trivial way, to implement it as a stored procedure with only a few lines of code (cp. 3.3.3), was a great solution. Only the AmosQL feature – to allow a foreign function with side effects – made this possible. The side effect is in this case, that it caches a read RDF-file the first time it is accessed.
Future work will be to make the parser reentrant and evaluate performance trade-offs with statement caching.

Finally, the possible uses of this project are reconsidered:
The system SWARD (Semantic Web Abridged Relational Databases) [1] is being developed for scalable RDF based wrapping of existing relational databases in the hidden web. It uses the query language QEL, which is a Datalog based query language for RDF.
Now, RDQL could replace this query language to allow RDQL-based mediation embedded in Jena2.
Especially, if RDQL or a similar language becomes the standard query language for RDF in the semantic web community, this project together with other Amos II functionality enables access from semantic web to data sources wrapped by Amos II.

References

[1] J. Petrini, T. Risch: Processing Queries over RDF Views of Wrapped Relational Databases, *1st International Workshop on Wrapper Techniques for Legacy Systems, WRAP 2004,* Delft, Holland, November 2004. http://user.it.uu.se/ udbl/publ/WRAP04.pdf

[2] T. Berners-Lee, J. Hendler and O. Lassila: The Semantic Web, *Scientific American, May 2001.*

[3] S. Decker et al: The Semantic Web - on the Roles of XML and RDF, *IEEE Internet Computing, Sept./Oct. 2000.*

[4] T. Risch, V. Josifovski, t. Katchaounov: Functional Data Integration in a Distributed Mediator System *in P. Gray, L.Kerschberg, P.King, and A.Poulovassils (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, Springer, ISBN 3-540-00375-4, 2003.* http://user.it.uu.se/%7Etorer/publ/FuncMedPaper.pdf

[5] S. Flodin, M. Hansson, V. Josifovski, T. Katchaounov, T. Risch, and M. Sköld: Amos II User's Manual, *http://user.it.uu.se/ udbl/amos/doc/amos_ users_ guide.html*

[6] A. Seaborne, HP Labs Bristol: RDQL - A Query Language for RDF, *W3C Member Submission 9 January 2004* http://www.w3.org/Submission/RDQL/

[7] K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds Efficient RDF Storage and Retrieval in Jena2, *Workshop on Semantic Web and Databases, 7 September 2003, Berlin, Germany* http://www.hpl.hp.com/techreports/2003/HPL-2003-266.pdf

[8] Uppsala Database Laboratory Amos II Wrappers, *http://user.it.uu.se/%7Eudbl/amos/wrappers.html*

[9] D.Brickley, R.V.Guha: RDF Vocabulary Description Language 1.0: RDF-Schema, *W3C Recommendation 10 February 2004* http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

[10] G.Klyne, J.J.Carroll: Resource Description Framework (RDF): Concepts and Abstract Syntax, *W3C Recommendation 10 February 2004* http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[11] T. Risch, D. Elin: Amos II Java Interfaces, *Uppsala Database Laboratory, 2000)* http://user.it.uu.se/ torer/publ/javaapi.pdf

[12] The jena semantic web toolkit, RDQL grammar, *http://www.hpl.hp.com/semweb/rdql-grammar.html*

[13] J. Hendler, T. Berners-Lee, E. Miller: Integrating Applications on the Semantic Web, *Journal of the Institute of Electrical Engineers of Japan, Vol 122(10)*, October, 2002, p. 676-680. http://www.w3.org/2002/07/swint

[14] M. Koivunen, E. Miller: W3C Semantic Web Activity, *Semantic Web Kick-off Seminar, Finland Nov 2, 2001* http://www.w3.org/2001/12/semweb-fin/w3csw

[15] D. Brickley, R.V. Guha: RDF Vocabulary Description Language 1.0: RDF Schema, *W3C Recommendation 10 February 2004* http://www.w3.org/TR/rdf-schema/

[16] F. Manola, E. Miller: RDF Primer, *W3C Recommendation 10 February 2004* http://www.w3.org/TR/2004/REC-rdf-primer-20040210/#dublincore

[17] A. Seaborne: RDQL - A Query Language for RDF, *W3C Member Submission 9 January 2004* http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/

[18] T. Risch: Functional Queries to Wrapped Educational Semantic Web Meta-Data, *in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, Springer, ISBN 3-540-00375-4, 2003.* http://user.it.uu.se/ torer/publ/semfdm.pdf

[19] RDQL Tutorial, *http://phpxmlclasses.sourceforge.net/rdql.html*

[20] Jena 2 - A Semantic Web Framework, *http://www.hpl.hp.com/semweb/jena2.htm, HP*

[21] H. Garcia-Molina, J. Ullman, J. Widom: Database Systems: The Complete Book (International Edition), *Prentice-Hall, 2003.*

[22] R. Elmasri, S. Navathe Fundamentals of Database Systems, *4th ed. 2004. XXVI, Addison-Wesley Longman, Amsterdam, ISBN 0321204484*

[23] RDQL-tests-2003-04-10, *http://swordfish.rdfweb.org/rdfquery/tests/old/tests/rdql-tests-2003-04-10/*

[24] BNF for rdql.jj, *http://www.hpl.hp.com/semweb/rdql-grammar.html*

A Appendix: RDQL grammar

This is a permissive grammar. It is designed for convenience and includes liberal interpretations of terms from other systems.

Lexical Tokens

QuotedURI	::=	'<' URI characters (from RFC 2396) '>'
NSPrefix	::=	NCName As defined in XML Namespace
LocalPart	::=	NCName As defined in XML Namespace
SELECT	::=	'SELECT' Case Insensitive match
FROM	::=	'FROM' Case Insensitive match
SOURCE	::=	'SOURCE' Case Insensitive match
WHERE	::=	'WHERE' Case Insensitive match
AND	::=	'AND' Case Insensitive match
USING	::=	'USING' Case Insensitive match
Identifier	::=	([a-z][A-Z][0-9]-_.])+
EOF	::=	End of file
COMMA	::=	','
INTEGER_LITERAL	::=	([0-9])+
FLOATING_POINT_LITERAL	::=	([0-9])*.'([0-9])+('e'('+'|'-')?([0-9])+)?
STRING_LITERAL1	::=	'"'UTF-8 characters'"' (with escaped ¨)
STRING_LITERAL2	::=	"'"UTF-8 characters"'" (with escaped ´)
LPAREN	::=	'('
RPAREN	::=	')'
COMMA	::=	','
DOT	::=	'.'
GT	::=	'>'
LT	::=	'<'
BANG	::=	'!'
TILDE	::=	' '
HOOK	::=	'?'
COLON	::=	':'
EQ	::=	'=='
NEQ	::=	'!='
LE	::=	'<='
GE	::=	'>='
SC_OR	::=	'||'
SC_AND	::=	'&&'
STR_EQ	::=	'EQ' Case Insensitive match
STR_NE	::=	'NE' Case Insensitive match
PLUS	::=	'+'
MINUS	::=	'-'
STAR	::=	'*'
SLASH	::=	'/'
REM	::=	'%'
STR_MATCH	::=	'= ' | ' '
STR_NMATCH	::=	'! '
DATATYPE	::=	'g
AT	::=	'@'

Table 1: RDQL grammar part 1

References to lexical tokens are enclosed in <>. Whitespace is skipped.

Part 2

CompilationUnit	::=	Query <EOF>
CommaOpt	::=	(<COMMA>)?
Query	::=	SelectClause (SourceClause)? TriplePatternClause (ConstraintClause)? (PrefixesClause)?
SelectClause	::=	(<SELECT> Var (CommaOpt Var)* | <SELECT> <STAR>)
SourceClause	::=	(<SOURCE> | <FROM>) SourceSelector
SourceSelector	::=	QName
TriplePatternClause	::=	<WHERE> TriplePattern (CommaOpt TriplePattern)*
ConstraintClause	::=	<SUCHTHAT> Expression ((<COMMA> | <SUCHTHAT>) Expression)*
TriplePattern	::=	<LPAREN> VarOrURI CommaOpt VarOrURI CommaOpt VarOrConst <RPAREN>
VarOrURI	::=	Var | URI
VarOrConst	::=	Var | Const
Var	::=	"?" Identifier
PrefixesClause	::=	<PREFIXES> PrefixDecl (CommaOpt PrefixDecl)*
PrefixDecl	::=	Identifier <FOR> <QuotedURI>
Expression	::=	ConditionalOrExpression
ConditionalOrExpression	::=	ConditionalAndExpression (<SC_OR> ConditionalAndExpression)*
ConditionalAndExpression	::=	StringEqualityExpression (<SC_AND> StringEqualityExpression)*
StringEqualityExpression	::=	ArithmeticCondition (<STR_EQ> ArithmeticCondition | <STR_NE> ArithmeticCondition | <STR_MATCH> PatternLiteral | <STR_NMATCH> PatternLiteral)*
ArithmeticCondition	::=	EqualityExpression
EqualityExpression	::=	RelationalExpression (<EQ> RelationalExpression | <NEQ> RelationalExpression)?
RelationalExpression	::=	AdditiveExpression (<LT> AdditiveExpression | <GT> AdditiveExpression | <LE> AdditiveExpression | <GE> AdditiveExpression)?
AdditiveExpression	::=	MultiplicativeExpression (<PLUS> MultiplicativeExpression | <MINUS> MultiplicativeExpression)*
MultiplicativeExpression	::=	UnaryExpression (<STAR> UnaryExpression | <SLASH> UnaryExpression | <REM> UnaryExpression)*

UnaryExpression	::=	UnaryExpressionNotPlusMinus | (<PLUS> UnaryExpression | <MINUS> UnaryExpression)
UnaryExpressionNotPlusMinus	::=	(<TILDE> | <BANG>) UnaryExpression | PrimaryExpression
PrimaryExpression	::=	Var | Const | <LPAREN> Expression <RPAREN>
Const	::=	URI | NumericLiteral | TextLiteral | BooleanLiteral | NullLiteral
NumericLiteral	::=	(<INTEGER_LITERAL> | <FLOATING_POINT_LITERAL>)
TextLiteral	::=	(<STRING_LITERAL1> | <STRING_LITERAL2>) (<AT> Identifier)? (<DATATYPE> URI)?
PatternLiteral	::=	..
BooleanLiteral	::=	<BOOLEAN_LITERAL>
NullLiteral	::=	<NULL_LITERAL>
URI	::=	<QuotedURI> | QName
QName	::=	<NSPrefix> ':' (<LocalPart>)? Unlilke XML Namespaces, the local part is optional
Identifier	::=	(<IDENTIFIER> | <SELECT> | <SOURCE> | <FROM> | <WHERE> | <PREFIXES> | <FOR> | <STR_EQ> | <STR_NE>)

Table 2: RDQL grammar part 2 in BNF

Note: The term "Literal" refers to a constant value, and not only an RDF literal.

B Appendix: Test queries, translations, and results

Schema:
RDQL-query

AmosQL-query
Result

Test query 1: Simple statement
```
SELECT ?x
FROM <vc-db-1.rdf>
WHERE (?x, <http://www.w3.org/2001/vcard-rdf/3.0#FN>, "John Smith")
```

```
SELECT x
FROM charstring x
WHERE rdf_triples('vc-db-1.rdf') =
<x, 'http://www.w3.org/2001/vcard-rdf/3.0#FN', "John Smith">;
```
Result 1:
"http://somewhere/JohnSmith/"

Test query 2: Select two variables
```
SELECT ?x, ?fname
FROM <vc-db-1.rdf>
WHERE (?x, <http://www.w3.org/2001/vcard-rdf/3.0#FN>, ?fname)
```

```
SELECT x, fname
FROM charstring x, charstring fname
WHERE rdf_triples('vc-db-1.rdf') =
<x,'http://www.w3.org/2001/vcard-rdf/3.0#FN', fname>;
```
Result 2:
<"http://somewhere/MattJones/","Matt Jones">
<"http://somewhere/SarahJones/","Sarah Jones">
<"http://somewhere/RebeccaSmith/","Becky Smith">
<"http://somewhere/JohnSmith/","John Smith">

Test query 3: Multi triple patterns

```
SELECT ?givenName
FROM <vc-db-1.rdf>
WHERE (?y, <http://www.w3.org/2001/vcard-rdf/3.0#Family>, "Smith") ,
(?y, <http://www.w3.org/2001/vcard-rdf/3.0#Given>, ?givenName)
```

```
SELECT givenName
FROM charstring givenname, charstring y
WHERE rdf_triples('vc-db-1.rdf') =
<y, "http://www.w3.org/2001/vcard-rdf/3.0#Family", "Smith">
AND rdf_triples('vc-db-1.rdf') =
<y, 'http://www.w3.org/2001/vcard-rdf/3.0#Given', givenName>;
```

Result 3:

"Rebecca"
"John"

Test query 4: Multi variables, multi triple patterns

```
SELECT ?resource, ?givenName
FROM <vc-db-1.rdf>
WHERE (?resource, <http://www.w3.org/2001/vcard-rdf/3.0#N>, ?z) ,
(?z, <http://www.w3.org/2001/vcard-rdf/3.0#Given>, ?givenName)
```

```
SELECT resource, givenName
FROM charstring resource, charstring givenName, charstring z
WHERE rdf_triples('vc-db-1.rdf') =
<resource, 'http://www.w3.org/2001/vcard-rdf/3.0#N', z>
AND rdf_triples('vc-db-1.rdf') =
<z, 'http://www.w3.org/2001/vcard-rdf/3.0#Given', givenName>;
```

Result 4:

<"http://somewhere/MattJones/","Matthew">
<"http://somewhere/SarahJones/","Sarah">
<"http://somewhere/RebeccaSmith/","Rebecca">
<"http://somewhere/JohnSmith/","John">

Test query 5: AND-clause with conditional expression and USING-clause, different source

```
SELECT ?resource
FROM <vc-db-2.rdf>
WHERE (?resource, <info:age>, ?age)
AND ?age >= 24
USING info FOR <http://somewhere/peopleInfo#>
```

```
SELECT resource
FROM charstring resource, charstring age
WHERE rdf_triples('vc-db-2.rdf') =
<resource, 'http://somewhere/peopleInfo#age', age>
AND age>='24';
```

Result 5:
"http://somewhere/JohnSmith/"

Test query 6: Multi prefixes

```
SELECT ?resource, ?familyName
FROM <vc-db-2.rdf>
WHERE (?resource, <info:age>, ?age) ,
(?resource, <vCard:N>, ?y) , (?y, <vCard:Family>, ?familyName)
AND ?age >= 24
USING info FOR <http://somewhere/peopleInfo#> ,
vCard FOR <http://www.w3.org/2001/vcard-rdf/3.0#>
```

```
SELECT resource, familyName
FROM charstring resource, charstring familyName, charstring age,
charstring y
WHERE rdf_triples('vc-db-2.rdf') =
<resource, "http://somewhere/peopleInfo#age", age>
AND rdf_triples('vc-db-2.rdf') =
<resource, 'http://www.w3.org/2001/vcard-rdf/3.0#N', y>
AND rdf_triples('vc-db-2.rdf') =
<y, 'http://www.w3.org/2001/vcard-rdf/3.0#Family', familyName>
AND age >= '24';
```

Result 6:
<"http://somewhere/JohnSmith/","Smith">

Test query 7: Select the predicate

```
SELECT ?prop
FROM <vc-db-2.rdf>
WHERE (<http://somewhere/JohnSmith/> , ?prop, "John Smith")
USING vCard FOR <http://www.w3.org/2001/vcard-rdf/3.0#>
```

```
SELECT prop
FROM charstring prop
WHERE rdf_triples('vc-db-2.rdf') =
<'http://somewhere/JohnSmith/', prop, "John Smith">;
```

Result 7:

"http://www.w3.org/2001/vcard-rdf/3.0#FN"

Test query 8:

```
SELECT ?b
FROM <vc-db-2.rdf>
WHERE (<http://somewhere/JohnSmith/> , <vCard:N>, ?b)
USING vCard FOR <http://www.w3.org/2001/vcard-rdf/3.0#>
```

```
SELECT b
FROM charstring b
WHERE rdf_triples('vc-db-2.rdf') =
<'http://somewhere/JohnSmith/','http://www.w3.org/2001/vcard-
rdf/3.0#N',b>;
```

Result 8:

"_:jARP1 "

Test query 9: Different source

```
SELECT ?property
FROM <vc-db-3.rdf>
WHERE (?person, <vCard:FN>, "John Smith") ,
(?person, ?property, ?obj)
USING
vCard FOR <http://www.w3.org/2001/vcard-rdf/3.0#> ,
rdf FOR <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
SELECT property
FROM charstring property, charstring person, charstring obj
WHERE rdf_triples('vc-db-3.rdf') =
<person, 'http://www.w3.org/2001/vcard-rdf/3.0#FN', "John Smith">
AND rdf_triples('vc-db-3.rdf') =
<person, property, obj>;
```

Result 9:

"http://www.w3.org/2001/vcard-rdf/3.0#TEL"
"http://www.w3.org/2001/vcard-rdf/3.0#TEL"
"http://www.w3.org/2001/vcard-rdf/3.0#N"
"http://somewhere/peopleInfo#age"
"http://www.w3.org/2001/vcard-rdf/3.0#FN"

Test query 10:

```
SELECT ?telephoneNumber
FROM <vc-db-3.rdf>
WHERE (?person, <vCard:FN>, "John Smith") ,
(?person, <vCard:TEL>, ?tel) ,
(?tel, <rdf:type>, <vCard:work>) ,
(?tel, <rdf:value>, ?telephoneNumber)
USING
vCard FOR <http://www.w3.org/2001/vcard-rdf/3.0#> ,
rdf FOR <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
SELECT telephoneNumber
FROM charstring telephoneNumber, charstring person, charstring tel
WHERE rdf_triples('vc-db-3.rdf') =
<person, 'http://www.w3.org/2001/vcard-rdf/3.0#FN', "John Smith">
AND rdf_triples('vc-db-3.rdf') =
<person, 'http://www.w3.org/2001/vcard-rdf/3.0#TEL', tel>
AND rdf_triples('vc-db-3.rdf') =
<tel, 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
'http://www.w3.org/2001/vcard-rdf/3.0#work'>
AND rdf_triples('vc-db-3.rdf') =
<tel, 'http://www.w3.org/1999/02/22-rdf-syntax-ns#value',
telephoneNumber>;
```

Result 10:

"+44 117 555 5555"