Three Implementations of SquishQL, a Simple RDF Query Language

Libby Miller[1], Andy Seaborne[2], and Alberto Reggiori[3]

[1] ILRT, Bristol University, UK
libby.miller@bristol.ac.uk
[2] Hewlett-Packard Laboratories, Bristol, UK
andy_seaborne@hp.com
[3] Web Weaving Internet Engineering, Arnhem, The Netherlands
areggiori@webweaving.org

Abstract. RDF provides a basic way to represent data for the Semantic Web. We have been experimenting with the query paradigm for working with RDF data in semantic web applications. Query of RDF data provides a declarative access mechanism that is suitable for application usage and remote access. We describe work on a conceptual model for querying RDF data that refines ideas first presented in at the W3C workshop on Query Languages [14] and the design of one possible syntax, derived from [7], that is suitable for application programmers. Further, we present experience gained in three implementations of the query language.

Introduction

An SQL-ish query language for RDF provides consistent, human-understandable, access to repositories of semantic data, whether stored files or large databases, enabling application programmers to create semantic web applications quickly. SquishQL syntax and model is designed to reflect RDF's graph syntax, and uses SQL-like constructs so that application developers can pick it up as quickly as possible.

As application programmers ourselves, working with RDF, we have implemented query software because we needed it to use RDF effectively. RDF APIs provide a high degree of control at a fine level of granularity that supports a range of programming paradigms; query is one such paradigm which is coarser-grained and useful when there is large amounts of data with semi-regular structure. The query paradigm makes for more intuitive access with a shorter learning curve, as well as making access possible in situations where there is a high operation overhead, for example, over remote access protocols such as SOAP.

The query paradigm is also suitable for scripting environments, enabling the creation of RDF-driven applications quickly and easily. Reduced "time-to-deploy" is an important issue in enabling the semantic web.

In this paper, we describe the conceptual framework for the query language: this is closely tied to the RDF graph, providing a base level of query of RDF data. We relate this work to other systems, then describe a syntax for our query language suitable for application programmers. We describe our experience with three different

I. Horrocks and J. Hendler (Eds.): ISWC 2002, LNCS 2342, pp. 423–435, 2002.

implementations and note experiments in providing inference support without modifying the query language itself.

A Query Framework for RDF

The RDF Model and Syntax specification [15], which has been formalized in the proposed model theory for RDF [16], makes the graph the primary syntax for RDF. This graph is a partially labeled, directed graph where the node labels are URIs or literals, and the arc labels are URIs. No two nodes can have the same label; there cannot be two arcs with the same label between the same pair of nodes. Not all nodes need be labeled giving rise to bNodes (anonymous nodes) which have no URI label. The N-Triples syntax [22] records such a graph as a list of its edges.

In [14], a query model for RDF is proposed in which the query is an RDF model, where any resource, property or literal can be replaced by a variable.

The result is a pair: a subgraph of the target knowledge base that matches the query and a table of sets of legal values for the variables. In addition, the paper notes that RDF schema constructs such as *subClassOf* and *subPropertyOf* provide some specific inferencing. Their query proposal is to provide a query parameter which specifies whether the query is on the underlying RDF graph or the deductive closure of the graph, thought of as a knowledge base.

Such a query model provides a baseline query model for the semantic web – the query language works against the RDF model without any higher-level constructs in the language for inference or interpretation. While an RDF model implementation may itself provide additional features, this approach to query works by assuming any feature (such as inferencing) manifests itself as an RDF graph that can be accessed by some graph API.

The SquishQL Model of Query

The RDF Working Group has defined a graph syntax as the primary representation of the conceptual model for RDF. Concrete syntaxes in XML and N-Triples are defined. We see that the query form proposed in [14] is a fixed-sized graph pattern over the syntax of the RDF graph where the pattern is formed from variables for nodes (labeled or unlabeled), arcs or literals. This matching does not depend on the graph interpretation. SquishQL adds to this baseline query model, introducing filter functions over the variables, which restrict the values that the variables can take. These filter functions do not change the expressive power of the graph pattern.

The pattern language is formed from:

- Triple patterns, which describe one edge of the graph, allowing either a variable or an explicit value for each of subject, predicate and object. In the syntax below, variables are indicated by '?': the most general pattern is (?x, ?y, ?z) which matches any triple.
- Graph patterns, which describe the graph shape, expressed as a collection of triple patterns. In the syntax below, there is a list of triple patterns which are interpreted

as the conjunction of the triple patterns. This list is an edge-list of the graph pattern.

This results in quite a weak pattern language but it does ensure that in a result all variables are bound (there is no disjunction). The language does not express transitivity or other forms of unknown length paths in the graph.

The filter functions are Boolean expressions over the values of URIs or literals.

We return all the possible ways the graph pattern and filter functions can be matched against the graph. Because the language does not have disjunction or repetition, we only need to return the values for the variables, and not the particular subgraph that caused each possible match, because substitution of the values for the variables into the original query is sufficient to identify the particular sub graph (set of edges) that caused a particular match.

Variables and bNodes

A variable in a query is not the same as a bNode (an unlabeled "blank" node in an RDF graph). Although bNodes are treated as existentially bound variables in the proposed model theory for RDF there are some differences from the variables in edge patterns:

1. The set of variables in a query is distinct from the set of bNodes in the data graph.
2. Query variables match nodes; hence can match resource URIs or literals or graph bNodes.
3. Query variables can label arcs, hence can match property URIs.

RDF as N-Triples

If we think of the RDF model as a table of triples in N-Triples form, then the subgraph pattern becomes a list of triples, except with the possibility of variables. Then, both triple patterns and Boolean expressions are constraints on the results of a query. Each of the constraints must be true, either because a triple pattern becomes a triple to be found in the RDF graph, or because a Boolean expression evaluates to true.

Other Systems

There are a number of other query languages of RDF available, some of which have been in use for several years. One of the earliest was rdfDB [7] and this is the basis for SquishQL syntax. It is a simple graph-matching query language designed for use in the rdfDB database system. It differs slightly syntactically from SquishQL and also does not contain the constraints on the variables used by SquishQL. It returns results as a table.

Algae [6] is another early query language for RDF. It uses an S-expression syntax to do graph matching. It is used to power the W3C's Annotea annotations system [12], and other software at the W3C. It is written in Perl, and can be used with an SQL database. It returns a set of triples in support of each result. One of our implementations (RDQL/Jena) does retain the triples used to bind variables and the application can access this information.

RQL [9,8] is a combined RDF store and query system. It also provides a schema validating parser and has a syntax targeted at RDFS[17] applications. It can perform similar queries to SquishQL, with the added power of support for transitive closure on RDFS subclass and subproperty. This is also the query language used by Sesame [10].

The EDUTELLA system [11] provides a hierarchy of query languages: the lowest level, RDF-QEL-1 provides a graph pattern language, expressed as an RDF model.

Other systems with similar query languages include RDFQL [20] and one described in [13]. RDFQL combines a rules language with query; the underlying database system interprets the rules. The language in [13] has syntactic support for regular expression path expressions.

SquishQL - A Textual Language

In this section, we describe a syntax that has been implemented in at least three systems [1,2,3]. The syntax described below is just one possible syntax for the abstract query model outlined. We have chosen to define a syntax for the application developer (a person). Other syntactic forms, more suited to production by tools, such as RDF or XML, will also be needed; see [23] for an experiment in a RuleML syntax.

In SquishQL, there are two classes of constraints; patterns and filter expressions. Patterns are generative (they create bindings) and the filters are restrictive (they remove possibilities). SquishQL separates these into the WHERE clause (generative) and the AND clause (restrictive).

Some query systems have followed the tradition of having predicate first. We have chosen instead to mimic N-Triples syntax and specify triple patterns as subject-predicate-object.

```
SELECT ?title
FROM http://example.com/xmleurope/presentations.rdf
WHERE
    (?doc, <dc:title>, ?title) ,
    (?doc, <rdf:type>, <foaf:Document>)
USING
  dc   FOR <http://purl.org/dc/elements/1.1/>,
  foaf FOR <http://xmlns.com/foaf/0.1/>,
  rdf  FOR <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

Fig. 1. Query: Find me the titles of documents in http://example.com/xmleurope/presentations.rdf

In SQL, a database is a closed world; the FROM clause identifies the tables in the database; the WHERE clause identifies constraints and can be extended with AND. By analogy, the web is the database and the FROM clause identifies the RDF models. Variables are introduced with a leading '?' and URIs are quoted with <> [19]; unquoted URIs can be used where there is no ambiguity.

SELECT Clause
: Identifies the variables to be returned to the application. If not all the variables are needed by the application, then specifying the required results can reduce the amount of memory needed for the results set as well as providing information to a query optimizer.

FROM Clause
: The FROM clause specifies the model by URI.

WHERE Clause
: Specifies the graph pattern as the conjunction of the list of triple patterns.

AND Clause
: Specifies the Boolean expressions over values of URIs and literals, including arithmetic comparisons, and boolean expressions, including disjunction and negation.

USING Clause
: A way to shorten the length of URIs. As SquishQL is likely to be written by people, this mechanism helps make for an easier to understand syntax. This is not a namespace mechanism; instead, it is simply an abbreviation mechanism for long URIs by defining a string prefix.

The RDF specification defined the form of containers and of reification. There is no explicit syntax for these in SquishQL. As shown in the examples, this does not affect retrieving data from containers, but the query can become cumbersome. Similarly, with reification, the lack of syntactic support can make expressing some queries awkward.

```
SELECT ?y
WHERE (<http://somewhere.com/aBag>, ?x, ?y)
AND ! ( ?x eq <rsyn:type> && ?y eq <rsyn:Bag>)
USING
  rsyn FOR http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

Fig. 2. Extract the contents of a bag

Query Evaluation Issues

A number of issues arise in defining the evaluation of a query:

1. Treatment of anonymous nodes, which are scoped to the document containing the syntactic RDF.
2. Expression evaluation in the absence of formal datatyping in RDF.

Anonymous Nodes

In general, it is not possible to write SquishQL syntax queries that contain bNodes as values in triple patterns because they have no syntactic representation. Some systems manifest bNodes as automatically generated URIs and could have queries with bNodes. In RDQL/Jena, it is possible to construct queries through a programmatic API, not just via a parser, so bNodes, as Jena resources, can be included. This has an

impact on remote operation where queries are passed to a server for execution – this requires a serialisable form.

Data Types and Filter Evaluation
RDF does not define a type system for literals. However, to provide a range of operators for Boolean filters, we have to decide on the type for a literal. This is currently done by attempting to parse the literal during query execution as the type required by the expression (number, string etc).

Inkling

Inkling [1,5] is a Java™ implementation of SquishQL created to be API and database-independent for testing the usefulness of SquishQL for comparatively small-scale projects. The aim was to have a query engine that could be used with almost any RDF database implementation written in Java, and which could be used for experimenting with the SquishQL query language.

API and Database Independence
Inkling can query most Java RDF database implementations and most Java RDF APIs, whether the implementations are in memory or use some form of persistent storage. For Inkling to be able to talk to an RDF database or service, that service just has to implement an extremely basic interface consisting of a single method. This method is a three-place search method:

```
queryDatabase(subject, predicate, object)
```

where any argument can be null, which was the lowest common denominator of methods supported by the APIs examined (Jena [4] and Stanford RDF [18] API).

This provides a generic interface to database storage. This mechanism allows the storage subsystem to make decisions about access to stored triples. Different database layouts can make different tradeoffs between efficient indexing structures and increased storage costs. There are some problems with the simplicity of the interface, for example, if there are optimizations the database can perform for certain kinds of queries such as reification, bags and sequences, then this per-triple interface won't be useful.

This is the simplest means for accessing the database. Where optimizations are possible, for example if the underlying database supports a similar query language, efficiencies can be made by passing the entire query to the underlying database.

JDBC Interfaces
Inkling uses the JDBC interfaces to make SquishQL queries. This enables the implementation to be fairly independent of the database to be searched, and also means that Java programmers will be familiar with the means of accessing the queries.

Example

One way we have used Inkling is as an access mechanism for a testbed of information about people, RDFWeb (http://rdfweb.org). The test data consists of files containing information about people, the people they know, images of them, and information about documents they have created, rather like homepages. The files are created by many different people, and may contain arbitrary information from any vocabulary. Much of the data is in the experimental 'foaf' (friend of a friend) vocabulary, which is continually expanding as people see the need for other characteristics.

Inkling enables the information to be harvested from the web, so that the database can be rebuilt at the frequency desired. Then the information can be pulled out of the database as it is required. RDFWeb displays a person-centric view of the data, where queries pull out information about a particular person within a server-side JSP page.

```
SELECT ?name, ?mbox
WHERE
  (?libby, <foaf:mbox>,
           <mailto:libby.miller@bristol.ac.uk>) ,
  (?libby, <foaf:knows>, ?someone) ,
  (?someone, <foaf:mbox>, ?mbox) ,
  (?someone, <foaf:name>, ?name)
USING foaf for <http://xmlns.com/foaf/0.1/>
```

Fig. 3. Query: Get names and email addresses of the people that the person with email address libby.miller@bristol.ac.uk knows.

This works well if all the people in the database have both mailboxes and names associated with them, but will fail if the data is inconsistent, which it may be, since this is a distributed database with many authors. There is no way of saying that a subquery is optional in SquishQL: a safer course may be to query for the email addresses of people known, and then query for their names.

```
SELECT ?title, ?description, ?name
WHERE
 (?libby, <foaf:mbox>,
          <mailto:libby.miller@bristol.ac.uk>) ,
 (?paper, <dc:contributor>, ?libby),
 (?paper, <dc:title>, ?title) ,
 (?paper, <dc:description>, ?description) ,
 (?paper, <dc:contributor>, ?someone) ,
 (?someone, <foaf:name>, ?name)
USING foaf for <http://xmlns.com/foaf/0.1/> ,
      dc   for <http://purl.org/dc/1.1/>
```

Fig. 4. Query: Get me the titles and descriptions of papers that Libby has written, and the names of anyone who wrote the paper with her.

This query will pull out anything to which the person with email address libby.miller@bristol.ac.uk is a contributor where there is at least one other contributor with an email address - although this could be libby.miller@bristol.ac.uk. Where there is more than one contributor, the result set will be repetitious: it will pull out the

title and description again for each contributor. Similarly, if a contributor has more than one name in the database it will pull out the title and description again for that. Finally, if libby has done a paper on her own, and is therefore listed as the dc:creator rather than the dc:contributor then this query will not get that information. There is no way of writing 'or' of graph patterns in SquishQL, for reasons of simplicity and tractability. Instead one would have to make two queries to access this information.

These types of queries can be made using one or more RDF files from an in-memory database, or from an SQL backed database. Because of the flexibility of query it is a simple matter to alter the information shown: if a great deal of data about, say, eye colour, is made available, then it is simple to add an additional query into the Java Server page.

RDQL

RDQL[2] is an implementation of SquishQL for the Jena RDF toolkit[4]. Jena is a collection of RDF tools written in Java that includes:

- A Java API
- ARP: An RDF parser
- RDQL: A query system
- Support classes for DAML+OIL ontologies
- Persistent storage based on BerkeleyDB.
- Persistent storage based on various relational databases.

Jena and RDQL

RDQL allows queries to be made on RDF models from Java on any Jena model so the query system is independent of the storage implementation and of the RDF syntax. RDQL can be mixed with Jena API calls because a query returns the underlying Jena objects that satisfy the query so the Resources, Properties or Literals retrieved can be used for model update or other API calls. Combining programming paradigms can be useful; for example, this has been used in calculating RDF Schema closures, where each rule condition is a query and each additional statement is added though the Jena API.

Execution

In order to reduce the amount of working memory needed by a query, the execution of a query is carried out in parallel with the application consuming the results. In the simple query engine currently supplied in the Jena toolkit, there are three threads, one is used for matching triple patterns against the model in a depth-first traversal, one is used for filter evaluation and one is the application thread that issued the query and is processing the results. The bounded buffers between the threads control the query execution so that the query execution engine is limited in how far ahead of the application processing. The amount of working memory needed is independent of the size of the data in the model, but not the query. If the application does not process results fast enough, the query system will pause when buffers become full.

Implementation in Jena

The triple pattern matcher uses a similar scheme to Inkling: the main Jena operation used takes a description of the triples used, in the form of fixing the Resources, Properties or Literals or leaving that item unconstrained. All the current implementations of a Jena model provide indexes by Subject, Predicate or Object so access is efficient.

However, the filter expressions used to restrict the values are not passed down to the underlying storage. RDF does not define type information so general stores hold only strings and RDQL interprets whether a value is, say, an integer, based on whether it can interpret it as such when filtering a result generated by the triple matcher.

An Inferencing Store for Jena

An experimental inferencing store based on Prolog has been developed. This inferencing system has been used for queries of RDFS [17] data. The properties *subClassOf* and *subPropertyOf* are made to behave transitively and the property type *rdf:type* returns inferred types of resources as well as the declared types. Like RQL, we have chosen to redefine the standard properties *subClassOf* and *subPropertyOf*: although this is done as part of the model, not as part of the query language. An alternative design would be to create new properties for the transitive relationships, *anySubClassOF*, *anySubPropertyOf*, leaving the original properties for access to the original data.

```
SELECT ?x
WHERE (<http://somewhere/resource>, <rdf:type>, ?x)
USING
  rdf FOR <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

Fig. 5. Query: return all the types for the resource

RDFStore

RDFStore [3] implements the SquishQL language to query RDF repositories directly from Perl. The toolkit consists of a Perl API, a streaming SiRPAC parser and a generic hashed data storage custom designed for the RDF model. The storage subsystem allows transparently storage and retrieval of RDF nodes, arcs and labels, either from an in-memory structure, from the local disk or from a very fast and scaleable remote storage [24]. The latter is a fast networked TCP/IP based transactional storage library that uses multiple single key hash based BerkeleyDB files together with an optimized network routing daemon with a single thread/process per database. The data indexing model is general enough to retrieve RDF subgraphs and properties using free-text and statement-group sensible matching. Each literal value gets indexed in its full Unicode [25] form and in-memory data structures or objects can also be serialised on disk. The API supports bNodes (blank Nodes or anonymous-resources) but the storage internally does treat them like any other resource. Being in Perl, an untyped language, the toolkit at the moment does not treat typed literals in any special

way; all query filtering operations on the values are processed using pure Perl regular expressions and eval constructs.

Running RDF Queries with RDFStore

The API implements the Stanford RDF API [18] and supports the basic three-place search method (or triple matching) which is the atomic query construct available; each result set consists of an RDF model which can be further queried, serialized or enumerated in its component statements. A much more property-centric programming interface to an RDF repository is provided via the RDQL driver; by running an SQL-like query on an actual storage it is possible to access the single nodes, arcs and labels as resources or literals. Such a query paradigm being much more consistent and human-understandable, has proved to be very practical and flexible compared to other similar approaches. Once the query has been carried out, the result set is actually being stored into a Perl hash data structure and further processed with common programming constructs.

Query Processing and Execution

The query processing and execution is performed on the client side and the approach is fundamentally different from other similar techniques such as rdfdb [7] where the query parsing, processing and execution is done entirely on the server side. Having it running on the client makes the DBMS server back-end much more generic and lightweight; then by using some kind of compression of data indexes both storage and network traffic can be reduced.

RDFStore contains a hand-written top-down LL(1) parser for the RDQL syntax with extensions to the basic pattern language to run free-text selection of RDF graph elements. When a query is processed, it gets parsed into an in-memory structure that is then used to run the actual query on the back-end database; the RDQL query internally is implemented as a join of a series of basic three-place searches on the RDF graph. Then the constraints are applied and the actual results returned to the user. RDFStore contains a module to make basic RDF Schema inferencing on triples but the RDQL sub-system is not using it yet. By using free-text words present into literal values it is possible to select the nodes matching a query criteria in a much more selective way.

```
SELECT ?link
FROM <http://xmlhack.com/rss10.php>
WHERE
  (?item, <rdf:type>, <rss:item>),
  (?item, <rss:title>, %"Apache"%),
  (?item, <rss:link>, ?link)
USING
  rdf for <http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
  rss for <http://purl.org/rss/1.0/>
```

Fig. 6. Query involving free-text matching in an RDFStore query.

Such an extension is similar to the SQL LIKE operator present in most of out-of-the-shelf database solutions today; this kind of filtering can also be applied to the restrictive part (constraints) of the query by using the LIKE operator directly with Perl

regular-expressions. The following returns the same result of the example above in less efficient way:

```
SELECT ?link
FROM <http://xmlhack.com/rss10.php>
WHERE
  (?item, <rdf:type>, <rss:item>),
  (?item, <rss:title>, ?title),
  (?item, <rss:link>, ?link)
AND ?title LIKE '/Apache/i'
USING
  rdf for <http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
  rss for <http://purl.org/rss/1.0/>
```

Real-World Example of RDQL

For the ETB project [26] we successfully used the RDQL syntax to query fully RDF DC/DCQ metadata records classified accordingly to a multilingual thesaurus in 8 languages. An ETB metadata record is describing an educational resource having a specific target audience, rights management and quality selection criteria; each single record describes a Web resource, which can reside on several distributed servers having several different multilingual translations of it. Each record description is put in the statement group of a specific indexing term of the multilingual thesaurus and associated with a Collection Level Description (CLD) [27].

```
SELECT ?title_value, ?title_language,
       ?subject_value,?subject_language,
       ?description_value, ?description_language,
       ?language, ?identifier
WHERE (?x, <dc:title>, ?tt),
      (?tt, <rdf:value>, ?title_value),
      (?tt, <dc:language>, ?ttl),
      (?ttl, <dcq:RFC1766>, ?title_language),
      (?x, <dc:subject>, ?ss1),
      (?ss1, <etbthes:ETBT>, ?ss2),
      (?ss2, <rdf:value>, ?subject_value),
      (?ss2, <dc:language>, ?ss3),
      (?ss3, <dcq:RFC1766>, ?subject_language),
      (?x, <dc:description>, ?dd),
      (?dd, <rdf:value>, ?description_value),
      (?dd, <dc:language>, ?ddl),
      (?ddl, <dcq:RFC1766>, ?description_language),
      (?x, <dc:identifier>, ?identifier),
      (?x, <dc:language>, ?lll),
      (?lll, <dcq:RFC1766>, ?language)
USING
  rdf for <http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
  rdfs for <http://www.w3.org/2000/01/rdf-schema#>,
  dc for <http://purl.org/dc/elements/1.1/>,
  dcq for <http://purl.org/dc/terms/>,
  dct for <http://purl.org/dc/dcmitype/>,
  etb for <http://eun.org/etb/elements/>,
  etbthes for <http://eun.org/etb/thesaurus/elements/>
```

Fig. 7. Example query from the ETB system.

The example below is in practice very simple compared to other full-blown Boolean SQL statements we need to run in the advanced search of the production system. Such an advanced feature has been implemented using a hybrid solution using RDQL, simple triple-matching and basic programming data structures; the result has shown quite a good scalability but we are investigating the real implications and aspect of running full queries with disjunction and negation.

Conclusions

We have described a refined framework for the querying of RDF data. We developed this to meet our needs in writing RDF applications. The query framework follows the RDF graph syntax very closely and provides for extraction of RDF data (URIs, literals, bNodes) from a data source.

The syntax we present is modeled after SQL. It is targeted at the application developer. Other syntaxes would be appropriate for different communities.

We have implemented SquishQL in three systems. The first, Inkling, stores RDF data in relational databases or in external XML files. The second implementation, RDQL, is part of the Jena RDF toolkit and combines query with manipulation of the RDF graph at a fine-grained level through the Jena RDF API. The third, in RDFStore, is close coupling of RDF and Perl data access styles.

We have shown the utility of this simple approach to query of RDF data in a number of applications as well as other applications developed by the RDF developer community.

References

1. L. Miller, “Inkling: RDF query using SquishQL”, web page: http://swordfish.rdfweb.org/rdfquery/
2. A. Seaborne, RDQL – RDF Data Query Language, part of the Jena RDF Toolkit, HPLabs Semantic Web activity, http://hpl.hp.com/semweb/, RDQL grammar: http://www.hpl.hp.com/semweb/rdql-grammar.html
3. A. Reggiori, D. W. van Gulik, RDFStore, http://rdfstore.sourceforge.net
4. B. McBride, “Jena: Implementing the RDF Model and Syntax Specification”, in: Steffen Staab et al (eds.): “Proceedings of the Second International Workshop on the Semantic Web - SemWeb'2001”, May 2001 http://www.hpl.hp.co.uk/people/bwm/papers/20001221-paper/
5. D. Brickley, L. Miller, “RDF, SQL and the Semantic Web - a case study”, http://ilrt.org/discovery/2000/10/swsql/
6. E. Prud’hommeaux, Algae in “RDF Database Library”, http://www.w3.org/2001/Talks/0505-perl-RDF-lib/slide5-0.html
7. R.V.Guha, “rdfDB : An RDF Database”, web page: http://guha.com/rdfdb/
8. G. Karvounarakis, V. Christophides, D. Plexousakis, S Alexaki, “Querying Community Web Portals”, SIGMOD2000, http://www.ics.forth.gr/proj/isst/RDF/RQL/rql.html
9. Greg Karvounarakis, “The RDF Query Language (RQL)”

10. Sesame, see http://sesame.aidministrator.nl/, part of the OntoKnowledge project, http://www.ontoknowledge.org/
11. W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér, T. Risch, "EDUTELLA: A P2P Networking Infrastructure Based on RDF", http://edutella.jxta.org/reports/edutella-whitepaper.pdf
12. J. Kahan , M Koivunen, E. Prud'Hommeaux, R R. Swick "Annotea: An Open RDF Infrastructure for Shared Web Annotations", http://www10.org/cdrom/papers/488/
13. D. Allsopp, P. Beautement, J. Carson, M Kirton "Toward Semantic Interoperability in Agent-Based Coalition Command Systems", Proceedings of the First International Semantic Web Workshop, July 30-31, 2001, http://www.semanticweb.org/SWWS/program/full/paper10.pdf
14. R.V. Guha, Ora Lassila, Eric Miller, Dan Brickley, Enabling Inference, W3C Query Language meeting, Boston, December 3-4, 1998.
15. Ora Lassila, Ralph R. Swick (editors), "Resource Description Framework (RDF) Model and Syntax Specification", 22 February 1999.
16. P. Hayes (editor), "RDF Model Theory" (work in progress) http://www.w3.org/TR/rdf-mt/
17. Dan Brickley, R.V. Guha (editors), "Resource Description Framework (RDF) Schema Specification 1.0", 27 March 2000 (W3C Candidate Recommendation).
18. Sergey Melnik, "Stanford RDF API", web page: http://www-db.stanford.edu/~melnik/rdf/api.html
19. T. Berners-Lee, R. Fielding, L. Mastiner, "Uniform Resource Identifiers (URI): Generic Syntax", RFC2396
20. Intellidimension Inc. "RDFQL" http://www.intellidimension.com/RDFGateway/Docs/rdfqlgettingstarted.asp
21. J. De Roo, Euler proof mechanism, http://www.agfa.com/w3c/euler/
22. N-Triples syntax in W3C Working Draft "RDF Test Cases" http://www.w3.org/TR/rdf-testcases/#ntriples
23. G. Chappell, RuleML combined with RDF query model: http://209.198.94.130/ruleml/query.asp
24. Dirk Willem-van Gulik, "The DB engine", August 1999, http://rdfstore.sourceforge.net/dbms.html
25. The Unicode Consortium, *"The Unicode Standard Version 3.0"*, ISBN 0-201-61633-5
26. European Schoolnet (EUN) European Treasury Browser (ETB) project, http://etb.eun.org
27. Andy Powell, *"Collection Description – Study, Recommendation, Specification"*, 3 September 1999, http://www.ukoln.ac.uk/metadata/rslp/proposal/