easy RDF Query Language (eRQL) v1.8 Tutorial

Karsten Tolle
Institut für Informatik, Datenbanken und Informationssysteme (DBIS)
Johann Wolfgang Goethe-Universität Frankfurt am Main
D-60325 Frankfurt am Main
tolle@dbis.cs.uni-frankfurt.de

Date: 17th March 2005

Abstract: This document is a tutorial for the easy RDF Query Language (eRQL). eRQL is designed to serve as an end user query language for RDF data (including schema information). It is including support for internal and external context. To support external context, like source information, it is build on an RDF data model that is extended by an context node for each RDF triple. An implementation for eRQL is provided by the eRqlEngine that is included to the RDF Source related Storage System (RDF-S3), that provides the needed context node extension of the RDF data model.

1 Introduction

The main goal of easy RDF Query Language (eRQL) is to be simple enough to be used without any knowledge of the underlying ontology used to describe the data. Also the query syntax itself should be intuitive. This goal is reached by being close to the syntax of Google. By simply entering keywords that can be combined by AND or OR, queries can be performed.

In addition and in contrast to other RDF query languages, eRQL provides additional support for context information, by providing different modes. Context information is here mend to be internal context represented by the surrounding graph influencing a single or set of triples (POI Mode), but also external context like source (provenance) information (Document Mode). To deal with external context eRQL is working on quadruples rather than on triples. It is important that this is provided by the underlying data structure.

In addition to simple keywords, eRQL also supports functions to get a quick overview of the underlying ontology. Most of these are similar to the functions provided by RQL. Functions are further described in 2.3.
The result structure of an eRQL query differs depending on the query. In section 3 these differences are discussed. The appendix contains a short description of the eRQL syntax.

Currently the eRqlEngine for RDF-S3 is an implementation for eRQL providing its full functionality. The eRqlEngine for RDF-S3 comes with a graphical user interface as described further in the appendix. It includes the settings for the database connection and also the mapping from source URLs to the internal used IDs to abbreviate them in the document mode. RDF-S3 including the eRqlEngine is 100% Java and can be downloaded as open source under GNU license1.

2 eRQL by examples

eRQL provides one-word-queries, modes, functions and Boolean operators. These enables users to query resources and schema descriptions stored in the underlying repository.

2.1 One-Word-Queries and Modes in eRQL

For this subsection we assume the data shown in Figure 1 to be stored into the underlying repository. The different colors of the nodes and edges represent the corresponding context nodes showing the different sources these triples belong to. As can be seen, the resource &r3 is used in both sources and is illustrated therefore in blue.

One-word-queries of eRQL are similar queries of existing search engines a la Google. This enables users, even without any knowledge of the query language or the data structure, to enter and run eRQL queries. As a result for these queries you would get those triples in return, that contain the given word as either subject, predicate or object.

The matching triples are called direct hits. One peculiarity of eRQL is to return not only the triples fitting the request, but also those surrounding them. This is the so called Point of Interest Mode (POI Mode), whereby the distance of the surrounding triples that are included to the result for an direct hit can be defined within the query. This way eRQL includes internal context information to the result and enables a better understanding of it. The POI Mode with a distance of 1 is the default mode, which means that the one-word-query Pablo

---

1 Online at: http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/RDFS3/index.html
will return the graph shown in Figure 2. Since eRQL is working on quadruples rather than on triples, the full result will also contain the context nodes for each triple illustrated in the graph by the color.

<table>
<thead>
<tr>
<th>eRQL Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pablo</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>Picasso</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**Figure 2** Showing the one-word-query *Pablo* and its result. The direct hit for the query is highlighted.

To disable the default POI Mode and do not include surrounding triples to the result, one-word-queries can be enclosed by brackets `one-word-query`. The result then only contains the direct hits (called *Statement Mode*). For the query `[Pablo]` the result is shown in Figure 3.

<table>
<thead>
<tr>
<th>eRQL Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Pablo]</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**Figure 3** One-word-query in Statement Mode: [Pablo].

To increase the distance for the surrounding graph that should be included, the query is either cumulative enclosed by braces `{ }`. The number of braces is representing the distance, meaning one braces pair results in a distance of one (equal to the default), two pairs in a distance of two and so on. Instead of braces also leading ~ signs can be used. The ~ does not affect the default mode, meaning encounter the distance for a one-word-query using ~, just count them and add 1. The queries `{{Pablo}}` and `~Pablo` are therefore equal. The result for it is shown in Figure 3.

<table>
<thead>
<tr>
<th>eRQL Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>~Pablo</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>~</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**Figure 4** One-word-query in POI Mode with a distance of two.

**Restricting one-word-queries**

One-word-queries compare the given string with URI references or literal values. This is a very general search. eRQL therefore provides various restrictions to limit one-word-queries.

- By putting a one-word-query into quotation marks, the request is restricted to literal values only. This way also blanks can be used. The eRQL query “*Reina Sofia Museum*” would be valid and result in the graph shown in Figure 5.
There is also the other way round possible, to restrict a one-word-query on URI references only. This is done by writing `res(one-word-query)`. Of course these restrictions can be combined with the Statement Mode. The query `[res(woman)]` would result in two triples plus context nodes as shown in Figure 6. In the default POI Mode these triples would be extended with the corresponding surrounding graph.

It is further possible to restrict the one-word-query on the single segments of a triple, by writing `subject(one-word-query)`, `predicate(one-word-query)` or `object(one-word-query)`. Note: All three of these restrictions are restricted on URI references. Literals without URI reference can only appear as an object and can be queried by using quotation marks as described above. To search for objects independent of being a literal or a URI reference a Boolean OR connection should be used.

### 2.2 Document Mode

With the Document Mode one can restrict the result to certain sources that are stored in the repository. This restriction is for finding direct hit and for selecting the surrounding graph for the POI Mode. The syntax for the Document Mode is: `<query; source_list; restrict>`. The `query` can be any valid eRQL query, the `source_list` is a list of source URLs separated by blanks and/or comma and `restrict` is either 0 to leave out the defined sources or 1 to restrict the query to them. As an abbreviation the sources in the `source_list` can also be identified by internal IDs, that must be retrieved from the database before.
Figure 8  Examples demonstrating the Document Mode.

Let’s assume the internal ID for the source http://www.museum.es is 1 and the internal ID for the source http://www.culture.net is 2. The first examples shown in Figure 8 can be rewritten as:

\[
<\text{guernica}; \text{http://www.museum.es}; 1> \equiv <\text{guernica}; 1; 1>
\]

Since the given example repository given here only contains two sources, we can also write:

\[
<\text{guernica}; \text{http://www.museum.es}; 1> \equiv <\text{guernica}; \text{http://www.culture.net}; 0> \equiv <\text{guernica}; 2; 0>
\]

2.3 Schema Functions in eRQL

eRQL also supports schema functions to get a quick overview of the underlying ontology. Most of these are similar to the functions provided by RQL\(^2\). To demonstrate these functions, we assume for the examples of this subsection the graph shown in Figure 9 should be contained in the repository. This means there are four sources stored in the repository (we assume that one namespace corresponds to one source with the given namespace URL as source URL).

---

\(^2\) RDF Query Language (RQL), part of the ICS-FORTH RDFSuite, see online at: http://athena.ics.forth.gr:9090/RDF/
The functions are split into general schema functions that do not need an input and schema functions on resources that will need an eRQL query as input parameter. The schema functions are always executed with the Statement Mode, meaning there is no surrounding graph included to the result. The Document Mode is functioning with schema functions.

### 2.3.1 General Schema Functions

- **classes()** – To list all classes. This does not include data types like strings or integers for typed literals.

```
<classes>(); http://www.oclc.org/schema2.rdf1
```

---

**Figure 9** The underlying content of the repository for the examples of this subsection.

**Figure 10** Example for the `classes()` function.
• properties() – To list all properties.

<table>
<thead>
<tr>
<th>eRQL Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;properties(); http://www.oclc.org/schema2.rdf;1&gt;</code></td>
<td>last_modified, title, mime_type, file_size</td>
</tr>
<tr>
<td><code>&lt;properties(); http://www.culture.net/rodin.rdf;1&gt;</code></td>
<td>last_modified, mime_type, title, frame, exhibited, paints, Iname</td>
</tr>
</tbody>
</table>

**Figure 11** Example for the `properties()` function.

• container() – To list all containers including their members.

• literals() – To list all literals.

• reifiedStatements() – To list all reified statements, including the statements they are standing for, meaning the `rdf:subject`, `rdf:predicate` and `rdf:object` links.

• triples() – To list all triples. The result returned here represents a graph equal to the graphs returned by one-word-queries.

<table>
<thead>
<tr>
<th>eRQL Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;triples(); http://www.culture.net/rodin.rdf;1&gt;</code></td>
<td>Rodin Museum, title, fframe, &amp;f5, &amp;f6, exhibited, &amp;f7, Rodin, Image/jpeg, 2000-02-01</td>
</tr>
</tbody>
</table>

**Figure 12** Example for the `triples()` function.

### 2.3.2 Schema Functions on Resources

Schema functions on resources need as an input parameter a one-word-query. Currently there are schema functions on classes and schema functions on properties only. First the corresponding classes or properties are requested from the system. Then the according function is evaluated on this set of classes or properties. The result for schema functions on resources are separated by the fitting classes or properties (more information about the result structure can be found in section 3).

Note: Classes and properties always have a namespace URL and a name. For schema functions on resources the one-word-query needs to match with the name part. In case the given parameter starts with a protocol definition like `http://`, the parameter will be handled as a URI and only the given URI is selected as the explored class or property.
Schema Functions on classes

- **instancesOf(URI | one-word-query)** – returns all instances of the fitting classes. It includes the instances of all sub classes.

  ![Figure 13](image)

  **Figure 13** To retrieve the instances of a class or a set of classes. Here two class names fit the given pattern ‘Arti’.

- **directInstancesOf(URI | one-word-query)** – returns all direct instances of the fitting classes. Direct instances means only those resources that are entered with a `rdf:type` link to the class. Instances of subclasses without a direct `rdf:type` link will not be returned.

  ![Figure 14](image)

  **Figure 14** Direct instances do not include instances of sub classes. Both classes are therefore returned with an empty set.

- **subClassOf(URI | one-word-query)** – returns all classes that are defined to be sub classes (defined by `rdfs:subClassOf`) of the fitting classes.

  ![Figure 15](image)

  **Figure 15** To retrieve the sub classes of classes, here combined with Document Mode using internal source ID to abbreviate the source URL.

- **superClassOf(URI | one-word-query)** – returns all classes that are defined to be super classes (defined by `rdfs:subClassOf`, interpreted in inverse direction) of the fitting classes.

  ![Figure 16](image)

  **Figure 16** Finding the super classes of a class or a set of classes.
Schema Functions on properties

- **domain(URI | one-word-query)** – returns the domain classes (defined by `rdfs:domain`) for the fitting properties.

  ```
  eRQL Query | Result
  ------------|-----------
  domain(http://www.icim.com/schema1.rdf#creates) | [creates : Arist!]
  
  Figure 17  Finding the domain classes for a specific property.
  ```

- **range(URI | one-word-query)** – returns the range classes (defined by `rdfs:range`) for the fitting properties.

  ```
  eRQL Query | Result
  ------------|-----------
  range(http://www.icim.com/schema1.rdf#creates) | [creates : Artifact]
  
  Figure 18  Finding the range classes for a specific property.
  ```

- **subPropertyOf(URI | one-word-query)** – returns the sub properties (defined by `rdfs:subPropertyOf`) for the fitting properties.

  ```
  eRQL Query | Result
  ------------|-----------
  subPropertyOf(creates) | [creates : sculps, paints]
  
  Figure 19  Finding sub properties for a property or a set of properties. Here only one property fits the given parameter.
  ```

- **superPropertyOf(URI | one-word-query)** – returns the super properties (defined by `rdfs:subPropertyOf`, interpreted in inverse direction) for the fitting properties.

  ```
  eRQL Query | Result
  ------------|-----------
  superPropertyOf(paints) | [paints : creates]
  
  Figure 20  Finding the super properties for a property or a set of properties.
  ```

3  Result structure

One doctrine of eRQL is to be fuzzy, to allow users that do not know 100% what they are searching for, to be able to find useful results. The result of an eRQL query will always be a set containing possible answers to it. There exist three types of these sub-results in eRQL. We can find:

- **POIResult** – for example for one-word-queries. A set of graphs containing fitting triples plus their context nodes. In the examples the context nodes are represented by the different color of the triples themselves.

- **SchemaResultList** – for example with the general schema queries properties() or classes(). These result type just represents a set of resources or literals. In some cases the context nodes will not be returned to avoid to many duplicate resources with just different context
nodes. To force the result to contain the context nodes the Document Mode can be used. To avoid a restriction to certain sources one can use `<query;0;0>` in this case.

- **SchemaResultMap** – for schema functions on resources, like `superPropertyOf(paints)`. In this case the result contains a list (keys) of fitting resources with the executed function for them as their value.

### 3.1 Interpreting the Boolean operators AND and OR

There is no restriction or limitation on queries in eRQL that can be combined using a disjunction (OR) or a conjunction (AND). The disjunction of eRQL results is defined to be the union of the result sets. As an effect, we can construct results containing different sub-result types.

The conjunction of eRQL results is more complex. We here define the interpretation for eRQL. Note that this is a decision make by us, there could other interpretations possible. The conjunction of two results is interpreted to be the result of the various conjunctions of their sub-results. For the sub-results we can find the following cases:

- **SchemaResultList AND SchemaResultList** – An example would be `<classes();1;1> AND <classes();2;1>`. This can be interpreted as: Show me all classes that are contained in the source with the internal ID 1 and contained in the source with the internal ID 2. The result of the conjunction therefore should be the intersection of the two sets. Note that for building the intersection the context node is ignored, otherwise most intersections would be empty.

- **SchemaResultMap AND SchemaResultList** – The query: `superClassOf(Sculp) AND <classes();6;1>` would be an example. It should be interpreted as: All super classes of classes having the pattern ‘Sculp’ in their name that are used inside the source with the internal ID of 6. The result therefore is a SchemaResultMap containing the same keys as the SchemaResultMap of the conjunction. The corresponding values are the intersection of the SchemaResultList and the corresponding value of the input SchemaResultMap.

- **SchemaResultMap AND SchemaResultMap** – An example would be the query: `subClassOf(Artist) AND subClassOf(Pers)`. The example would be interpreted as: Give me all sub classes of classes containing the pattern ‘Artist’ and being sub class of a class containing the pattern ‘Pers’. The result would be a SchemaResultMap. The key entries are all combinations of the two key sets. The corresponding value is the intersection of the two values.

- **SchemaResultList AND POIResult** – An example would be: `properties() AND ~~“Pablo”`. This would be interpreted as: Show me all properties, that are contained in graphs with a distance of max. 3 around literals containing the string “Pablo” inside their value. Such queries are resulting in a SchemaResultMap. For each graph of the POIResult an key entry will be entered in the SchemaResultMap. The value for the entries will be calculated by the intersection of the SchemaResultList and the set of the graph elements (subjects, predicates and objects). This value corresponds to the execution of the query for the list result (here properties()) on the given graph.

- **SchemaResultMap AND POIResult** – An example would be: `superClassOf(Sculp) AND “Pablo”`. This would be interpreted as: Retrieve the super classes of classes having the pattern ‘Sculp’ contained in their name and that are used near by triples containing the literal Pablo. The result of the conjunction would be a SchemaResultMap. It can be thought of executing the query for the SchemaResultMap on the single graphs of the
POIResult. The final result is generated by the union of the single resulting SchemaResultMap, whereby the keys are extended to track back the corresponding graph.

- **POIResult AND POIResult** – An example would be: Pablo AND Rodin. In the cases before, there were always an intersection defined and this intersection was returned as the result of the conjunction. Returning just the intersection of graphs would not satisfy a user since he would not understand the relation of the results and the hits that might not be contained in the intersection. Therefore, we here decided to return as an result the union of the two graphs in case there is an intersection. This means the example would be interpreted as: *In case there is an intersection of graphs containing Pablo and graphs containing Rodin, show me the union of them.* As a more abstract interpretation one also can say: *In case there is a path between triples fitting the query Pablo and triples fitting the query Rodin, show me the union.* This provides an easy way to search for possible and shortest paths between resources by slowly increasing the POI distance for queries.

Two graphs in our definition have an intersection in case there is a quadruple q that is contained in both graphs. This means an overlapping in just one resource or an triple is not sufficient.

So far, we looked at the conjunction of two results. In case of more results joining the conjunction, the single conjunctions can be treated one after the other. The only exception would be the conjunction of graphs results. Executing the conjunction pair wise, the order would have an impact on the result, as demonstrated in Figure 21. The conjunction of more than two POIResults will return therefore the union of the single graphs in case there is a path connection all participating graphs.

![Diagram showing POI-A, POI-B, and POI-C with direct hits](image)

POI-A AND POI-B = POI-A \( \cup \) POI-B
POI-B AND POI-C = POI-B \( \cup \) POI-C
POI-A AND POI-C = \{\}
(POI-A AND POI-C) AND POI-B = \{\}
(POI-A AND POI-B) AND POI-C = (POI-A \( \cup \) POI-B) \( \cup \) POI-C
POI-A AND (POI-B AND POI-C) = POI-A \( \cup \) (POI-B \( \cup \) POI-C)

**Figure 21** Demonstrating the importance of the order in case of conjunction for graphs results.

### 4 Additional Settings

There are some settings in eRQL that influence the whole query. These settings are:

- Case Sensitive – with this setting activated the given one-word-queries are executed case sensitive.
- Allow wildcards * and ? usage inside source URLs – with this setting activated one can use inside the *source_list* of queries using the Document Mode the wildcards * and ?. For
example the query: `<guernica; http://www.museum.* ; I>`, would execute the query `guernica` on all sources starting with ‘http://www.museum.’.

- Follow double property usage – in some cases using the POI Mode the single graphs of the result are getting too big. This for example is often the case if classes are contained inside the graph. Building the surrounding graph results in listening all direct instances of the class (as shown in Figure 22). To avoid this behavior, a double usage of the same property will not be included to the result. If the user wants to include them, this setting must be activated.

![Diagram](image)

**Figure 22** Example showing the effect of the setting: Follow double property usage. The Query Pablo results in the green graph. By increasing the POI distance using ~Pablo, also the blue part is included to the result. Is the setting Follow double property usage activated, in addition also the red triples are included.

5 Appendix

**eRQL syntax**

```
+--------+        +--------+
| Query  |        | Disjunction |
+--------+        +--------+
         |        | Conjunction |
         |        +--------+
         |        | Subquery   |
```

```
+--------+        +--------+
| Disjunction |        | OR |
+--------+        +--------+
         +--------+        +--------+
         | OR |
```

```
+--------+        +--------+
| Conjunction |        | AND |
+--------+        +--------+
         +--------+        +--------+
         | AND |
```
oneWord ::=  \[^ ;()\[\]{}<>~"	
\]+ 

ClassesFunction ::= "classes()" | "c()" | "C()"
ContainerFunction ::= "container()" | "con()" | "CON()"
LiteralsFunction ::= "literals()" | "l()" | "L()"
PropertiesFunction ::= "properties()" | "p()" | "P()"
ReifiedStatementsFunction ::= "reifiedStatements()" | "rs()" | "RS()"
TriplesFunction ::= "triples()" | "t()" | "T()"
DirectInstancesOfFunction ::= "directInstancesOf()" | "di()" | "dI()"
InstancesOfFunction ::= "instancesOf()" | "i()" | "I()"
SubClassOfFunction ::= "subClassOf()" | "subc()" | "subC()"
SuperClassOfFunction ::= "superClassOf()" | "superc()" | "superC()"

DomainFunction ::= "domain()" | "d()" | "D()"
RangeFunction ::= "range()" | "r()" | "R()"
SubPropertyOfFunction ::= "subPropertyOf()" | "subp()" | "subP()"
SuperPropertyOfFunction ::= "superPropertyOf()" | "superp()" | "superP()"

**The GUI of the eRqlEngine for RDF-S3**

Below you see the graphical user interface of eRqlEngine for RDF-S3. It can be used to retrieve the data stored by RDF-S3. A short description of the fields, checkboxes and buttons can be found underneath.
The eRQL query can be entered here. To find more information about how to write eRQL queries and their syntax please see eRQLModi.

The stored sources and their IDs

For eRQL document queries the internal IDs of the sources are used to abbreviate them. The IDs can be retrieved from the given table that can be actualized by pressing the refresh button under the table.

Case Sensitive

This influences the eRQL query to distinguish between upper and lower case. If selected the queries "picasso" and "Picasso" will return different results.

Follow double property usage

Important for the POI modus of eRQL queries. It will result in a smaller number of surrounding triples per hit by ignoring a double usage of the same property. In case a resource r is an instance of a class c (r rdf:type c), and the defined distance for the POI query goes further this triple, all instances i of the class c (i rdf:type c) would be included to the POI result. In case the checkbox is not selected these double usage of a property (in this case rdf:type) will not be included to the result set.
The query result starts with the time needed to execute the query followed by the number of direct hits. Depending on the distance that was specified with the query, each hit is shown with its surrounding triples separately. Each of these RDF graphs start with their size (number of triples), followed by the triples themselves. The first triple is always the hit itself. The triples are written in the form subject, predicate, object, followed by the source of the triple. For abbreviation the namespaces and source URLs are substituted. At the end of the query result the long versions can be found.

**Buttons**

**Execute! (ALT-ENTER)**

By pressing this button the entered eRQL query will be executed.

**Close**

Closes the program. The last used eRQL query and the DB Connection settings will be stored in the preferences. The next time you start the program these settings will be recreated.