

# Optimierung komplexer Anfragen in Peer-to-Peer Netzen

## Optimization of Complex Queries in Peer-to-Peer Networks

Bachelor Thesis

im Studiengang Angewandte Informatik  
an der Universität Hannover von

David Kuhn

Erstprüfer Prof. Dr. W. Nejdl,  
Institut für Informationssysteme  
Fachgebiet Wissensbasierte Systeme

Zweitprüfer PD Dr. F. Steimann,  
Institut für Informationssysteme  
Fachgebiet Wissensbasierte Systeme

Hiermit erkläre ich, die vorliegende Bachelorarbeit „Optimierung komplexer Anfragen in Peer-to-Peer Netzen“ selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Hannover, den 11. September 2003

## Abstract

This work is part of the efforts of the Edutella project which has the aim to create a peer-to-peer network for storing, querying, exchanging and processing any kinds of RDF metadata. The Edutella project itself is a sub module of the WGLN PADLR project (Personalized Access to Distributed Learning Repositories) with participating research groups from universities in Sweden, Germany and the U.S.

This document focuses on the development of a "wrapper" for the Rssdb data base. This wrapper provides the features for being part of an Edutella client, for translating Edutella queries into the local language of Rssdb, for connecting to the Rssdb client and transmitting the query and for getting the results from Rssdb, translating them for Edutella and handing them back to Edutella. The creation of this new wrapper has the intention to increase the compatibility of Edutella to local data bases. This shall enable more people to use Edutella since it is the more the beneficial the more people are using it and the more files are shared.

# **„Optimization of Complex Queries in Peer-to-Peer Networks“ - Contents**

1. Edutella and Wrapper
  2. QEL
  3. RDF
  4. Rssdb and RQL
  5. EQM
  6. Mapping (QEL => RQL)
    - A. Mapping Structure
    - B. BuiltInLiterals
    - C. Negation and BuiltInPredicates
    - D. Namespaces
    - E. OuterJoin Literals
  7. Decision Tree
  8. Cross Product
  9. Unification
  10. How to become an OuterJoin Literal
  11. Combination: OuterJoins, Rules and Namespaces
  12. RQL – Bug
  13. Results
  14. Future Work and Conclusion
- Appendix Translation Examples
- Appendix Rssdb Configuration
- References

## 1. Edutella and Wrapper

Every time a lecturer wants to create a lecture, a speech or material for the course or a student is looking for material concerning a very specific topic for learning purposes there are unlimited resources worldwide even for the most unusual topic but it is out of reach.

For this reason Edutella provides a computer network which makes it possible for everyone to share his knowledge and his created material and to get the document which other people are willing to share.

As the authors want to keep control over their work and the possibility of deleting, updating or replacing it, there is no controlling centre but a network of equal peers. It is a peer-to-peer network in the fashion of Napster or Kazaa but the difference is that it is easy to query for a music file but it is difficult to retrieve a file concerning a very special topic. Thus, it is required that the query language is much more complex and allows a lot of features for specialising the query.

Edutella is a distributed data base system without a central instance. This data base consists of a lot of local data bases which have very different hardware and software and of course they may have a local query language which is different from the query language of the network. The language of the network is called Query Exchange Language (QEL) and a peer receives a query always in this RDF based language. The peer forwards the query to a Translator program which takes also care of forwarding the translated query to the actual data base. This Translator is called Wrapper. When the data base has obtained the results it hands them back to the Wrapper formulated in the local data base language and the Wrapper translates it back into QEL. The peer sends the response then to the questioner.

This work is part of the Edutella project. It is the development of one of these wrappers and it has the aim to increase the compatibility of Edutella to local data bases. This shall enable more people to use Edutella since it is the more the beneficial the more people are using it and the more files are shared.

This work creates the data base connection for Edutella to the Rssdb data base which is RDF based and has a local query language called RQL.

## 2 . QEL

The source language the wrapper has to translate from is the Query Exchange Language (QEL). It is used by the Edutella peers to communicate with each other and has to be translated by the wrappers into the local language of the particular data base.

It provides different features and the wrappers have to announce which of these features they support. They will then receive only the queries they are able to translate.

QEL is based on Datalog which is a language for modelling knowledge in a way a human can probably think of it. Datalog provides facts and rules for expressing a knowledge base (a set of information).

Like Datalog, QEL provides rules and facts. The facts are called literals in QEL and they are small sentences each explaining a small piece of information. They may contain variables which means then that they ask for a piece of information which makes this sentence correct. This piece of information can be found by looking for appropriate literals stored in the data base and by applying rules to these stored literals.

A literal containing a variable is called queryliteral. Which argument represents a value and which one is a variable has to be defined in the QEL query. All variables are listed at the beginning. All not listed arguments are no variables.

But it does not make much sense to use literals without variables in a QEL query. Thus, normally I say literal also when I talk about queryliterals. Although it is not very useful to use literals without variables in the query, the wrapper is able to translate literals without variables correctly.

Rules are used for expressing reasoning and for connecting facts together. A Rule has always a head which shows the conclusion and a body which contains the constraints. If the constraints are fulfilled you can use the conclusion. Example rule:

```
AIBook(X) :- Book(X), Title(X, "%Artificial Intelligence%")
```

It means that X is an aibook if X is a book and its title contains the words "Artificial Intelligence". The wildcard in QEL is the "%".

A rule is conjunctive when it is the only one with this head. It allows then only the logical "and" in its body for the reasoning. A disjunctive rule means that there is more than one rule with the same head. This expresses the logical "or" since the body of only one of the rules has to be fulfilled to make the head a true fact. Two heads are equal if they have

the same name and the same amount of arguments. It is not necessary that the variables have the same names.

A literal may be nonouterjoin or outerjoin. Nonouterjoin literals mean that this piece of information is a necessary part of the result.

An outerjoin literal is only an optional part of the query and asks for additional information. If a piece of information does not have this optional part it can nonetheless be a result. In the result table the optional field of this result will be filled with null. But if a piece of information does not provide a nonouterjoin literal it can not be considered as a result for the query.

Example:

`AIBook(X), Author*(X,S)`

Which asks for all AIBooks according to the rule from above and the author if available. The outerjoin literals of the QEL-datalog examples herein are marked with a "\*" behind the predicates.

Set of data:

1. `Title(_x,"Artificial Intelligence for Dummies"),Book(_x)`
2. `Title(_y,"Artificial Intelligence for Experts"),Book(_y),Author(_y,"Shodan")`
3. `Title(_z,"Artificial Intelligence for Professionals"),Author(_z,"Edward Diego")`

The books `_x` and `_y` are in the result set. `_z` does not belong to the result set because of the missing type information. For `_y` is also the title mentioned in the result set.

The recursion means that a rule may contain its own head in the body. Linear recursion is a limited recursion and general recursion allows all types of recursion. But there has always to be also a rule with the same head like the recursive rule but without being recursive for making it possible to end the recursion.

The BuiltInPredicates are predefined predicates in Edutella. There are eight predicates available each with two arguments:

`Equals`, `like`, `language`, `datatype`, `nodetype`, `greaterThan`, `lessThan`, `member`.

`Like` allows the wildcard "%" and `equals` does not allow it. `Language` and `datatype` ask for this XML information of an object. `Member` asks if an object is inside a RDF container like sequence or bag. `Nodetype` asks if an object is a resource, literal, container or a (resource or container).

The features a wrapper may support or may not support are the negation of predicates, conjunctive rules, disjunctive rules, outerjoin literals, linear recursion, general recursion and the BuiltInPredicates of Edutella. The negation is only allowed for BuiltInPredicates.

### 3 . RDF

The Resource Description Framework (RDF) is a language for expressing information. This language is designed in a way that computers can easily read them and work with them. This means that it has tokens for marking the beginning and the end of every piece of information. The tokens make the language easy to parse (read) for the programs.

RDF is intended for representing information for programs. This information is often metadata, information about other pieces of data. For example it could be used to mention the author and the creating date of a document in the internet or for representing data about a video file like length and required codes.

The author of RDF is the World Wide Web Consortium (W3C).

The core piece of information in RDF is a triple consisting of a subject, a predicate and an object. The subject has to be a resource located in the web or on the localhost. The predicate is a property of this resource and the object is the value of this property. It can also be a resource or a literal.

These simple sentences may be combined to more complex structures.

The subject may have more than one property and the object may have properties too if it is a resource. The complex structures may be expressed by a sequence of such triples or by a RDF graph.

The subjects and objects are represented by nodes and the predicates are represented by edges.

Also empty nodes are possible which are only bags for other properties.

The empty nodes have no names and are called anonymous resources.

For example there could be an anonymous resource like an address which contains a street property, a city property and a postal code property but it does not need a name itself.

Properties and other things which have a definition have a namespace.

This is the path where the definition is stored. This can be in the web or on the localhost (with file:/<path>). Several things may have the same namespace which ends then on a "#". The namespace plus their name is the full path to their definition.

## 4. Rssdb / RQL

Rssdb is a data base for storing and querying RDF data. It is based on the PostgreSQL v7.3.3 data base server which language is the relational SQL (Structured Query Language). The Rssdb (RDF Schema Specific DataBase) client (Rssdb Loader v2.0) stores RDF files into this SQL database. It has a very exact data validation program which checks at first if the data is valid before storing it.

For querying the data there is the rql interpreter v2.0. It uses the RDF Query Language (RQL) which is a descriptive query language. That means that it describes how the data should look like instead of saying where to look for the data. The RDF data looks like a huge RDF graph. The query is actually a small RDF graph which is matched against the big one.

Wherever the small graph fits to a part of the big graph is data stored which will be included into the result set.

This is a descriptive approach since the query says only which characteristics the data has to fulfil but it explains the data base, not in which relation (table) and in which column or row the data is stored. For example:

**Descriptive:**

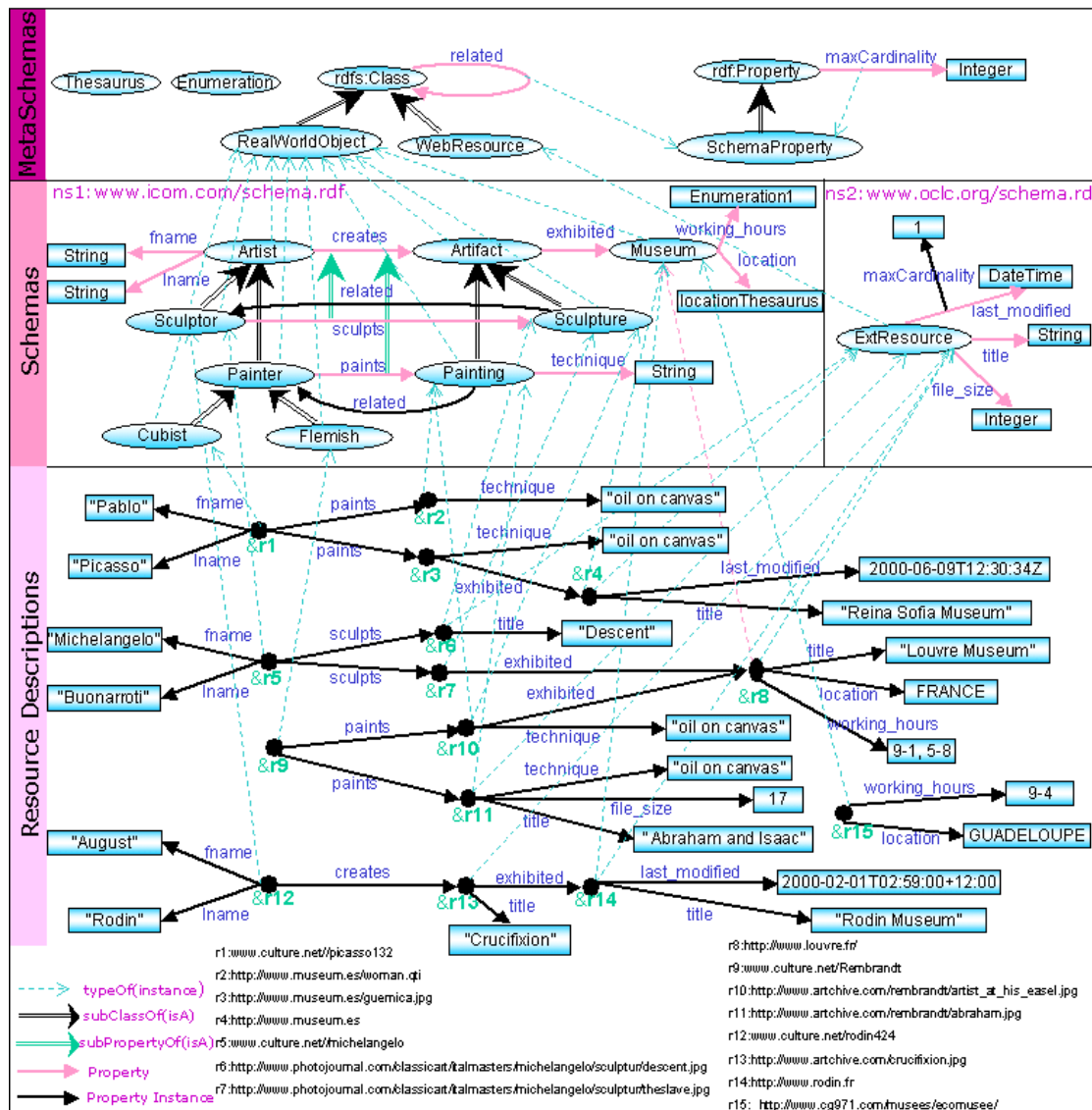
Print all authors who have written a book which contains `"*cycling*"` in its title and which was published before the year 2000.

**Procedural:**

Print all data stored in the column `"author"` of the table `"Book"` which contain `"*cycling*"` in the column `"title"` and a number smaller than 2000 in the column `"year"`.

The Rssdb data base uses a schema hierarchy for storing data. At First a metaschema has to be stored which defines the basic structure of the data like classes and properties. The next thing which can be loaded into the data base are schemas which define specific classes and their properties. These schemas can use other schemas which then have to be loaded into the data base first. For example a first schema may define properties like `title` and `file_size` which allow a resource to have a string value as its `title` and an integer value as its `file_size`. The data is stored after the schemas. It contain real pieces of information e. g. `Michelangelo made a sculpture which is called "Descent"`.

The hierarchy including the data could look like:



There are a lot of possibilities in RQL for querying this data base. The most important one for this wrapper is the querying of the Resource Descriptions since QEL asks for this kind of information. A query for this has a structure like:

```
Select X, Y
From {X} paints {Y}, {Y} technique {Z}
Where Z = "oil on canvas"
```

The query enquires for all paintings which are painted with oil and its painters. The fromclause describes a small RDF graph by mentioning some predicates. The two {Y} are the same variable since the variable names are unique in the query. Thus, the paints predicate and the technique predicate are connected. The whereclause gives a constraint for this small RDF graph because it mentions that the technique has to be "oil on canvas". And the selectclause finally mentions which information of the

small RDF graph have to be in the result set. The chapter “Mapping” contains more about this kind of queries.

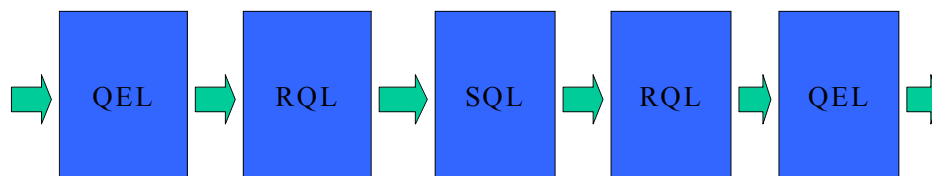
From the Edutella features mentioned above are the following features provided by RQL:

This wrapper supports equals, like, greaterThan, lessThan, the negation of this predicates, conjunctive rules, disjunctive rules and outerjoin literals inside or outside of the rules. Recursion does not exist in RQL and it is therefore difficult to implement. Outerjoin literals and rules does also not exist in RQL but it is possible to translate them into logically equivalent queries. The three missing BuiltInPredicates are not supported by RQL until now.

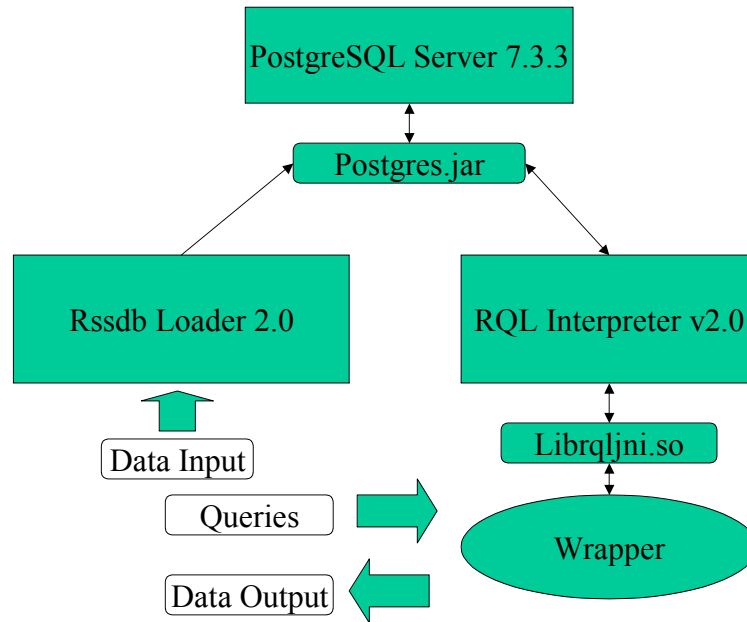
The RQL wrapper of Edutella connects to the RQL interpreter and sends it a query as a string and a file name including a path for the results. The interpreter translate the RQL query into SQL and queries the PostgreSQL server. The result are stored by the interpreter into the file for the results. If this file does not exist it will be created by the RQL interpreter otherwise the existing file will be overwritten. The wrapper reads the file and transforms it back into QEL.

We do not use the SQL wrapper for querying the data from PostgreSQL directly since we want to have the advantages of using RDF data instead of relational data. The RDF data can be queried descriptively and contains semantics. And the RQL interpreter does already a lot of work with translating the relations in the data base into RDF graphs.

Another problem of querying PostgreSQL directly would be that the internal data representation of Rssdb will maybe be changed in the next versions. But the RQL language will remain and will only be extended. Thus, the way of translation is:



For the programs are also two libraries required. The PostgreSQL JDBC driver (postgres.jar) is needed by the Rssdb Loader. And the librqjni.so library is needed by the wrapper for using the rql interpreter.

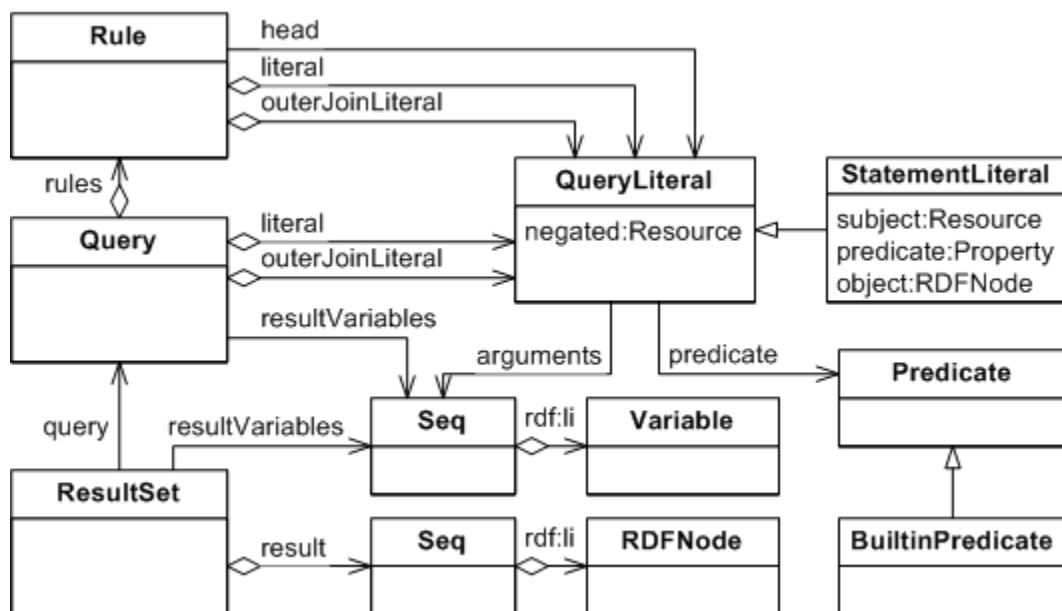


## 5. EQM

First, the QEL query has to be parsed and translated into objects by Edutella. The object data model used by Edutella for this is called EQM (Edutella Query Model).

As this has to be done for all wrappers in the same way it is parsed by Edutella itself. Edutella receives the query as a text file formulated in QEL which is based on RDF. And in order to be translated by an object-oriented program (the wrapper) it must be represented as a set of objects. Thus, the EQM serves as a step on the way to the final translation and as an interface language between Edutella and its wrapper for bringing the query from Edutella to a local data base.

Inside EQM the whole query is a "Query" object which contains a list of literals, a list of outerjoinliterals, a sequence of result variables and a list of rules. The literals and outerjoin literals are represented by "QueryLiteral" objects. They contain a predicate and a sequence of arguments. There are two subclasses of QueryLiteral: the "StatementLiteral" and the "BuiltinLiteral". The former one contains a subject, a predicate and an object. It is the representation of the classical RDF triple. The subject may be a RDF resource or a variable. The predicate may be a RDF property or a variable and the object may be an "RDFNode". The "BuiltinLiteral" is the same triple like RDF sentence but only with the predefined predicates of Edutella. An "RDFNode" is the object which represents the nodes of a RDF graph. Subclasses are the "Resource" and the "Literal". A subject of a StatementLiteral must never be a "Literal". This is neither allowed in QEL nor in RDF. There is also the "ResultSet" which contains the query it belongs to, a sequence of result variables and the results values which is a sequence of "RDFNodes".



## 6. Mapping

### A. Mapping Structure

The RQL Query is of the basic structure:

```
Select <resultVariables>  
From <Part of a RDF Graph>  
Where <Conditions>  
Using Namespace <Namespace Definitions>
```

The <resultVariables> is a list of Variables which will be replaced by data in the Result table. Since they correspond to the sequence "resultVariables" of the class Query in the EQM Model, the Selectclause is built by adding the unique label of all resultVariables (which are no outerjoin result variables). The Labels are separated by ",".

The only exceptions are variables for predicates. In QEL they look like every variable but in RQL they have to start with a "@".

During the building of the selectclause it is tested for every single variable if it is contained in the vector and a "@" is inserted in front of the label of the variable in the Selectclause.

For an example translation with a predicate variable serves the translation of QEL-example-query1 in the appendix:

```
SELECT  VAR1,@VAR2,VAR0 FROM  {VAR1} @VAR2 {VAR0}
```

There are several possibilities for describing a part of a RDF Graph in the Fromclause in RQL. I have chosen the notation <subject> <predicate> <object>, <subject> <predicate> <object>, ... since this notation is similar to the RDF Graph description in QEL.

A part of the graph is described by a list of all his edges (the properties or predicates) and the two nodes which are on its ends (subject and object). The edges are separated by "," and the order of the edges is not important. If a subject and an object are both variables and they have the same variable label name it is clear to the data base that they are the same node since the variable label names are unique.

This clause is the only one in which the RDF node variables have to be surrounded by "{ }". This indicates that they are object variables and that the results must be literals like strings or numbers or resources like URLs. A predicate variable starts here with a "@" , too.

As it is not possible to mention resources or literals like strings or integers in the Fromclause it is necessary to replace them by variable labels. These variables are bound later to their values of the QEL query.

Resources are replaced by "R" and a resourceVariableCounter which starts with one and is increased every time a resource appears. Literals are replaced by "L" and a literalVariableCounter which starts with one and is increased every time a literal appears.

And these variables have to be bound to their values in the whereclause. For example: "R11 = <http://www.l3s.com/bachelorthesis.htm>" or "L3 = 100".

A good example for the using of two literals in the fromclause and a resource and a literal mentioned in the query is the translation of the QEL-example-query6 in the Appendix:

```
SELECT VAR0 FROM {VAR0} n0:type {R1} , {VAR0} n1:title
{L1}WHERE R1 like "http://www.lit.gel/types#Book" and L1
like "Artificial Intelligence" Using Namespace n0 =
&http://www.w3.org/1999/02/22-rdf-syntax-ns#, n1 =
&http://purl.org/dc/elements/1.1/
```

## B. BuiltInLiterals

The whereclause also contains the conditions expressed by BuiltInLiterals. The single conditions have to be separated by "and" operators. A query which contains only BuiltInLiterals and no StatementLiterals is not allowed in this mapping since it is complex in RQL to express a query like "Give me all strings which start with an "a". But you may make such a query in a way like "Give me all Z from a sentence <X(subject) Y(predicate) Z(object)> which start with an "a". This is a little bit complicated but the power of the QEL queries is not constricted.

## C. Negation and BuiltInPredicates

BuiltInLiterals are the predefined literals of QEL. They are the only literals which may be negated in QEL. Fortunately, the negation in RQL is quite simple. The whole expression representing a QEL BuiltInLiteral in the whereclause of RQL is parenthesized and a "!" is put in front of it. There are four BuiltInLiterals allowed so far in the wrapper. Firstly, "equal", which does not allow wildcards and is translated by a "=" into RQL.

Secondly, "like" which allows wildcards and is the same in RQL. Thirdly, "greaterThan" and "LessThan" which are translated to ">" and "<". Whereas it is mandatory that the subject of a StatementLiteral is a resource and not a literal the BuiltInLiteral may be start with a literal. But RQL does not work properly in some cases if both arguments are literals. But this does not make much sense anyway. The translation of the QEL-example-query5 in the appendix is an example for negation and the equals operator:

```
SELECT VAR1,VAR0 FROM {VAR1} n0:title {VAR0}WHERE
!(VAR0 = "Artificial Intelligence") Using Namespace n0 =
&http://purl.org/dc/elements/1.1/
```

#### D. Namespaces

The Using Namespaceclause defines all necessary namespaces for the predicates. Since it is not possible to use a namespace in front of the predicate in the Fromclause there has to be a namespace variable instead followed by a ":". The namespacevariable looks like "n <namespacevariablecounter>". The namespacevariablecounter starts with 1.

In the Using namespaceclause all these variables are listed and each one is followed by "=" and the namespace. The namespaces are extracted from the QEL predicate by cutting off the part before the "#". If there is no "#" in the namespace the last "/" will be used instead.

But there is a problem with these namespaces in RQL. It is not allowed in RQL to define the same namespace twice and there are plenty of predicates which have the same namespaces.

For that reason there has to be some kind of an Administration inside the wrapper for organizing these namespaces. Every time a namespace appears which is unknown in the namespaces it is added to the other namespaces. If a predicate appears which namespace is already known the namespacevariable of this namespace definition has to be used in the Fromclause in front of the predicate.

For an example translation with namespaces please see the query 3 to 14 in the appendix.

#### E. OuterJoin Literals

The outerjoin literal is actually not a feature of RQL. But it is possible to make it work with a trick. Every outerjoin literal from QEL gets its own

RQL query with a selectclause, fromclause and whereclause. This becomes a sub query of the query formed by the nonouterjoin literals. The sub query will be parenthesised and put into the selectclause of the main query. It is separated by a “,” from the other entries of the selectclause. There may be an unlimited amount of such sub queries in the selectclause. This will be treated as an optional attribute of the main query by the rql interpreter.

It only makes sense to use StatementLiterals and QueryLiterals which require rules and are covered below as outerjoin literals. BuiltInLiterals should not be used as outerjoin literals. If they contain only variables from other outerjoin literals they are not allowed in QEL. For example:

Book(X), Title\*(X,S), Like\*(S,"%www%")

If there is a title it will be added to the results and the BuiltInLiteral can not be taken into consideration.

If the outerjoin BuiltInLiteral contains a variable from the main query it does not make sense. The outerjoin literals ask for optional information. But the BuiltInLiteral formulates a constraint which the data has to fulfil for being into the result set. An optional constraint is senseless. Either it is a mandatory constraint or it is an optional attribute. It cannot be both at the same time.

The translation of the example-query14 of Edutella is a good example for outerjoin literals in RQL (for the QEL query of this please see Appendix):

```
SELECT VAR2,VAR1, (SELECT VAR4 FROM {VAR2} n0:creator
{VAR4}), (SELECT VAR3 FROM {VAR2} n0:language {VAR3}),
(SELECT VAR0 FROM {VAR2} n0:identifier {VAR0}) FROM
{VAR2} n0:title {VAR1} Using Namespace n0 =
&http://purl.org/dc/elements/1.1/
```

## 7. Decision Tree

If there are no disjunctive rules in a query the translation is only linear. But if there are several rules for one queryliteral the translation process has to apply each of them.

This will produce more than one query and it is necessary to connect the results somehow.

In SQL it is possible to nest these queries into each other. The resulting structure represents afterwards the order in which the rules have been applied.

For example:

```
SELECT STMT0.PREDICATE STMT0_PREDICATE, STMT0.ARG0 STMT0_ARG0
FROM ( SELECT
'file:./src/test/net/jxta/Edutella/eqm/queries/qel_8.xml#aibook' PREDICATE,
STMT0.SUBJECT ARG0
FROM EDUTELLA_STATEMENTS STMT0, EDUTELLA_STATEMENTS STMT1
WHERE STMT0.PREDICATE = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
AND
STMT0.OBJECT = 'http://www.lit.gel/types#Book' AND
STMT1.PREDICATE = 'http://purl.org/dc/elements/1.1/title' AND
STMT1.OBJECT = 'Artificial Intelligence' AND
STMT0.SUBJECT = STMT1.SUBJECT
UNION SELECT
'file:./src/test/net/jxta/Edutella/eqm/queries/qel_8.xml#aibook' PREDICATE,
STMT0.SUBJECT ARG0
FROM EDUTELLA_STATEMENTS STMT0
WHERE STMT0.PREDICATE = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
AND
STMT0.OBJECT = 'http://www.lit.gel/types#AI-Book' ) STMT0
WHERE STMT0.PREDICATE =
'file:./src/test/net/jxta/Edutella/eqm/queries/qel_8.xml#aibook'
```

This is probably useful because you can work through the rules and translate into your data base language simultaneously.

In RQL it looks like this:

```
Select X From ((Select Var0
                From {Var0} type {Resource1}, {Var0} title {Literal1}
                Where Resource1 = book and Literal1 = "Artificial
Intelligence") union
                (Select Var0
                From {Var0} type {Resource2}
                Where Resource2 = AIbook)) {X}
```

But this has a big disadvantage: it works only with one single argument for the nested query. This single argument is here named {X} in the main query and {Var0} in the subquery. In SQL more than one argument is possible but there is no chance to do this in RQL.

The arguments of the subqueries are the arguments of the disjunctive rules. But in QEL queries have often more than one argument. Therefore, this is no solution for the translation of disjunctive rules into RQL.

A solution which allows more arguments is to produce various RQL queries from one QEL query and uniting them by the union operator.

Without disjunctive rules there will be only one RQL query but every disjunctive rule for a queryliteral will produce one more query. If there is a second query literal to which disjunctive rules will be applied the number of the disjunctive rules of the first queryliteral have to be multiplied by the number of the rules for the second literal to get the number of resulting RQL queries. It is important to use every combination of the two sets of rules for getting all possible results.

My first attempt for creating the RQL queries was to have a vector which contains more vectors and each of them contains the literals of one query. It starts with one inner vector which contains all literals of the query. These literals are searched until a literal is found which requires rules or the end is reached. The literal will then be replaced by the literals of the rule body. If there are disjunctive rules the inner vector will be cloned. There have to be one clone for every rule minus one. The literal in the inner vector and its clones will be replaced by the rule bodies. This vector construction has to be searched until the end of the last vector is reached and the cloning and replacing has to be done for each found queryliteral which requires rules. The method was recursive so that this one method could do all the work.

This algorithm worked and it was a proper solution. The problem with it was that this was very complex and I had still to add more functions like the outerjoin literals in the rules for example. As it was already so complex the adding of new functions was difficult. And it was not very object-oriented but more like imperative programming code.

Therefore, I created the current solution:

If you try to visualize the nondisjunctive query as a line and you make a new branch for every disjunctive rule you use, you will get a tree. A Decision Tree since every node is a decision.

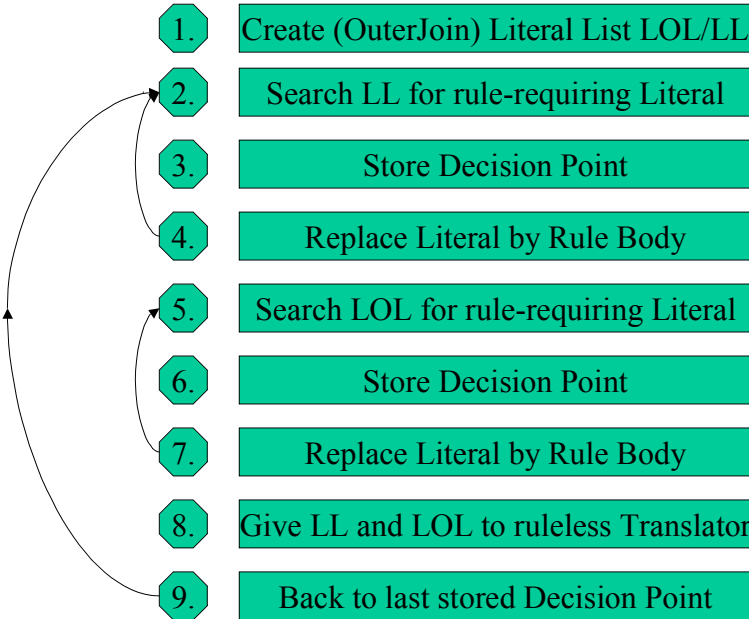
The root of the tree is the Edutella query itself or the beginning of the literals. And the leaf is the end of the literals. The path from the root to one leaf represents one query containing only literals which no longer require rules. This rulefree query can be translated by a simpler subprogram.

Outerjoin literals can be searched after the nonouterjoin literals. As they can also contain rule requiring literals there may be also decision nodes in these branches of the tree.

This Decision Tree is a list of Decision points. At the beginning this list is empty. The algorithm starts to search the literals and if it needs to use disjunctive rules a new Point of Decision is created and stored into the list of Decision Points. A Decision Point stores the nonouterjoinliterals, the outerjoinliterals, the currently analyzed queryliteral, the rules and the information if the nonouterjoin literals are already done. Then the Point of Decision clones the currently processed literals (outerjoin or nonouterjoin literals) and replaces the currently analysed literal by the first rule body. The tree works on with these literals and stops if the end of the literals is reached or another disjunctive rule has to be applied. The tree decides when the nonouterjoin literals are finished and goes on with the outerjoin literals. If the outerjoin literals are finished too a leaf is reached. The tree gives the current literals and outer join literals to a translation sub routine. This returns a RQL query and the tree unites the query with the

other translated queries. Next, the tree loads the Decision Point last stored and uses its data as current literals and goes on. If the list of Points of Decision is empty again the whole QEL query has been translated and can be sending to the data base.

This solution is very object oriented and therefore much easier to understand. Thus it is easier to make changes and to correct errors.



Implementation in Java

The main wrapper class is the DecisionTree class. It implements the interface QueryFormat and has to be instantiated before using. The interface is necessary in order that the Edutella main program recognizes the DecisionTree as a wrapper.

As the wrapper is instantiated only once and is used then for every query it is impossible to use the constructor to load all the data like literals from the QEL query into the appropriate fields of the object. The method which does the translation and gets the queries is called format. The first thing it does when it is called is to use the initialize method which sets the fields of the DecicionTree on the new values of the queries.

A do while loop is then running through all the PointOfDecisions until the list of them is empty. At the beginning the list is empty but since it is a do while loop its body has to be done at least one time. Inside it are two while loops and the translation method called for the translation of the single RQL query when a leaf of the tree is reached. The two loops take care of the literals. The first one looks through the nonouterjoin literals and the second loop through the outerjoin literals.

They run until there are no more literals. If they find a literal which require a rule they search for the adequate rules and create a new PointOfDecision. Then they update the tree with the update method. It calls the makeYourChoice method of the PointOfDecision and gets from it a Decision object. If the Point object has more rules available the update method stores it into the PointOfDecision list of the Tree object. The Decision object is then used to give the Tree new values for the literals and outerjoinliterals.

The translation of the single RQL queries are assembled in an appropriate way and are returned as the result of the method format.

The method makeYourChoice has to do the replacing of the rule-requiring literals by rules and it must sort the nonouterjoin literals and the outerjoin literals into the correct fields of the Decision it returns. It needs for this the information if the nonouterjoin literals have already been finished. Please see the chapter about the outerjoin literals for details.

The Decision Tree is an appropriate algorithm for nested rules. Please look at the translation of the example queries 8 and 10. Both have the same meaning but number 8 uses 2 rules and number 10 uses 4 rules. The four rules express the same meaning like the two rules in number 8 but the bodies of the two rules in example 10 have two nested rules which have the same meaning of the bodies of the rules from example 8. (example 10 asks also for all tuples (X,Y,Z) with X as the AIBook). The Tree is able to solve the nested rules and the queries look the same. (Please see also the Appendix for the QEL queries).

Translation of example query 8:

```
seq((SELECT VAR0 FROM {VAR0} n0:type {R1} , {VAR0} n1:title
  {L1}WHERE R1 like "http://www.lit.qel/types#Book" and L1
  like "Artificial Intelligence") , (SELECT VAR0 FROM
  {VAR0} n0:type {R1} WHERE R1 like
  "http://www.lit.qel/types#AI-Book")) Using Namespace n0 =
  &http://www.w3.org/1999/02/22-rdf-syntax-ns#, n1 =
  &http://purl.org/dc/elements/1.1/
```

Translation of example query 10:

```
seq((SELECT VAR0,@VAR1,VAR2 FROM {VAR0} @VAR1 {VAR2}, {VAR0}
  n0:title {L1}, {VAR0} n1:type {R1} WHERE L1 like
  "Artificial Intelligence" and R1 like
  "http://www.lit.qel/types#Book") , (SELECT
  VAR0,@VAR1,VAR2 FROM {VAR0} @VAR1 {VAR2}, {VAR0} n1:type
  {R1} WHERE R1 like "http://www.lit.qel/types#AI-Book"))
  Using Namespace n0 = &http://purl.org/dc/elements/1.1/,
  n1 = &http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

## 8. Cross Product

A query can contain both rule-requiring QueryLiterals and ruleless QueryLiterals (StatementLiterals and BuiltInLiterals). It could be easier to write two subprograms. One method for all literals which require no rules and one for the translation of the direct literals and their rules. But if the translation process distinguish between the different Literals since they require different rules or no rules at all it will be complicated to determine if the ruleless QueryLiterals are connected to one or more of the direct queryliterals. They are connected together if they have at least one common variable. If the rule-requiring queryliterals were translated separately and the ruleless ones were used with every rule-requiring literal, the ruleless ones would possibly used together with a rule-requiring one with which they have no variable in common.

The ruleless literal would not be bound to the rule-requiring literal with which it has a variable in common in this subquery and this would produce plenty of results which were not required.

For example:

```
AIbook(X)
Novel(U)
(X,Y,Z)
« rules for AIbook(X) and Novel(U) »
```

The last literal has to be translated together with the first literal but must not be used together with the second literal.

Another Problem of separating the queryliterals is that it would result in totally different queries in rql. The results of these queries would be united.

But this is not what the above query means. The literals are together in one query and the result has to be a table with the cross product of the result sets of each literal.

And the ruleless literal has always to be connected to its rule-requiring queryliteral.

For example: the results of AIbook(X) and (X,Y,Z) could be ((a,b,c),(d,e,f)) and the results of Novel(U) could be (u,v). The result of the whole query should not be ((a,b,c),(d,e,f),u,v) but ((u,a,b,c),(v,a,b,c),(u,d,e,f),(v,d,e,f)).

For these reasons a translation process should do all literals together. This would solve both problems, the cross product of the results and the ruleless literals which belong or do not belong to rule-requiring literals.

But for a translation development process it could be useful to separate the tasks. Two developers could work simultaneously on the tasks or one developer could first do the ruleless part and test it and then he could implement the complicated rule part. And this separation of the task is still

possible. First there should be implemented and tested the ruleless translation method. Then a method (which uses auxiliary methods) has to be developed which firstly works through all rules without separating the queryliterals and then gives the whole set of literals to the ruleless interpreter.

The decision tree translates all literals together and computes automatically the cross product of the results.

If there are no queryliterals which use disjunctive rules the cross product of the results is computed by the RQL interpreter like in the example above.

The cross product of the results of a query like:

StatementLiteral 1, QueryLiteral 1, StatementLiteral 2,  
QueryLiteral 2

With two disjunctive rules, rule 1a and rule 1b for the former QueryLiteral and two rules 2a and 2b for the latter Queryliteral will be computed by the decision tree.

The decision tree produces four queries (in this order):

1. StatementLiteral 1, QueryLiteral 1a, StatementLiteral 2,  
QueryLiteral 2a
2. StatementLiteral 1, QueryLiteral 1a, StatementLiteral 2,  
QueryLiteral 2b
3. StatementLiteral 1, QueryLiteral 1b, StatementLiteral 2,  
QueryLiteral 2a
4. StatementLiteral 1, QueryLiteral 1b, StatementLiteral 2,  
QueryLiteral 2b

The result of each query will be computed by the RQL interpreter. It will contain the cross product of its results.

The queries will be united by the decision tree and that means that also the results will be united.

As the decision tree has already generated the cross product of the disjunctive rules, also the union of the single results of the queries will be the cross product of all results.

## 9. Unification

Literals which belong together use the same variables for defining their connection.

But this does not apply for rules. A rule can be applied to a queryliteral if they have the same predicate name and the same amount of arguments. The variable names in the rule are not important.

Thus, before using the rule it should be copied and the variables in the copy should be renamed. It is necessary to replace the first variable in the head of the rule by the first variable in the queryliteral and so on.

And it has not only to be replaced in the head but a replacement in the head has to be done also in every literal in the body. The outerjoinliterals have also to be taken into consideration.

For example the rule:

```
AIBook(Y,Z) :- Type(Y,"book"), Title*(Y,Z)
```

Can be applied to:

```
AIBook(X,S)
```

But the rule has to be copied and the variables have to be replaced:

```
AIBook(X,S) :- Type(X,"book"), Title*(X,S)
```

The original of the rule has to remain in the query since it may be useful for another literal which may require other variable names.

The replacement is necessary for binding the literals in the body of the rules to the literals in the context of the literal requiring the rule. These context literals may be bound to the rule-requiring queryliteral and shall be bound also to the new literals from the rules.

For example:

```
AIBook(X,S)  
(X,Y,Z)
```

Will be transformed to :

```
Type(X,"book")  
Title*(X,S)  
(X,Y,Z)
```

Another possibility is that the queryliteral contains constants. These constants have to be inserted into the rule in order to bind the rule to it.

And the rule can contain a constant. In this case it is important not to overwrite it.

Example rule:

```
AIBook(X, "Artificial Intelligence") :- Type(X,"book"), Title(X, "Artificial Intelligence")
```

Annotation:

Title(X, "Artificial Intelligence") is in this case a constraint for the books which shall be selected.

Title\*(X,S) in the query above is an additional attribute for which the query asks. A combination of these things like Title\*(X, "Artificial Intelligence") does not make sense. It is an "optional constraint" which is a contradiction in terms. It would cause an error during the execution of the query in RQL.

It is only useful to ask like\*/equals\*/=(X, "Artificial Intelligence").

But if the queryliteral and the rule have constants which are not equal at the same position it is not possible to apply the rule for the literal.

## 10. How to become an OuterJoinLiteral

There are four possibilities for a literal to become an OuterJoinLiteral. The wrapper has to recognize them as outerjoins and to translate them in a separated query in brackets in the Select close of the main query. The decision tree is taking care of this matter by separating the literals into two lists: the list Of Literals, and the list Of OuterJoinLiterals. When the decision tree has reached a leaf all rules for this query have been evaluated and the two lists are calculated to a RQL query by the auxiliary translation method.

The normal way of being outerjoin is that the literal is marked as an OuterJoinLiteral in the query by a definition in the head of the query like:

```
<qel:outerJoinLiteral rdf:resource="#st1" />
```

The decision tree is handling it in the initialize method and it stores all OuterJoinLiterals into the appropriate list.

The second way is that the literal is stored in the outerjoinliterals of a rule and the queryliteral requiring the rule is stored in the nonouterjoinliterals of the query.

This case is handled by the makeYourChoice method in the PointOfDecision. It stores these kinds of literals into the list of outerjoins in the Decision. The Decision is then appropriately evaluated by the update method in the Decision Tree. The literals are also stored into the list Of OuterJoin Literals of the Tree and translated correctly later.

The third way is that the queryliteral requires a rule, which contains nonouterjoinliterals, but the rule-requiring literal is stored in the outerjoinliterals of the query or of another rule.

The format method of the Decision Tree will analyze the outerjoinliterals in the second while loop. This loop will store the information that this literal is outerjoin into the PointOfDecision by setting the last argument of the its constructor true.

The update method will call then the makeYourChoice method of the Decision Point and it will store both the nonouterjoinliterals and the outerjoinliterals into the outerjoinliterals of this query.

And this is the fourth way of becoming an outerjoinliteral. A literal is stored in the outerjoinliterals of a rule used for evaluating an outerjoinliteral of the query.

It is possible to make it as complicated as you want to. An outerjoinliteral of a query may need a rule for evaluation and this rule may contain an outerjoinliteral which may require a rule which may have outerjoinliterals and so on.

## 11. Combination of OuterJoin Literals, Namespaces and disjunctive rules in one single query.

Unfortunately, RQL does not allow using the **Union** operator and OuterJoin literals in one query. A query like:

```
( Select X, (Select Z From {X} title {Z})  
From {X} creates {Y} )  
Union  
( Select U, (Select W From {U} exhibited {W})  
From {U} title {V})
```

Will create an error message like "**Type error in operation: union**" as a result.

The only solution for this is to use the **seq** operator instead of the **Union**. It looks like:

```
Seq(( Select X, (Select Z From {X} title {Z})
From {X} creates {Y} ),
( Select U, (Select W From {U} exhibited {W})
From {U} title {V}))
```

This works but it does not have such a nice result table like **union**. **Union** writes all result tuples in one single table and the table can be translated into ResultSet Objects easily. But the seq operator creates one table which contains the other tables.

But it is not a worse disadvantage since the user never sees these kinds of tables. The method translating the RQL – tables into QEL – ResultSet has to deal with these various tables.

The next difficult question is how to integrate the namespaces into the query which considers disjunctive rules.

Example query:

```
( Select X
From {X} n1:creates {Y}Using Namespace n1 =
&http://139.91.183.30:9090/RDF/VRP/Examples/demo/culture.rdf# )
union
( Select U
From {U} n1:title {V}Using Namespace n1 =
&http://139.91.183.30:9090/RDF/VRP/Examples/demo/admin.rdf# )
```

RQL throws an error message for this query. The text of it is: **“Syntactical error on token: Using”**. Each query between the brackets is a correct query and works. And so does the hole query without the “Using namespaces” clauses and without the “n1:” variables. But the combination of the two queries and the two namespaces cannot be evaluated by the RQL parser. Thus, it has to be one single “Using Namespace” clause, which defines the namespaces of both single queries, at the end of the query. In our example:

```
( Select X
From {X} n1:creates {Y} )
union
( Select U
From {U} n2:title {V})Using Namespace n1 =
&http://139.91.183.30:9090/RDF/VRP/Examples/demo/culture.rdf# , n2
= &http://139.91.183.30:9090/RDF/VRP/Examples/demo/admin.rdf#
```

Note that the namespace declarations are also valid for the first query although there are no brackets around the queries. But as mentioned above it is not possible to use union and outerjoin literals together. Therefore, it has to be:

```
seq(( Select X, (Select Z From {X} title {Z})
From {X} n1:creates {Y} ),
( Select U, (Select W From {U} exhibited {W})
From {U} n2:title {V}))Using Namespace n1 =
&http://139.91.183.30:9090/RDF/VRP/Examples/demo/culture.rdf# , n2
= &http://139.91.183.30:9090/RDF/VRP/Examples/demo/admin.rdf#
```

The namespace declarations have to be outside of the seq - brackets. The interesting question is now if the namespaces do also work for the outerjoin literals. The answer is that RQL accepts a query like:

```
seq(( Select X, (Select Z From {X} n2:title {Z})
From {X} n1:creates {Y} ),
( Select U, (Select W From {U} n1:exhibited {W})
From {U} n2:title {V}))Using Namespace n1 =
&http://139.91.183.30:9090/RDF/VRP/Examples/demo/culture.rdf# , n2
= &http://139.91.183.30:9090/RDF/VRP/Examples/demo/admin.rdf#
```

Thus, it is possible to use disjunctive rules, which can contain outerjoin literals, which are using namespaces. But this has to be formulated in the way shown above. The normal formulation with namespace definitions for each query and the union operator does not work. Also you have to take care of the namespaces. There must not be duplicate namespaces or RQL will show the error message:

**"Two namespaces with same URI:"** followed by the namespace, which caused the error.

## 12. RQL - BUG

Unfortunately, has RQL/Rssdb not yet reached its final version. It is difficult to install and configure Rssdb and RQL. And there are some bugs in the RQL interpreter. One bug is especially difficult for the wrapper. If the last literal in the where clause of the main query in the last query of a sequence of more than one query is negated, the parser will react with an error message and a nice smiley:

### **Syntactical error on token: )**

An example for this kind of query is the translation of the test query 11: (See Appendix for the QEL example query 11)

```
seq((SELECT VAR0,@VAR2,VAR1 FROM {VAR0} @VAR2 {VAR1}, {VAR0}
n0:type {R1} WHERE R1 like "http://www.lit.qel/types#AI-Book"
and !(VAR1 like "*Software*")) , (SELECT VAR0,@VAR2,VAR1 FROM
{VAR0} @VAR2 {VAR1}, {VAR0} n1:title {L1}, {VAR0} n0:type {R1}
WHERE L1 like "Artificial Intelligence" and R1 like
```

```
"http://www.lit.gel/types#Book" and !(VAR1 like
"*Software*")) Using Namespace n0 =
&http://www.w3.org/1999/02/22-rdf-syntax-ns#, n1 =
&http://purl.org/dc/elements/1.1/
```

If the order of the literals in the where clause is changed in a way that the negated literal is at the first position (highlighted) it astonishingly works:

```
seq((SELECT VAR0,@VAR2,VAR1 FROM {VAR0} @VAR2 {VAR1}, {VAR0}
n0:type {R1} WHERE R1 like "http://www.lit.gel/types#AI-Book"
and !(VAR1 like "*Software*")) , (SELECT VAR0,@VAR2,VAR1 FROM
{VAR0} @VAR2 {VAR1}, {VAR0} n1:title {L1}, {VAR0} n0:type {R1}
WHERE !(VAR1 like "*Software*") and L1 like "Artificial
Intelligence" and R1 like "http://www.lit.gel/types#Book"))
Using Namespace n0 = &http://www.w3.org/1999/02/22-rdf-syntax-
ns#, n1 = &http://purl.org/dc/elements/1.1/
```

But the and operator is commutative and this change of the order should not make any difference. This means it is a Parser error in RQL.

But for example the test query 5 which has also a negated literal in the where clause as last literal works only with all closing brackets. The difference is that it has only one query in the sequence.

Correct Translation with all brackets:

```
seq((SELECT VAR1,VAR0 FROM {VAR1} n0:title {VAR0}WHERE
!(VAR0 = "Artificial Intelligence"))) Using Namespace n0 =
&http://purl.org/dc/elements/1.1/";
```

Another trick to make it work is to leave out the last bracket (highlighted). This is not very logical since the query has then five opening brackets but only four closing brackets:

```
seq((SELECT VAR0,@VAR2,VAR1 FROM {VAR0} @VAR2 {VAR1}, {VAR0}
n0:type {R1} WHERE R1 like "http://www.lit.gel/types#AI-Book"
and !(VAR1 like "*Software*")) , (SELECT VAR0,@VAR2,VAR1 FROM
{VAR0} @VAR2 {VAR1}, {VAR0} n1:title {L1}, {VAR0} n0:type {R1}
WHERE L1 like "Artificial Intelligence" and R1 like
"http://www.lit.gel/types#Book" and !(VAR1 like "*Software*"))
Using Namespace n0 = &http://www.w3.org/1999/02/22-rdf-syntax-
ns#, n1 = &http://purl.org/dc/elements/1.1
```

This is better than the first trick because there may be only negated literals and it would be not possible to put the negated one at the first position.

In the wrapper is a switch called RQL\_BUG which is set to true in the class Decision Tree. But if the bug will be fixed one day it is possible to set it to false and the wrapper will use always all brackets which are required. It would increase the performance to delete all involved lines:

```
{DecisionTree.lastLiteralIsNegated = true;
```

in class RQLQueryFormatAuxiliaryMethods and

```
} else DecisionTree.lastLiteralIsNegated = false;
```

in this class, too. Also:

```
if (!(RQL_BUG & lastLiteralIsNegated & moreThanOneQuery))
```

and

```
moreThanOneQuery = true;
```

in the Decision Tree class and the declaration of the variables at the top of the class and their initialization in the method initialize():

```
static final boolean RQL_BUG = true;  
static boolean lastLiteralIsNegated;  
static boolean moreThanOneQuery;
```

### 13. Results

The Results in Edutella have to be formulated in QEL. For every Result a "ResultSet" is created which contains the query and a variable binding for all results. The variable binding contains a variable and the value it is bound to in this binding.

For example the RQL query:

```
seq((SELECT VAR3,VAR0, (SELECT VAR2 FROM {VAR3} n1:creates  
{VAR2}),  
(SELECT VAR1 FROM {VAR3} n1:technique {VAR1}), (SELECT VAR4  
FROM  
{VAR3} n1:exhibited {VAR4})) FROM {VAR3} n0:title {VAR0}))  
Using Namespace  
n0 = &file:/home/david/tmp/Rssdb.new/examples/admin.rdf#, n1 =  
&file:/home/david/tmp/Rssdb.new/examples/culture.rdf#
```

asks for all objects which have a title. The result is their resource, their title and three optional attributes. These are what they have created, what the technique is with which they have been created and where they are exhibited. These optional attributes do not make sense for every object but the query for all things with a title is a very general questions and the attributes are only outerjoin literals.

The result formulated in RQL could look like:

```

<rdf:Seq>
<rdf:li>
<rdf:Bag>
<rdf:li>
<rdf:Seq>
<rdf:li rdf:type="resource"
rdf:resource="http://www.artchive.com/crucifixion.jpg"/>
<rdf:li rdf:type="string">Crucifixion</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
<rdf:li rdf:type="resource" rdf:resource="http://www.rodin.fr"/>
</rdf:Bag>
</rdf:li>
</rdf:Seq>
</rdf:li>
<rdf:li>
<rdf:Seq>
<rdf:li rdf:type="resource" rdf:resource="http://www.museum.es"/>
<rdf:li rdf:type="string">Reina Sofia Museum</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
</rdf:Seq>
</rdf:li>
<rdf:li>
<rdf:Seq>
<rdf:li rdf:type="resource" rdf:resource="http://www.rodin.fr"/>
<rdf:li rdf:type="string">Rodin Museum</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
</rdf:Seq>
</rdf:li>
<rdf:li>
<rdf:Seq>

```

```

<rdf:li rdf:type="resource"
rdf:resource="http://www.artchive.com/rembrandt/abraham.jpg"/>
<rdf:li rdf:type="string">Abraham and Isaac</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
<rdf:li rdf:type="string">oil on canvas</rdf:li>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
</rdf:Seq>
</rdf:li>
<rdf:li>
<rdf:Seq>
<rdf:li rdf:type="resource" rdf:resource="http://www.louvre.fr/" />
<rdf:li rdf:type="string">Louvre Museum</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
</rdf:Seq>
</rdf:li>
<rdf:li>
<rdf:Seq>
<rdf:li rdf:type="resource"
rdf:resource="http://www.photojournal.com/classicart/italmasters/michelange
lo/sculptur/descent.jpg"/>
<rdf:li rdf:type="string">Descent</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
<rdf:li>
<rdf:Bag>
</rdf:Bag>
</rdf:li>
</rdf:Seq>
</rdf:li>
</rdf:Bag>
</rdf:li>
</rdf:Seq>
</RDF>

```

The result is stored by the RQL interpreter in a file which is configured with its path in Edutella. The file name given to Edutella is then extended by a number which is increased by one for every Edutella query. Next time Edutella starts the number is set to null again. This is important if more than one query are performed in Rssdb at the same time. Every query has then its one result file and does not overwrite the result of another query before Edutella can read the result file.

This RDF file from Rssdb is then parsed by the Jena Parser which is a special parser for RDF files. The text file is transformed into a set of objects and the objects are translated by the wrapper into a valid Edutella ResultSet.

The wrapper has to analyse the outer seq constructions for its bags. Every bag contains the results of one of the queries of the sequence of the original query. Every bag contains in turn several sequences which represent the rows of a result table. They are result tuples and each one contains a value for every result variable from every selectclause. Remember that there is one selectclause in the main query and one selectclause for every outerjoin subquery.

The inner sequence starts with the values for the nonouterjoin literals since they have always the first position in the RQL query. They are followed by the outerjoinliterals each one of them wrapped in its own bag because they are formulated as subqueries. Their order comply with their appearance in the RQL query.

For every tuple the wrapper creates a Result and adds it to the ResultSet. Then the ResultSet is filled with the variable bindings for the attributes of every Result.

These attributes are the same ones for every Result and they are defined already in the QEL query as the result variables. The outerjoin attributes which does not have a value are bound to null.

#### Annotation:

In the Java implementation there is a problem with the EQM model of the query. Somehow the literals have all the same order like in the QEL query but the result variables are mixed up a little bit. Thus, the result variables do not have the same order like the query literals. But the translator of the wrapper uses exactly the order of the literals for translating the query and also the RQL interpreter uses the order of the literals as the order the results will have. But the variable names are no longer mentioned inside the RQL result. The wrapper has to assign the result variables to their values without having the names of the variables in the RQL result set and without having the order of the variables in the query. This is solved by creating a new list and inserting the result variables into it during the creation of the selectclauses. This assures that the correct order of the results is known to the wrapper.

The results look in the Jena model like:

```
ResultTuple:
Resource: http://www.artchive.com/crucifixion.jpg
Literal: Crucifixion
Bag: Bag: Bag: OuterJoinResource: http://www.rodin.fr
ResultTuple:
Resource: http://www.museum.es
Literal: Reina Sofia Museum
Bag: Bag: Bag:
ResultTuple:
Resource: http://www.rodin.fr
Literal: Rodin Museum
Bag: Bag: Bag:
ResultTuple:
Resource: http://www.artchive.com/rembrandt/abraham.jpg
Literal: Abraham and Isaac
Bag: Bag: OuterJoinResource: oil on canvas
Bag:
ResultTuple:
Resource: http://www.louvre.fr/
Literal: Louvre Museum
Bag: Bag: Bag:
ResultTuple:
Resource:
http://www.photojournal.com/classicart/italmasters/michelangelo/sculptur/descent.jpg
Literal: Descent
Bag: Bag: Bag:
```

This is only an example display of this object model for testing purposes. Normally there is no visualisation of these data.

And this is how a visualisation of the Edutella ResultSet looks like ( but not every information is presented here, for example the variable names are missing):

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:edu='http://www.Edutella.org/qel#' >
  <edu:ResultSet>
    <edu:query

rdf:resource='file:./src/test/net/jxta/Edutella//eqm/queries/qel_18.xml#gen
Query' />
    <edu:result>
      <rdf:Seq>
        <rdf:li
rdf:resource='http://www.artchive.com/crucifixion.jpg' />
          <rdf:li>Crucifixion</rdf:li>
          <rdf:li rdf:resource='http://www.rodin.fr' />
        </rdf:Seq>
      </edu:result>
    <edu:result>
      <rdf:Seq>
        <rdf:li rdf:resource='http://www.museum.es' />
          <rdf:li>Reina Sofia Museum</rdf:li>
        </rdf:Seq>
      </edu:result>
    <edu:result>
```

```

        <rdf:Seq>
          <rdf:li rdf:resource='http://www.rodin.fr/'/>
          <rdf:li>Rodin Museum</rdf:li>
        </rdf:Seq>
      </edu:result>
    <edu:result>
      <rdf:Seq>
        <rdf:li
rdf:resource='http://www.artchive.com/rembrandt/abraham.jpg'/>
          <rdf:li>Abraham and Isaac</rdf:li>
          <rdf:li>oil on canvas</rdf:li>
        </rdf:Seq>
      </edu:result>
    <edu:result>
      <rdf:Seq>
        <rdf:li rdf:resource='http://www.louvre.fr/'/>
        <rdf:li>Louvre Museum</rdf:li>
      </rdf:Seq>
    </edu:result>
  <edu:result>
    <rdf:Seq>
      <rdf:li
rdf:resource='http://www.photojournal.com/classicart/italmasters/michelange
lo/sculptur/descent.jpg'/>
        <rdf:li>Descent</rdf:li>
      </rdf:Seq>
    </edu:result>
  </edu:ResultSet>
</rdf:RDF>

```

## 14. Future Work and Conclusion

This work has discussed the architecture of a wrapper and how it shall translate queries. Most important points of this analysis were the cross product of the results, handling of nested rules, handling of nested outerjoin literals and the combination of rules and outerjoin literals. The decision tree algorithm has proven to be the best solution for calculating the cross product and for evaluating nested expressions including rules and outerjoin literals.

This work has also dealt with the integration of two semantic web technologies. Edutella is a P2P network for exchanging and searching for RDF data and metadata and Rssdb/RQL is a semantic data base for storing and retrieving RDF data. This enhanced the field of application of Edutella.

Future work might include using the new XML datatypes of Rssdb. The Edutella XML-based BuiltInLiterals language and datatype can be considered now by the wrapper but these new features of RQL were introduced right when I was finishing this work. And the translation of RQL into QEL could be a point of interest since you could query then Edutella with the RQL language. Another development could be new features for

the data base connection like storing and altering data in the Rssdb data base.

## Appendix - Examples for QEL queries and their translation into RQL

The following queries are meant for a set of data called knowledgebase.rdf. This file can be found in the Edutella source code for testing purposes. But The queries can be may serve also as theoretical examples for the translation.

Example-Query1 from Edutella: Asks for all (X,Y,Z) tuples.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
  <!DOCTYPE rdf:RDF (View Source for full doctype...)>
  _ <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:qel="http://www.edutella.org/qel#">
    _ <qel:Query rdf:about="#genQuery">
      <qel:literal rdf:resource="#st0" />
      _ <qel:resultVariables>
        _ <rdf:Seq>
          <rdf:_1 rdf:resource="#X" />
          <rdf:_2 rdf:resource="#Y" />
          <rdf:_3 rdf:resource="#Z" />
        </rdf:Seq>
      </qel:resultVariables>
    </qel:Query>
    <qel:Variable rdf:about="#X" />
    <qel:Variable rdf:about="#Y" />
    <qel:Variable rdf:about="#Z" />
    _ <qel:StatementLiteral rdf:about="#st0">
      <rdf:subject rdf:resource="#X" />
      <rdf:predicate rdf:resource="#Y" />
      <rdf:object rdf:resource="#Z" />
    </qel:StatementLiteral>
  </rdf:RDF>
```

Translation:

```
seq((SELECT VAR1,@VAR2,VAR0 FROM {VAR1} @VAR2 {VAR0}))
```

Example-Query3: Asks for all (X, title, Y)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
  <!DOCTYPE rdf:RDF (View Source for full doctype...)>
```

```

_ <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:qel="http://www.edutella.org/qel#" >
  _ <qel:Query rdf:about="#genQuery">
    <qel:literal rdf:resource="#st0" />
    _ <qel:resultVariables>
      _ <rdf:Seq>
        <rdf:_1 rdf:resource="#X" />
        <rdf:_2 rdf:resource="#Y" />
      </rdf:Seq>
    </qel:resultVariables>
  </qel:Query>
  <qel:Variable rdf:about="#X" />
  <qel:Variable rdf:about="#Y" />
  _ <qel:StatementLiteral rdf:about="#st0">
    <rdf:subject rdf:resource="#X" />
    <rdf:object rdf:resource="#Y" />
    <rdf:predicate
      rdf:resource="http://purl.org/dc/elements/1.1/title" />
  </qel:StatementLiteral>
</rdf:RDF>

```

Translation:

```

seq((SELECT VAR0,VAR1 FROM {VAR0} n0:title {VAR1})) Using
  Namespace n0 = &http://purl.org/dc/elements/1.1/

```

Example-Query5: All resources which title are not "Artificial Intelligence".

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE rdf:RDF (View Source for full doctype...)>
_ <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:qel="http://www.edutella.org/qel#" >
  _ <qel:Query rdf:about="#genQuery">
    <qel:literal rdf:resource="#st0" />
    <qel:literal rdf:resource="#st1" />
    _ <qel:resultVariables>
      _ <rdf:Seq>
        <rdf:_1 rdf:resource="#X" />
        <rdf:_2 rdf:resource="#Y" />
      </rdf:Seq>
    </qel:resultVariables>
  </qel:Query>
  <qel:Variable rdf:about="#X" />
  <qel:Variable rdf:about="#Y" />
  _ <qel:QueryLiteral rdf:about="#st0">
    <qel:predicate rdf:resource="http://www.edutella.org/qel#equal"
      />
    <qel:negated rdf:resource="http://www.edutella.org/qel#equal"
      />
  </qel:QueryLiteral>

```

```

_ <qel:arguments>
  _ <rdf:Seq>
    <rdf:_1 rdf:resource="#Y" />
    <rdf:_2>Artificial Intelligence</rdf:_2>
  </rdf:Seq>
</qel:arguments>
</qel:QueryLiteral>
_ <qel:StatementLiteral rdf:about="#st1">
  <rdf:subject rdf:resource="#X" />
  <rdf:predicate
    rdf:resource="http://purl.org/dc/elements/1.1/title" />
  <rdf:object rdf:resource="#Y" />
</qel:StatementLiteral>
</rdf:RDF>

```

### Translation:

```

seq((SELECT VAR1,VAR0 FROM {VAR1} n0:title {VAR0}WHERE
  !(VAR0 = "Artificial Intelligence"))) Using Namespace n0
= &http://purl.org/dc/elements/1.1/

```

Example-Query6: All X which are of the type book and have the title "Artificial Intelligence".

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE rdf:RDF (View Source for full doctype...)>
_ <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:qel="http://www.edutella.org/qel#"
  xmlns:lit="http://www.lit.qel/types#">
  _ <qel:Query rdf:about="#genQuery">
    <qel:literal rdf:resource="#st0" />
    <qel:literal rdf:resource="#st1" />
  _ <qel:resultVariables>
    _ <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
    </rdf:Seq>
  </qel:resultVariables>
</qel:Query>
<qel:Variable rdf:about="#X" />
_ <qel:StatementLiteral rdf:about="#st0">
  <rdf:object>Artificial Intelligence</rdf:object>
  <rdf:subject rdf:resource="#X" />
  <rdf:predicate
    rdf:resource="http://purl.org/dc/elements/1.1/title" />
</qel:StatementLiteral>
_ <qel:StatementLiteral rdf:about="#st1">
  <rdf:subject rdf:resource="#X" />
  <rdf:object rdf:resource="http://www.lit.qel/types#Book" />
  <rdf:predicate rdf:resource="http://www.w3.org/1999/02/22-
    rdf-syntax-ns#type" />

```

```

    </qel:StatementLiteral>
  </rdf:RDF>

```

## Translation:

```

seq((SELECT VAR0 FROM {VAR0} n0:type {R1} , {VAR0} n1:title
  {L1}WHERE R1 like "http://www.lit.qel/types#Book" and L1
  like "Artificial Intelligence")) Using Namespace n0 =
  &http://www.w3.org/1999/02/22-rdf-syntax-ns#, n1 =
  &http://purl.org/dc/elements/1.1/

```

Example-Query8: All resources which are of the type Aibook or of the type book and have the title "Artificial Intelligence".

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
  <!DOCTYPE rdf:RDF (View Source for full doctype...)>
  _ <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:qel="http://www.edutella.org/qel#"
    xmlns:lit="http://www.lit.qel/types#">
  _ <qel:Query rdf:ID="AI_Book_Query">
    <qel:rule rdf:resource="#r1" />
    <qel:rule rdf:resource="#r2" />
    <qel:literal rdf:resource="#l1" />
  _ <qel:resultVariables>
    _ <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
    </rdf:Seq>
    </qel:resultVariables>
  </qel:Query>
  <qel:Variable rdf:ID="X" />
  _ <qel:Rule rdf:about="#r1">
  _ <qel:head>
    _ <qel:QueryLiteral rdf:about="#h1">
      <qel:predicate rdf:resource="#aibook" />
    _ <qel:arguments>
      _ <rdf:Seq>
        <rdf:_1 rdf:resource="#X" />
      </rdf:Seq>
    </qel:arguments>
    </qel:QueryLiteral>
  </qel:head>
  _ <qel:literal>
    _ <qel:StatementLiteral rdf:ID="st2">
      <rdf:subject rdf:resource="#X" />
      <rdf:predicate
        rdf:resource="http://www.w3.org/1999/02/22-rdf-
        syntax-ns#type" />
      <rdf:object
        rdf:resource="http://www.lit.qel/types#Book" />
    </qel:StatementLiteral>
  </qel:literal>
  _ <qel:literal>

```

```

    _ <qel:StatementLiteral rdf:ID="st3">
      <rdf:subject rdf:resource="#X" />
      <rdf:predicate
        rdf:resource="http://purl.org/dc/elements/1.1/title"
        />
      <rdf:object>Artificial Intelligence</rdf:object>
    </qel:StatementLiteral>
  </qel:literal>
</qel:Rule>
_ <qel:Rule rdf:about="#r2">
  _ <qel:head>
    _ <qel:QueryLiteral rdf:about="#h2">
      <qel:predicate rdf:resource="#aibook" />
      _ <qel:arguments>
        _ <rdf:Seq>
          <rdf:_1 rdf:resource="#X" />
        </rdf:Seq>
      </qel:arguments>
    </qel:QueryLiteral>
  </qel:head>
  _ <qel:literal>
    _ <qel:StatementLiteral rdf:ID="st5">
      <rdf:subject rdf:resource="#X" />
      <rdf:predicate
        rdf:resource="http://www.w3.org/1999/02/22-rdf-
syntax-ns#type" />
      <rdf:object rdf:resource="http://www.lit.qel/types#AI-
Book" />
    </qel:StatementLiteral>
  </qel:literal>
</qel:Rule>
_ <qel:QueryLiteral rdf:ID="I1">
  <qel:predicate rdf:resource="#aibook" />
  _ <qel:arguments>
    _ <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
    </rdf:Seq>
  </qel:arguments>
</qel:QueryLiteral>
</rdf:RDF>

```

## Translation:

```

seq((SELECT VAR0 FROM {VAR0} n0:type {R1} , {VAR0} n1:title
{L1}WHERE R1 like "http://www.lit.qel/types#Book" and L1
like "Artificial Intelligence") , (SELECT VAR0 FROM
{VAR0} n0:type {R1} WHERE R1 like
"http://www.lit.qel/types#AI-Book")) Using Namespace n0 =
&http://www.w3.org/1999/02/22-rdf-syntax-ns#, n1 =
&http://purl.org/dc/elements/1.1/

```

Example-Query10: The same meaning like query 8 but expressed with 4 rules and some of them are nested. And it asks for all (X,Y,Z) tuples were

X is the Aibook.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
  <!DOCTYPE rdf:RDF (View Source for full doctype...)>
  _ <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:qel="http://www.edutella.org/qel#"
    xmlns:lit="http://www.lit.qel/types#">
    _ <qel:Query rdf:about="#genQuery">
      <qel:rule rdf:resource="#r0" />
      <qel:rule rdf:resource="#r1" />
      <qel:rule rdf:resource="#r2" />
      <qel:rule rdf:resource="#r3" />
      <qel:literal rdf:resource="#st0" />
      <qel:literal rdf:resource="#st1" />
      _ <qel:resultVariables>
        _ <rdf:Seq>
          <rdf:_1 rdf:resource="#X" />
          <rdf:_2 rdf:resource="#Y" />
          <rdf:_3 rdf:resource="#Z" />
        </rdf:Seq>
      </qel:resultVariables>
    </qel:Query>
    <qel:Variable rdf:about="#X" />
    <qel:Variable rdf:about="#Y" />
    <qel:Variable rdf:about="#Z" />
    _ <qel:Rule rdf:about="#r0">
      <qel:head rdf:resource="#st2" />
      <qel:literal rdf:resource="#st3" />
      <qel:literal rdf:resource="#st4" />
    </qel:Rule>
    _ <qel:Rule rdf:about="#r1">
      <qel:head rdf:resource="#st5" />
      <qel:literal rdf:resource="#st6" />
    </qel:Rule>
    _ <qel:Rule rdf:about="#r2">
      <qel:head rdf:resource="#st7" />
      <qel:literal rdf:resource="#st8" />
    </qel:Rule>
    _ <qel:Rule rdf:about="#r3">
      <qel:literal rdf:resource="#st10" />
      <qel:head rdf:resource="#st9" />
    </qel:Rule>
    _ <qel:StatementLiteral rdf:about="#st0">
      <rdf:subject rdf:resource="#X" />
      <rdf:predicate rdf:resource="#Y" />
      <rdf:object rdf:resource="#Z" />
    </qel:StatementLiteral>
    _ <qel:QueryLiteral rdf:about="#st1">
      <qel:predicate rdf:resource="#aibook" />
      _ <qel:arguments>
        _ <rdf:Seq>
          <rdf:_1 rdf:resource="#X" />
```

```

    </rdf:Seq>
  </qel:arguments>
</qel:QueryLiteral>
_ <qel:StatementLiteral rdf:about="#st10">
  <rdf:subject rdf:resource="#X" />
  <rdf:object rdf:resource="#Y" />
  <rdf:predicate
    rdf:resource="http://purl.org/dc/elements/1.1/title" />
</qel:StatementLiteral>
_ <qel:QueryLiteral rdf:about="#st2">
  <qel:predicate rdf:resource="#aibook" />
  <qel:arguments>
  _ <rdf:Seq>
    <rdf:_1 rdf:resource="#X" />
  </rdf:Seq>
</qel:arguments>
</qel:QueryLiteral>
_ <qel:QueryLiteral rdf:about="#st3">
  <qel:arguments>
  _ <rdf:Seq>
    <rdf:_1 rdf:resource="#X" />
    <rdf:_2 rdf:resource="http://www.lit.qel/types#Book"
      />
  </rdf:Seq>
</qel:arguments>
  <qel:predicate rdf:resource="#type" />
</qel:QueryLiteral>
_ <qel:QueryLiteral rdf:about="#st4">
  <qel:arguments>
  _ <rdf:Seq rdf:_2="Artificial Intelligence">
    <rdf:_1 rdf:resource="#X" />
  </rdf:Seq>
</qel:arguments>
  <qel:predicate rdf:resource="#title" />
</qel:QueryLiteral>
_ <qel:QueryLiteral rdf:about="#st5">
  <qel:predicate rdf:resource="#aibook" />
  <qel:arguments>
  _ <rdf:Seq>
    <rdf:_1 rdf:resource="#X" />
  </rdf:Seq>
</qel:arguments>
</qel:QueryLiteral>
_ <qel:QueryLiteral rdf:about="#st6">
  <qel:arguments>
  _ <rdf:Seq>
    <rdf:_1 rdf:resource="#X" />
    <rdf:_2 rdf:resource="http://www.lit.qel/types#AI-
      Book" />
  </rdf:Seq>
</qel:arguments>
  <qel:predicate rdf:resource="#type" />
</qel:QueryLiteral>
_ <qel:QueryLiteral rdf:about="#st7">

```

```

_ <qel:arguments>
  _ <rdf:Seq>
    <rdf:_1 rdf:resource="#X" />
    <rdf:_2 rdf:resource="#Y" />
  </rdf:Seq>
</qel:arguments>
<qel:predicate rdf:resource="#type" />
</qel:QueryLiteral>
_ <qel:StatementLiteral rdf:about="#st8">
  <rdf:subject rdf:resource="#X" />
  <rdf:object rdf:resource="#Y" />
  <rdf:predicate rdf:resource="http://www.w3.org/1999/02/22-
    rdf-syntax-ns#type" />
</qel:StatementLiteral>
_ <qel:QueryLiteral rdf:about="#st9">
  _ <qel:arguments>
    _ <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
      <rdf:_2 rdf:resource="#Y" />
    </rdf:Seq>
  </qel:arguments>
  <qel:predicate rdf:resource="#title" />
</qel:QueryLiteral>
</rdf:RDF>

```

## Translation:

```

seq((SELECT VAR0,@VAR1,VAR2 FROM {VAR0} @VAR1 {VAR2}, {VAR0}
  n0:title {L1}, {VAR0} n1:type {R1} WHERE L1 like
  "Artificial Intelligence" and R1 like
  "http://www.lit.qel/types#Book") , (SELECT
  VAR0,@VAR1,VAR2 FROM {VAR0} @VAR1 {VAR2}, {VAR0} n1:type
  {R1} WHERE R1 like "http://www.lit.qel/types#AI-Book"))
Using Namespace n0 = &http://purl.org/dc/elements/1.1/,
  n1 = &http://www.w3.org/1999/02/22-rdf-syntax-ns#

```

Example-Query14: All resources which have a title and the title and three outerjoin information: creator, identifier and language.

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
  <!DOCTYPE rdf:RDF (View Source for full doctype...)>
  _ <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:qel="http://www.edutella.org/qel#" >
    _ <qel:Query rdf:about="#genQuery">
      <qel:literal rdf:resource="#st0" />
      <qel:outerJoinLiteral rdf:resource="#st1" />
      <qel:outerJoinLiteral rdf:resource="#st2" />
      <qel:outerJoinLiteral rdf:resource="#st3" />
    _ <qel:resultVariables>
      _ <rdf:Seq>
        <rdf:_1 rdf:resource="#X" />
        <rdf:_2 rdf:resource="#Y" />

```

```

    <rdf:_3 rdf:resource="#Z" />
    <rdf:_4 rdf:resource="#S" />
    <rdf:_5 rdf:resource="#T" />
  </rdf:Seq>
</qel:resultVariables>
</qel:Query>
<qel:Variable rdf:about="#X" />
<qel:Variable rdf:about="#Y" />
<qel:Variable rdf:about="#Z" />
<qel:Variable rdf:about="#S" />
<qel:Variable rdf:about="#T" />
_ <qel:StatementLiteral rdf:about="#st0">
  <rdf:subject rdf:resource="#X" />
  <rdf:object rdf:resource="#Y" />
  <rdf:predicate
    rdf:resource="http://purl.org/dc/elements/1.1/title" />
</qel:StatementLiteral>
_ <qel:StatementLiteral rdf:about="#st1">
  <rdf:subject rdf:resource="#X" />
  <rdf:object rdf:resource="#Z" />
  <rdf:predicate
    rdf:resource="http://purl.org/dc/elements/1.1/creator" />
</qel:StatementLiteral>
_ <qel:StatementLiteral rdf:about="#st2">
  <rdf:subject rdf:resource="#X" />
  <rdf:object rdf:resource="#S" />
  <rdf:predicate
    rdf:resource="http://purl.org/dc/elements/1.1/identifier" />
</qel:StatementLiteral>
_ <qel:StatementLiteral rdf:about="#st3">
  <rdf:subject rdf:resource="#X" />
  <rdf:object rdf:resource="#T" />
  <rdf:predicate
    rdf:resource="http://purl.org/dc/elements/1.1/language" />
</qel:StatementLiteral>
</rdf:RDF>

```

Translation:

```

seq((SELECT VAR2,VAR1, (SELECT VAR4 FROM {VAR2} n0:creator
  {VAR4}), (SELECT VAR3 FROM {VAR2} n0:language {VAR3}),
  (SELECT VAR0 FROM {VAR2} n0:identifier {VAR0}) FROM
  {VAR2} n0:title {VAR1})) Using Namespace n0 =
  &http://purl.org/dc/elements/1.1/

```

Example-Query11: All AIBooks which does not have the string software in its title and the (X, Y,Z) tuples with the AIBook as subject.

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE rdf:RDF (View Source for full doctype...)>
_ <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"

```

```

xmlns:qel="http://www.edutella.org/qel#"
xmlns:lit="http://www.lit.qel/types#">
- <qel:Query rdf:about="#genQuery">
  <qel:rule rdf:resource="#r0" />
  <qel:rule rdf:resource="#r1" />
  <qel:rule rdf:resource="#r2" />
  <qel:rule rdf:resource="#r3" />
  <qel:literal rdf:resource="#st0" />
  <qel:literal rdf:resource="#st1" />
  <qel:literal rdf:resource="#st11" />
  - <qel:resultVariables>
    - <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
      <rdf:_2 rdf:resource="#Y" />
      <rdf:_3 rdf:resource="#Z" />
    </rdf:Seq>
  </qel:resultVariables>
</qel:Query>
<qel:Variable rdf:about="#X" />
<qel:Variable rdf:about="#Y" />
<qel:Variable rdf:about="#Z" />
- <qel:Rule rdf:about="#r0">
  <qel:head rdf:resource="#st2" />
  <qel:literal rdf:resource="#st3" />
  <qel:literal rdf:resource="#st4" />
</qel:Rule>
- <qel:Rule rdf:about="#r1">
  <qel:head rdf:resource="#st5" />
  <qel:literal rdf:resource="#st6" />
</qel:Rule>
- <qel:Rule rdf:about="#r2">
  <qel:head rdf:resource="#st7" />
  <qel:literal rdf:resource="#st8" />
</qel:Rule>
- <qel:Rule rdf:about="#r3">
  <qel:literal rdf:resource="#st10" />
  <qel:head rdf:resource="#st9" />
</qel:Rule>
- <qel:StatementLiteral rdf:about="#st0">
  <rdf:subject rdf:resource="#X" />
  <rdf:predicate rdf:resource="#Y" />
  <rdf:object rdf:resource="#Z" />
</qel:StatementLiteral>
- <qel:QueryLiteral rdf:about="#st1">
  <qel:predicate rdf:resource="#aibook" />
  - <qel:arguments>
    - <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
    </rdf:Seq>
  </qel:arguments>
</qel:QueryLiteral>
- <qel:QueryLiteral rdf:about="#st11">
  <qel:predicate rdf:resource="http://www.edutella.org/qel#like"
  />

```

```

    <qel:negated rdf:resource="http://www.edutella.org/qel#like" />
  <qel:arguments>
    <rdf:Seq>
      <rdf:_1 rdf:resource="#Z" />
      <rdf:_2>%Software%</rdf:_2>
    </rdf:Seq>
  </qel:arguments>
</qel:QueryLiteral>
<qel:StatementLiteral rdf:about="#st10">
  <rdf:subject rdf:resource="#X" />
  <rdf:object rdf:resource="#Y" />
  <rdf:predicate
    rdf:resource="http://purl.org/dc/elements/1.1/title" />
</qel:StatementLiteral>
<qel:QueryLiteral rdf:about="#st2">
  <qel:predicate rdf:resource="#aibook" />
  <qel:arguments>
    <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
    </rdf:Seq>
  </qel:arguments>
</qel:QueryLiteral>
<qel:QueryLiteral rdf:about="#st3">
  <qel:arguments>
    <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
      <rdf:_2 rdf:resource="http://www.lit.qel/types#Book"
        />
    </rdf:Seq>
  </qel:arguments>
  <qel:predicate rdf:resource="#type" />
</qel:QueryLiteral>
<qel:QueryLiteral rdf:about="#st4">
  <qel:arguments>
    <rdf:Seq rdf:_2="Artificial Intelligence">
      <rdf:_1 rdf:resource="#X" />
    </rdf:Seq>
  </qel:arguments>
  <qel:predicate rdf:resource="#title" />
</qel:QueryLiteral>
<qel:QueryLiteral rdf:about="#st5">
  <qel:predicate rdf:resource="#aibook" />
  <qel:arguments>
    <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
    </rdf:Seq>
  </qel:arguments>
</qel:QueryLiteral>
<qel:QueryLiteral rdf:about="#st6">
  <qel:arguments>
    <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
      <rdf:_2 rdf:resource="http://www.lit.qel/types#AI-
        Book" />
    </rdf:Seq>
  </qel:arguments>

```

```

        </rdf:Seq>
    </qel:arguments>
    <qel:predicate rdf:resource="#type" />
</qel:QueryLiteral>
_ <qel:QueryLiteral rdf:about="#st7">
  _ <qel:arguments>
    _ <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
      <rdf:_2 rdf:resource="#Y" />
    </rdf:Seq>
  </qel:arguments>
  <qel:predicate rdf:resource="#type" />
</qel:QueryLiteral>
_ <qel:StatementLiteral rdf:about="#st8">
  <rdf:subject rdf:resource="#X" />
  <rdf:object rdf:resource="#Y" />
  <rdf:predicate rdf:resource="http://www.w3.org/1999/02/22-
    rdf-syntax-ns#type" />
</qel:StatementLiteral>
_ <qel:QueryLiteral rdf:about="#st9">
  _ <qel:arguments>
    _ <rdf:Seq>
      <rdf:_1 rdf:resource="#X" />
      <rdf:_2 rdf:resource="#Y" />
    </rdf:Seq>
  </qel:arguments>
  <qel:predicate rdf:resource="#title" />
</qel:QueryLiteral>
</rdf:RDF>

```

Translation : See also chapter RQL – Bug

```

seq((SELECT VAR0,@VAR2,VAR1 FROM {VAR0} @VAR2 {VAR1}, {VAR0}
n0:type {R1} WHERE R1 like "http://www.lit.gel/types#AI-
Book" and !(VAR1 like "*Software*")), (SELECT
VAR0,@VAR2,VAR1 FROM {VAR0} @VAR2 {VAR1}, {VAR0}
n1:title {L1}, {VAR0} n0:type {R1} WHERE L1 like
"Artificial Intelligence" and R1 like
"http://www.lit.gel/types#Book" and !(VAR1 like
"*Software*")) Using Namespace n0 =
&http://www.w3.org/1999/02/22-rdf-syntax-ns#, n1 =
&http://purl.org/dc/elements/1.1/

```

## Appendix – Rssdb Configuration

The configuration in the Rssdb window is a little bit difficult. First you have to select the appropriate input file. The order of the files is very important. At first you have to load the metaschema then the schemas then the data. The schemas are maybe connected to each other and you have to load them in a specific order. Then you have to choose a database name from an existing postgresSQL database. The default port number of postgresSQL

is 5432. And you have to enter the host where the data base system is running. Use the appropriate button: store RDF or store thesaurus for storing your file. But important is to choose the right option. Use the second option which is "Schema Specific Without SubTable Representation". The default option is always "Schema Specific Representation" after you have started Rssdb. But you cannot read the data if you store it with this option unless you have installed your RQL Interpreter with the option: "--enable-pgisa". The default installation without option can only read the data stored without subtables. There is a third option please see for details:

<http://athena.ics.forth.gr:9090/RDF/RQL/Install.html#extra>

## References

- [1] Edutella Project.  
<http://edutella.jxta.org/>
  
- [2] QEL Introduction.  
<http://edutella.jxta.org/spec/qel.html>
  
- [3] RDF. <http://www.w3c.org/rdf/>
  
- [4] Boris Wolf. Diplomarbeit: Peer-to-Peer Networking for Distributed Learning Repositories  
<http://www.kbs.uni-hannover.de/Arbeiten/Diplomarbeiten/02/Edutella.pdf>
  
- [5] RQL Online Demo.  
<http://139.91.183.30:8999/RQLdemo/>
  
- [6] ICS Forth. The RDF Schema Specific DataBase (RSSDB)  
<http://athena.ics.forth.gr:9090/RDF/RSSDB/index.html>
  
- [7] ICS Forth. RDF Query Language (RQL)  
<http://athena.ics.forth.gr:9090/RDF/RQL/index.html>

- [8] PADLR. Personalized Access to Distributed Learning Repositories.  
[http://sll.stanford.edu/communications/news/wgln\\_projects\\_april\\_01/padlr\\_17.pdf](http://sll.stanford.edu/communications/news/wgln_projects_april_01/padlr_17.pdf)
  
- [9] Eclipse. Java Development.  
<http://www.eclipse.org/>
  
- [10] Jena. Semantic Web Toolkit.  
<http://www.hpl.hp.com/semweb/jena>
  
- [11] PostGreSQL JDBC Driver.  
<http://jdbc.postgresql.org/>
  
- [12] PostgreSQL. Open Source Data Base.  
<http://www.postgresql.org/>
  
- [13] Database Systems: The Complete Book  
by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author),  
[Jennifer D. Widom](#) (Author)