Appendix B: Brief descriptions of the main mentioned tools

B1. Protégé 4.2

http://protege.stanford.edu/

Protégé was developed by the "Stanford Center for Biomedical Informatics Research" at the Stanford University School of Medicine. This is a tool which allows a user to construct domain ontology, customize data entry forms and enter data. The tool can be easily extended to access other knowledge based embedded applications. For example, Graphical widgets can be added for tables and diagrams. Protégé can also be used by other applications to access the data.

Protégé allows a user to simultaneously work on classes and instances. This is provided for by a uniform GUI whose top level is composed of overlapping tabs for compact representation [protégé, 2012; protégé-owl, 2012].

Protégé platform supports two main ways of modeling ontologies:

- *Protégé-Frames editor*: enables users to build and distribute ontologies, which are based on frame structures corresponding to "Open Knowledge Base Connectivity" (OKBC) protocol;
- *Protégé-OWL editor*: enables users to build ontologies on the Semantic Web, especially the using W3C's Web Ontology Language (OWL) [OWL, 2004].

Protégé Basic features:

- Import format - XML, RDF(S) and XML Schema;
- Export format - XML, RDF(S), XML Schema, FLogic, CLIPS and Java HTML;
- Graph view - Via GraphViz plug-in (browsing of classes and global properties); Via Jambalaya plug-in (nested graph view);
- Consistency check - Via plug-ins (PAL and FaCT);
- Limited multi-user support - Protégé has some multi-user capabilities added to it. It is intended for experienced Protégé users. Multiple users can read the same database and make incremental changes or changes that don't conflict with one another. However, there's no support for multiple users trying to modify the same elements of a knowledge base or notification of changes made by other users. Concurrent changes to the same section will cause severe problems;

- Web support - Via Protégé-OWL plug-in; Protégé doesn't provide direct support for accessing knowledge base from the web, but it can easily be done. A number of users have communicated with Protégé knowledge bases from the Web via servlets. Protégé can be run as an applet.

Additional features:

- Merging - Via Anchor-PROMPT plug-in;
- Not support to add a new basic type;
- Extensible plug-in architecture;
- Ontology storage - File and DBMS (JDBC).

There is an additional option in Protégé, which serves the storing of ontologies in various relational databases, called OntoBase [Yabloko, 2011].

It should be noted that the same name “OntoBase” is used in [Pan & Pan, 2006], but without any connection to Protégé.

Originally, Protégé has a single table that stores entire contents of the knowledge base which is developed as a frame based one [protégé, 2012].

The frame table has a fixed number of columns which are listed below in Table 74. It includes classes, slots, facets and instances. The Protégé meta-class architecture is used explicitly in the table to simplify things: all classes, slots, and facets are treated as frames.

Each entry in the database corresponds to a frame in Protégé. Classes have slots such as ":DIRECT_SUPERCLASS" to maintain the inheritance hierarchy. All frames have a “:NAME” slot which contains name of frame.

Table 74. Protégé database format

Column	Description
frame - [integer]	frame id Frame ID's < 10000 are reserved for the system. The frame ids for system frames are declared in the file: edu.stanford.smi.protege.model.Model.java
frame_type - [smallint]	same as "value_type" but for the frame column
slot - [integer]	slot frame id
facet - [integer]	facet frame id (0 if not a facet value)
is_template - [smallint]	0 => value is OKBC "own", 1 => value is OKBC "template"
value_index - [integer]	number used to maintain relative ordering of slot_or_facet_value entries for a frame-slot(-facet) combination
value_type - [smallint]	number used to indicate the "type" of the value stored in slot_or_facet_value. The number-to-type conversion is given in the file: edu.stanford.smi.protege.storage.database.DatabaseUtils.java
slot_or_facet_value - [varchar(N)]	facet value if facet is not 0, slot value otherwise. Holds values of length that will fit in a varchar (typically <= 255)
long_slot_or_facet_value - [longvarchar]	same as slot_or_facet_value but holds values too long to fit in slot_or_facet_value

In the case of the superclass and subclass relations, Protégé stores duplicated information.

For example with class A it stores that its subclass is B and with B it stores that its superclass is A.

Maintaining separate records for these relations is necessary to maintain the ordering of both subclasses and superclasses. So while the "slot value" information is indeed duplicated in these records, the "index" information is unique (Subclass ordering is a user-interface feature that a number of users have requested. Protégé attaches no meaning to the ordering of superclasses or subclasses.) [protégé, 2012].

To illustrate the using of Protégé we use our sample graph.

For easy reading, below we reproduce the description by triples of the sample graph.

Subject	*Relation*	*Object*
Alice	*has_characteristics*	**Alice – Age : 18**
Alice	*knows*	**Bob – since : 2001/10/03**
Alice	*is_member*	**Chess – since : 2005/07/01**
Bob	*has_characteristics*	**Bob – Age : 22**
Bob	*knows*	**Alice – since : 2001/10/04**
Bob	*is_member*	**Chess – since : 2011/02/14**
Chess	*has_characteristics*	**Chess –Type : Group**
Chess	*members*	**Alice; Bob**

The visualization is shown at Figure 110. Some information could not be viewed (like dates and etc.). The corresponded OWL and RDF descriptions of the sample graph are given in Table 75 and Table 76.

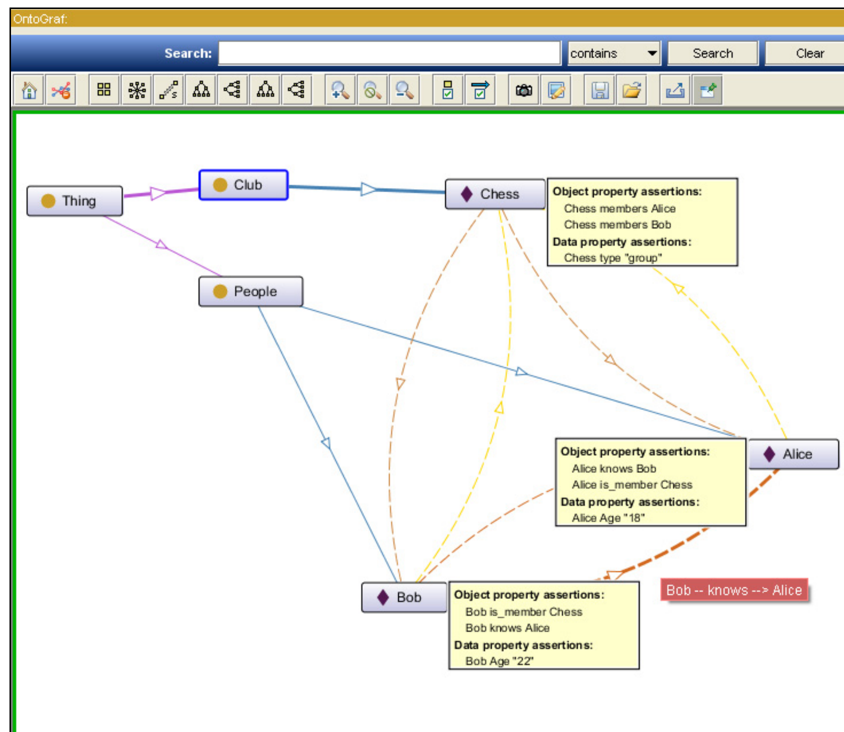

Figure 110. Protégé graphical representation of the sample graph

Table 75. The Protégé QWL description of the sample graph

```
Prefix(owl:=<http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
        18#http://www.w3.org/2002/07/owl#>)
Prefix(rdf:=<http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
        18#http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
Prefix(xml:=<http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
        18#http://www.w3.org/XML/1998/namespace>)
Prefix(xsd:=<http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
        18#http://www.w3.org/2001/XMLSchema#>)
Prefix(rdfs:=<http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
        18#http://www.w3.org/2000/01/rdf-schema#>)

Ontology(<http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
        18#http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-18>

Declaration(Class(<http://www.co-ode.org/ontologies/ont.owl#Club>))
Declaration(Class(<http://www.co-ode.org/ontologies/ont.owl#People>))
Declaration(ObjectProperty(<http://www.co-ode.org/ontologies/ont.owl#is_member>))
AnnotationAssertion(<http://www.co-ode.org/ontologies/ont.owl#since> <http://www.co-
        ode.org/ontologies/ont.owl#is_member> "")
InverseObjectProperties(<http://www.co-ode.org/ontologies/ont.owl#members> <http://www.co-
        ode.org/ontologies/ont.owl#is_member>)
InverseFunctionalObjectProperty(<http://www.co-ode.org/ontologies/ont.owl#is_member>)
Declaration(ObjectProperty(<http://www.co-ode.org/ontologies/ont.owl#knows>))
AnnotationAssertion(<http://www.co-ode.org/ontologies/ont.owl#since> <http://www.co-
        ode.org/ontologies/ont.owl#knows> "")
Declaration(ObjectProperty(<http://www.co-ode.org/ontologies/ont.owl#members>))
AnnotationAssertion(<http://www.co-ode.org/ontologies/ont.owl#since> <http://www.co-
        ode.org/ontologies/ont.owl#members> "")
InverseObjectProperties(<http://www.co-ode.org/ontologies/ont.owl#members> <http://www.co-
        ode.org/ontologies/ont.owl#is_member>)
FunctionalObjectProperty(<http://www.co-ode.org/ontologies/ont.owl#members>)
Declaration(DataProperty(<http://www.co-ode.org/ontologies/ont.owl#Age>))
DataPropertyDomain(<http://www.co-ode.org/ontologies/ont.owl#Age> <http://www.co-
        ode.org/ontologies/ont.owl#People>)
DataPropertyDomain(<http://www.co-ode.org/ontologies/ont.owl#Age>
        DataAllValuesFrom(<http://www.co-ode.org/ontologies/ont.owl#Age>
        <http://www.w3.org/2001/XMLSchema#decimal>))
Declaration(DataProperty(<http://www.co-ode.org/ontologies/ont.owl#type>))
DataPropertyDomain(<http://www.co-ode.org/ontologies/ont.owl#type>
        DataAllValuesFrom(<http://www.co-ode.org/ontologies/ont.owl#type>
        <http://www.w3.org/2000/01/rdf-schema#Literal>))
Declaration(NamedIndividual(<http://www.co-ode.org/ontologies/ont.owl#Alice>))
ClassAssertion(<http://www.co-ode.org/ontologies/ont.owl#People> <http://www.co-
        ode.org/ontologies/ont.owl#Alice>)
ObjectPropertyAssertion(Annotation(<http://www.co-ode.org/ontologies/ont.owl#since>
        "2005/07/01") <http://www.co-ode.org/ontologies/ont.owl#is_member> <http://www.co-
        ode.org/ontologies/ont.owl#Alice> <http://www.co-ode.org/ontologies/ont.owl#Chess>)
ObjectPropertyAssertion(Annotation(<http://www.co-ode.org/ontologies/ont.owl#since>
        "2001/10/03") <http://www.co-ode.org/ontologies/ont.owl#knows> <http://www.co-
        ode.org/ontologies/ont.owl#Alice> <http://www.co-ode.org/ontologies/ont.owl#Bob>)
```

```
DataPropertyAssertion(<http://www.co-ode.org/ontologies/ont.owl#Age> <http://www.co-
        ode.org/ontologies/ont.owl#Alice> "18")
Declaration(NamedIndividual(<http://www.co-ode.org/ontologies/ont.owl#Bob>))
ClassAssertion(<http://www.co-ode.org/ontologies/ont.owl#People> <http://www.co-
        ode.org/ontologies/ont.owl#Bob>)
ObjectPropertyAssertion(Annotation(<http://www.co-ode.org/ontologies/ont.owl#since>
        "2011/02/14") <http://www.co-ode.org/ontologies/ont.owl#is_member> <http://www.co-
        ode.org/ontologies/ont.owl#Bob> <http://www.co-ode.org/ontologies/ont.owl#Chess>)
ObjectPropertyAssertion(Annotation(<http://www.co-ode.org/ontologies/ont.owl#since>
        "2001/10/04") <http://www.co-ode.org/ontologies/ont.owl#knows> <http://www.co-
        ode.org/ontologies/ont.owl#Bob> <http://www.co-ode.org/ontologies/ont.owl#Alice>)
DataPropertyAssertion(<http://www.co-ode.org/ontologies/ont.owl#Age> <http://www.co-
        ode.org/ontologies/ont.owl#Bob> "22")
Declaration(NamedIndividual(<http://www.co-ode.org/ontologies/ont.owl#Chess>))
ClassAssertion(<http://www.co-ode.org/ontologies/ont.owl#Club> <http://www.co-
        ode.org/ontologies/ont.owl#Chess>)
ObjectPropertyAssertion(Annotation(<http://www.co-ode.org/ontologies/ont.owl#since>
        "2011/02/14") <http://www.co-ode.org/ontologies/ont.owl#members> <http://www.co-
        ode.org/ontologies/ont.owl#Chess> <http://www.co-ode.org/ontologies/ont.owl#Bob>)
ObjectPropertyAssertion(Annotation(<http://www.co-ode.org/ontologies/ont.owl#since>
        "2005/07/01") <http://www.co-ode.org/ontologies/ont.owl#members> <http://www.co-
        ode.org/ontologies/ont.owl#Chess> <http://www.co-ode.org/ontologies/ont.owl#Alice>)
DataPropertyAssertion(<http://www.co-ode.org/ontologies/ont.owl#type> <http://www.co-
        ode.org/ontologies/ont.owl#Chess> "group")
Declaration(AnnotationProperty(<http://www.co-ode.org/ontologies/ont.owl#since>))
)
```

Table 76. The Protégé RDF description of the sample graph

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

<!DOCTYPE rdf:RDF [
   <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
   <!ENTITY ont "http://www.co-ode.org/ontologies/ont.owl#" >
   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
   <!ENTITY owl "http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
18#http://www.w3.org/2002/07/owl#" >
   <!ENTITY xsd "http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
18#http://www.w3.org/2001/XMLSchema#" >
   <!ENTITY xml "http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
18#http://www.w3.org/XML/1998/namespace" >
   <!ENTITY rdfs "http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
18#http://www.w3.org/2000/01/rdf-schema#" >
   <!ENTITY rdf "http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
18#http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<rdf:RDF xmlns="http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-
18#http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-18#"
```

```
    xml:base="http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-18#http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-18"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:ont="http://www.co-ode.org/ontologies/ont.owl#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-18#&rdf;"
     xmlns:xml="http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-18#http://www.w3.org/XML/1998/namespace">
    <owl:Ontology rdf:about="http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-18#http://www.semanticweb.org/wiki/ontologies/2013/2/untitled-ontology-18"/>

    <!--
    ///////////////////////////////////////////////////////////////////////////////////////
    //
    // Annotation properties
    //
    ///////////////////////////////////////////////////////////////////////////////////////
     -->

    <!-- http://www.co-ode.org/ontologies/ont.owl#since -->

    <owl:AnnotationProperty rdf:about="&ont;since"/>

    <!--
    ///////////////////////////////////////////////////////////////////////////////////////
    //
    // Object Properties
    //
    ///////////////////////////////////////////////////////////////////////////////////////
     -->

    <!-- http://www.co-ode.org/ontologies/ont.owl#is_member -->

    <owl:ObjectProperty rdf:about="&ont;is_member">
        <rdf:type rdf:resource="&owl;InverseFunctionalProperty"/>
        <ont:since></ont:since>
    </owl:ObjectProperty>

    <!-- http://www.co-ode.org/ontologies/ont.owl#knows -->

    <owl:ObjectProperty rdf:about="&ont;knows">
        <ont:since></ont:since>
    </owl:ObjectProperty>

    <!-- http://www.co-ode.org/ontologies/ont.owl#members -->

    <owl:ObjectProperty rdf:about="&ont;members">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <ont:since></ont:since>
        <owl:inverseOf rdf:resource="&ont;is_member"/>
    </owl:ObjectProperty>
```

```
<!--
////////////////////////////////////////////////////////////////////////////////////////////////////
//
// Data properties
//
////////////////////////////////////////////////////////////////////////////////////////////////////
 -->

<!-- http://www.co-ode.org/ontologies/ont.owl#Age -->

<owl:DatatypeProperty rdf:about="&ont;Age">
   <rdfs:domain rdf:resource="&ont;People"/>
   <rdfs:domain>
      <owl:Restriction>
         <owl:onProperty rdf:resource="&ont;Age"/>
         <owl:allValuesFrom rdf:resource="&xsd;decimal"/>
      </owl:Restriction>
   </rdfs:domain>
</owl:DatatypeProperty>

<!-- http://www.co-ode.org/ontologies/ont.owl#type -->

<owl:DatatypeProperty rdf:about="&ont;type">
   <rdfs:domain>
      <owl:Restriction>
         <owl:onProperty rdf:resource="&ont;type"/>
         <owl:allValuesFrom rdf:resource="&rdfs;Literal"/>
      </owl:Restriction>
   </rdfs:domain>
</owl:DatatypeProperty>

<!--
////////////////////////////////////////////////////////////////////////////////////////////////////
//
// Classes
//
////////////////////////////////////////////////////////////////////////////////////////////////////
 -->

<!-- http://www.co-ode.org/ontologies/ont.owl#Club -->

<owl:Class rdf:about="&ont;Club"/>


<!-- http://www.co-ode.org/ontologies/ont.owl#People -->

<owl:Class rdf:about="&ont;People"/>

<!--
////////////////////////////////////////////////////////////////////////////////////////////////////
//
// Individuals
//
```

```
////////////////////////////////////////////////////////////////////////////////
 -->

<!-- http://www.co-ode.org/ontologies/ont.owl#Alice -->

<owl:NamedIndividual rdf:about="&ont;Alice">
   <rdf:type rdf:resource="&ont;People"/>
   <ont:Age>18</ont:Age>
   <ont:knows rdf:resource="&ont;Bob"/>
   <ont:is_member rdf:resource="&ont;Chess"/>
</owl:NamedIndividual>
<owl:Axiom>
   <ont:since>2001/10/03</ont:since>
   <owl:annotatedSource rdf:resource="&ont;Alice"/>
   <owl:annotatedTarget rdf:resource="&ont;Bob"/>
   <owl:annotatedProperty rdf:resource="&ont;knows"/>
</owl:Axiom>
<owl:Axiom>
   <ont:since>2005/07/01</ont:since>
   <owl:annotatedSource rdf:resource="&ont;Alice"/>
   <owl:annotatedTarget rdf:resource="&ont;Chess"/>
   <owl:annotatedProperty rdf:resource="&ont;is_member"/>
</owl:Axiom>

<!-- http://www.co-ode.org/ontologies/ont.owl#Bob -->

<owl:NamedIndividual rdf:about="&ont;Bob">
   <rdf:type rdf:resource="&ont;People"/>
   <ont:Age>22</ont:Age>
   <ont:knows rdf:resource="&ont;Alice"/>
   <ont:is_member rdf:resource="&ont;Chess"/>
</owl:NamedIndividual>
<owl:Axiom>
   <ont:since>2011/02/14</ont:since>
   <owl:annotatedSource rdf:resource="&ont;Bob"/>
   <owl:annotatedTarget rdf:resource="&ont;Chess"/>
   <owl:annotatedProperty rdf:resource="&ont;is_member"/>
</owl:Axiom>
<owl:Axiom>
   <ont:since>2001/10/04</ont:since>
   <owl:annotatedTarget rdf:resource="&ont;Alice"/>
   <owl:annotatedSource rdf:resource="&ont;Bob"/>
   <owl:annotatedProperty rdf:resource="&ont;knows"/>
</owl:Axiom>

<!-- http://www.co-ode.org/ontologies/ont.owl#Chess -->

<owl:NamedIndividual rdf:about="&ont;Chess">
   <rdf:type rdf:resource="&ont;Club"/>
   <ont:type>group</ont:type>
   <ont:members rdf:resource="&ont;Alice"/>
   <ont:members rdf:resource="&ont;Bob"/>
</owl:NamedIndividual>
```

```
  <owl:Axiom>
    <ont:since>2005/07/01</ont:since>
    <owl:annotatedTarget rdf:resource="&ont;Alice"/>
    <owl:annotatedSource rdf:resource="&ont;Chess"/>
    <owl:annotatedProperty rdf:resource="&ont;members"/>
  </owl:Axiom>
  <owl:Axiom>
    <ont:since>2011/02/14</ont:since>
    <owl:annotatedTarget rdf:resource="&ont;Bob"/>
    <owl:annotatedSource rdf:resource="&ont;Chess"/>
    <owl:annotatedProperty rdf:resource="&ont;members"/>
  </owl:Axiom>
</rdf:RDF>

<!-- Generated by the OWL API (version 3.4.2) http://owlapi.sourceforge.net -->
```


This example show that the sentence "*OWL and RDF are easy readable by humans*" is not the all truth. Linearizing the information is suitable solution for telecommunication and computer processing, but it is not easy understandable by humans.

Let remember multi-layer representation of the sample graph. What is presented in four rows below is the same as one presented on two (OWL) or four and half pages (RDF) above.

	space addresses		
file name	*Alice*	*Bob*	*Chess*
has_characteristics	**Alice - Age: 18**	**Bob - Age: 22**	**Chess - Type: Group**
knows	**Bob - since : 2001/10/03**	**Alice - since: 2001/10/04**	
members			**Alice - since: 2005/07/01; Bob - since: 2011/02/14**
is_member	**Chess - since: 2005/07/01**	**Chess - since: 2011/02/14**	

The main conclusion is that we still need new approaches for representing the knowledge which will correspond both to human and machine possibilities. Because of this, in addition to Protégé, ICON implements the NL-addressing features for storing the dictionaries, thesauruses and ontologies.

B2. SPARQL

The “Simple Protocol and RDF Query Language” (SPARQL) is a SQL-like language for querying RDF data. SPARQL allows querying for triples from an RDF database (or triple store). RDF doesn't use foreign and primary keys either. It uses URIs, the standard reference format for the World Wide Web. By using URIs, a triple store immediately has the potential to link to any other data in any triple store. That plays to the distributed strengths of the Web.

Because triple stores are large amorphous collections of triples, SPARQL queries by defining a template for matching triples, called a Graph Pattern. To get data out of the triple store using SPARQL, you need to define a pattern that matches the statements in the graph. Those will be questions like this: find me the subjects of all the statements that say 'plays guitar'. Example below shows a query over data defined using the ontology about music:

```
PREFIX : <http://aabs.purl.org/music#>
SELECT ?instrument
WHERE {:andrew :playsInstrument ?instrument }
```

The query says "find all the triples that have a subject of :andrew and a predicate of :playsInstrument, then get the objects of the matching triples and return them" [SPARQL, 2013].

Another example of a SELECT query follows.

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE { ?x foaf:name ?name .
        ?x foaf:mbox ?mbox . }
```

The first line defines namespace prefix, the last two lines use the prefix to express a RDF graph to be matched. Identifiers beginning with question mark ? identify variables. In this query, we are looking for resource ?x participating in triples with predicates foaf:name and foaf:mbox and want the subjects of these triples. Syntactic shortcuts of TURTLE can be used in the matching part.

In addition to specifying graph to be matched, constraints can be added for values using FILTER construct. An example of string value restriction is FILTER regex(?mbox, "company") that specifies regular expression query. An example of number value restriction is FILTER (?price < 20) that specifies that ?price must be less than 20. A few special operators are defined for the FILTER construct. They include isIRI for testing whether variable is IRI/URI, isLiteral for testing whether variable is literal, bound to test whether variable was bound and others - see the specification.

The matching part of the query may include OPTIONAL triples. If the triple to be matched is optional, it is evaluated when it is present, but the matching does not fail when it is not present. Optional sections may be nested. It is possible to make UNION of multiple matching graphs - if any of the graphs matches, the match will be returned as a result. The FROM part of the query is optional and may specify the RDF dataset on which query is performed.

The sequence of result may be modified using the following keywords with the meaning similar to SQL:

- ORDER BY - ordering by variable value;
- DISTINCT - unique results only;
- OFFSET - offset from which to show results;
- LIMIT - the maximum number of results.

There are four query result forms. In addition to the possibility of getting the list of values found it is also possible to construct RDF graph or to confirm whether a match was found or not.

- SELECT - returns the list of values of variables bound in a query pattern;
- CONSTRUCT - returns an RDF graph constructed by substituting variables in the query pattern;
- DESCRIBE - returns an RDF graph describing the resources that were found;
- ASK - returns a Boolean value indicating whether the query pattern matches or not.

The CONSTRUCT form specifies a graph to be returned with variables to be substituted from the query pattern, such as in the following example that will return graph saying that Alice knows last two people when ordered by alphabet from the given URI (the result in the RDF graph is not ordered, it is a graph and so the order of triples is not important).

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
CONSTRUCT { <http://example.org/person#Alice> foaf:knows ?x }
FROM <http://example.org/foaf/people>
WHERE { ?x foaf:name ?name }
ORDER BY desc(?name)
LIMIT 2
```

The DESCRIBE form will return information about matched resources in a form of an RDF graph. The exact form of this information is not standardized yet, but usually a blank node closure like, for example, Concise Bounded Description (CBD) is expected. In short, all the triples that have the matched resource in the object are returned; when a blank node is in the subject, then the triples in which this node participates as object are recursively added as well.

The ASK form is intended for asking yes/no questions about matching - no information about matched variables is returned, the result is only indicating whether matching exists or not [Obitko, 2007a].

B3. Storage characteristics of analyzed RDF triple stores

➢ *B3.1. DBMS based approaches*

✓ *3store*

3Store is MySQL based triple store, currently holding over 30 million RDF triples used by a range of Knowledgeable Services developed within the Advanced Knowledge Technologies project (AKT) [AKT Project, 2013] led by Nigel Shadbolt from Southampton. It is concerned with the management of the knowledge life cycle.

3store is a core C library that uses MySQL to store its raw RDF data and caches. The library offers OKBC and RDQL query interfaces, over HTTP (via an Apache web server module), or directly though the C library.

The server software itself does not expose any interfaces directly to the user, but it can be queried by a number of services, including a column based view and a direct RDF browser.

3store is distributed under the Gnu General Public License, and is available from Source forge [3storeSF, 2013].

✓ *Jena*

Jena is a Java toolkit for manipulating RDF models which has been developed by Hewlett-Packard Labs [Jena, 2013]. It has excellent support for RDQL queries, but does not provide an OKBC interface (however, given the level of RDQL support provided, the addition of an OKBC compatibility layer would be straightforward for the portion of the OKBC API that was implemented in the previous version of 3store).

Jena is a java framework for building semantic web applications. Jena implements APIs for dealing with Semantic Web building blocks such as RDF and OWL. Jena's fundamental class for users is the Model, an API for dealing with a set of RDF triples. A Model can be created from the file system or from a remote file. Using JDBC, it can also be tied to an existing RDBMS such as MySQL or PostgreSQL.

SDB is a component of Jena [CTS, 2012]. It provides for scalable storage and query of RDF datasets using conventional SQL databases for use in standalone applications, J2EE, and other application frameworks. The storage, as mentioned, is provided by an SQL database and many databases are supported, both open source and proprietary. An SDB store can be accessed and managed with the provided command line scripts and via the Jena API. In addition on the fine grain access provided by the Jena API, SDB can be coupled with the web-server - ‘Joseki’ - which is SPARQL query server. This enables an SDB store to be queried over HTTP. Jena recently introduced a non-transactional native store called TDB [CTS, 2012].

The attempt to import one of hyphen's larger RDF files (comprising around 5% of the data) into Jena, using its default MySQL back-end, had not completed after 24 hours it the import (the preliminary indications are that it repeatedly refreshes its database indexes during the import). From this experience, conclusion is that Jena is *unsuitable* for storing large volume of data [Harris & Gibbins, 2003].

✓ *RDFSuite*

RDF Suite is a set of high-level and scalable services enabling the realization of the full potential of the Semantic Web.

The RDF Suite is used by academics and software developers to produce scalable applications that rely on validating RDF/S compatible data for the Semantic Web.

RDF Suite supports the storage, query and update of semantic conceptualizations and thus can be used in order to store and better track and understand changes and evolution in the way users create and understand knowledge [RDF Suite, 2013].

✓ *Sesame*

Sesame is a de-facto standard framework for processing RDF data. This includes parsing, storing, inference and querying of/over such data. It offers an easy-to-use API that can be connected to all leading RDF storage solutions [Sesame, 2012].

Sesame can be deployed on top of a variety of storage systems (relational databases, in-memory, file systems, keyword indexers, etc.), and offers a large scale of tools to developers to leverage the power of RDF and related standards. Sesame fully supports the SPARQL query language for expressive querying and offers transparent access to remote RDF repositories using the exact same API as for local access. Finally, Sesame supports all main stream RDF file formats, including RDF/XML, Turtle, N-Triples, TriG and TriX.

Sesame is an open source framework for storage, inference and querying of RDF data. Sesame matches the features of Jena with the availability of a connection API, inference support, availability of a web server and SPARQL endpoint. Like Jena SDB it provides support for multiple backends like MySQL and Postsgre.

✓ *4store*

4store is an RDF database that was designed by Steve Harris and developed at "Garlik" Co. to underpin their Semantic Web applications. It has been used by Garlik as their primary RDF platform for three years, and has proved itself to be robust and secure. 4store makes use of the Raptor and Rasqal libraries that have been developed for Redland [4store, 2013].

4store is a database storage and query engine that holds RDF data. 4store's main strengths are its performance, scalability and stability. It does not provide many features over and above RDF

storage and SPARQL queries, but if you are looking for a scalable, secure, fast and efficient RDF store, then 4store should be on your shortlist.

- ***Oracle***

Oracle Spatial and Graph RDF Semantic Graph (Formerly Oracle Database Semantic Technologies) is an open, standards-based, scalable, secure, reliable, and performance RDF management platform. Based on a graph data model, RDF data (triples) are persisted, indexed and queried, like other object-relational data types [Oracle, 2013].

Application developers use the power of the Oracle Database to design and develop a wide range of semantic-enhanced business applications in areas that include intelligence, law enforcement, integrated bioinformatics and health care informatics, finance, web social network, and media, games, and content management.

Oracle provides the industry's leading spatial database management platform. Oracle Spatial and Graph option includes advanced features for spatial data and analysis as well as for physical, network and social graph applications. The geospatial data features support complex Geographic Information Systems (GIS) applications, enterprise applications and location-based services applications. The graph features include a network data model (NDM) graph to model and analyze link-node graphs to represent physical and logical networks used in industries such as transportation and utilities. In addition, Oracle Spatial and Graph includes support for RDF semantic graphs used in social networks and social interactions.

The RDF semantic graph feature of Oracles Spatial and Graph provides a robust and standards-based platform on which to build semantic solutions. Identifying the business requirements and benefits, project requirements, application functionality, and design considerations helps in planning and discussing the project with Oracle. This information can help Oracle provide recommendations and best practices to facilitate a successful project.

Storing RDF data in a relational database requires an appropriate table design. There are different approaches that can be classified in generic schemas, i.e. schemas that do not depend on the ontology, and ontology specific schemas.

Current Object-oriented databases (ORDBMS) provide the suitable facility which allows for a better modeling of the subclass and sub-property relationships [Broekstra, 2005; Alexaki et al, 2001].

DBMS "Oracle" offers another object-relational feature: an own data type to store RDF based on a graph data model [oracledb, 2012; OSTI, 2009]. RDF triples can be persisted, indexed and queried, similar to other object-relational data types.

Although the RDF model has several object-oriented characteristics and most RDF stores are internally working with an object model, approaches to store RDF data and schema information using object database management systems (ODBMS) are rarely known. (Object-) Relational databases are still predominant, when large amounts of data have to be persisted on a server and object databases did not and will most probably not replace them. However, new developments of ODBMS may show

some advantages over RDBMS in certain applications, e.g. for embeddable persistence solutions in mobile devices [Hertel et al, 2009].

A special attention has to be paid to the Oracle "Berkeley DB" as a tool for storing RDF information [Berkeley DB, 2012]. Oracle Berkeley DB is the industry-leading open source, embeddable storage engine that provides developers a fast, reliable, local database with zero administration. Oracle Berkeley DB is a library that links directly into your application. Your application makes simple function calls, rather than sending messages to a remote server, eliminating the performance penalty of client-server architectures.

Berkeley DB has a number of key advantages over comparable systems. It is simple to use, supports concurrent access by multiple users, and provides industrial-strength transaction support, including surviving system and disk crashes.

Berkeley DB supports three access methods: B+tree, Extended Linear Hashing (Hash), and Fixed- or Variable- length Records (Recno). All three operate on records composed of a key and a data value. In the B+tree and Hash access methods, keys can have arbitrary structure. In the Recno access method, each record is assigned a record number, which serves as the key. In all the access methods, the value can have arbitrary structure. The programmer can supply comparison or hashing functions for keys, and Berkeley DB stores and retrieves values without interpreting them. All of the access methods use the host file system as a backing store [Olson et al, 1999] (Figure 111).

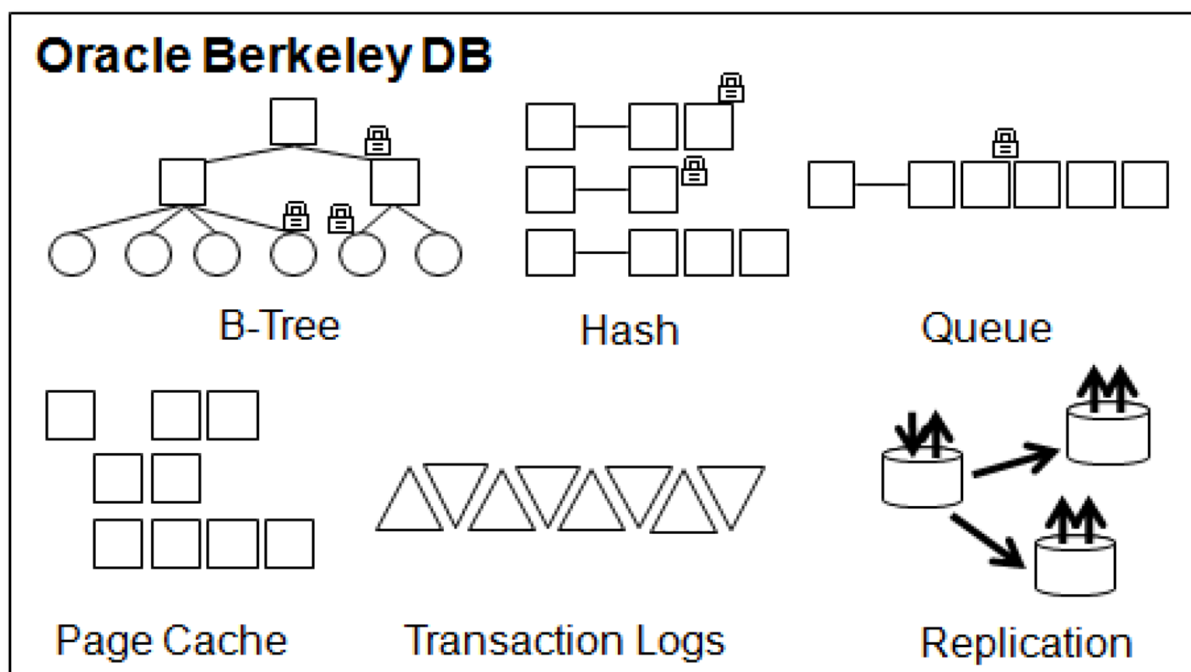

Figure 111. Main features of Oracle Berkeley DB

➢ *B3.2. Multiple indexing frameworks*

✓ *YARS*

YARS (Yet Another RDF Store) [YARS, 2013] is a data store for RDF in Java and allows for querying RDF based on a declarative query language, which offers a somewhat higher abstraction layer than the APIs of RDF toolkits such as Jena or Redland. YARS uses Notation3 as a way of encoding facts and queries.

The main requirement for YARS is to enable fast storage and retrieval of large amounts of RDF (in the order of millions of triples) while keeping a small footprint and a lightweight architecture approach.

There is a JDBC-like API for YARS available that can be used to issue calls either locally or via HTTP within Java programs. Active RDF is a library for accessing RDF data from within Ruby programs by addressing RDF resources, classes, properties, etc. programmatically, without queries. RDF2Go is an abstraction over triple (and quad) stores and has support for YARS as a backend store.

The YARS system combines methods from Information Retrieval and Databases to allow for better query answering performance over RDF data. It stores RDF data persistently by using six B+ tree indices. It not only stores the subject, the predicate and the object, but also the context information about the origin of the data. Each element of the corresponding quad (i.e., 4-uplet) is encoded in a dictionary storing mappings from literals and URIs to object IDs (OIDs-stored as number identifiers for compactness). To speed up keyword queries, the lexicon keeps an inverted index on string literals to allow fast full-text searches. In each B+ tree, the key is a concatenation of the subject, predicate, object and context. The six indices constructed cover all the possible access patterns of quads in the form (s, p, o, c) where c is the context of the triple (s, p, o). This representation allows fast retrieval of all triple access patterns. Thus, it is also oriented towards simple statement-based queries and has limitations for efficient processing of more complex queries. The proposal sacrifices space and insertion speed for query performance since, to retrieve any access pattern with a single index lookup, each triple is encoded in the dictionary six times, in different sorting order. Inference is not supported.

✓ *Kowari*

Kowari™ is an Open Source, massively scalable, transaction-safe, purpose-built database for the storage and retrieval of metadata [Kowari, 2004].

Much like a relational database, one stores information in Kowari and retrieves it via a query language. Unlike a relational database, Kowari is optimized for the storage and retrieval of many short statements (in the form of subject-predicate-object, like "Kowari is fun" or "Kowari imports RDF"). Kowari is not based on a relational database due to the large numbers of table joins encountered by relational systems when dealing with metadata. Instead, Kowari is a completely new database optimized for metadata management.

Kowari is implemented in the Java programming language and is 100% Java. It depends on standard Java packages available from Sun Microsystems^{TM}. Kowari also includes Java code from other projects. Details may be found on the Legal page [Kowari, 2004].

The Kowari system uses an approach similar to YARS. Indeed, the RDF statements are also stored as quads in which the first three items form a standard RDF triple and the fourth describes in which model the statement appears. The approach also uses six different orderings of quad elements acting as a compound index, and independently contains all the statements of the RDF store. In this ordering, the four quad elements can be arranged such that any collection of one to four elements can be used to find any matching statement or group of statements. However, Kowari uses a hybrid of AVL and B trees instead of B+ trees for multiple indexing purposes. Kowari solution also envisions simple statement-based queries like YARS.

✓ ***Virtuoso***

OpenLink Virtuoso is the first CROSS PLATFORM Universal Server to implement Web, File, and Database server functionality alongside Native XML Storage, and Universal Data Access Middleware, as a single server solution [Virtuoso, 2013]. Virtuoso is a native triple store available in both open source and commercial licenses. It provides command line loaders, a connection API, and support for SPARQL and web server to perform SPARQL queries and uploading of data over HTTP. A number of evaluations have tested virtuoso and found it to be scalable to the region of 1B+ triples.

The commercial system Virtuoso stores quads combining a graph to each triple (s, p, o). It, thus, conceptually stores the quads in a triples table expanded by one column. The columns are **g** for graph, **p** for predicate, **s** for subject and **o** for object. While technically rooted in an RDBMS, it closely follows the model of YARS but with fewer indices. The quads are stored in two covering indices, (g, s, p, o) and (o, g, p, s), where the URI's are dictionary encoded. Several further optimizations are added, including bitmap indexing. In this approach, the use of fewer indices tips the balance slightly towards insertion performance from query performance, but still favors query one.

✓ ***RDF-3X***

RDF-3X (RDF Triple eXpress) is the experimental RDF storage and retrieval system [Neumann & Weikum, 2008]. RDF-3X can import N-Triples/Turtle RDF data. RDF-3X is an RDF storage system with advanced indexes and query optimization that eliminates the need of physical database design by the use of exhaustive indexes for all permutations of subject-property-object triples.

RDF-3X uses a potentially huge triples table, with own storage implementation underneath (as opposed to using an RDBMS). It overcomes the problem of expensive self-joins by creating a suitable set of indexes. All the triples are stored in a compressed clustered B+ tree. The triples are sorted lexicographically in the B+ tree. The triple store is compressed by replacing long string literals

in the triples IDs using a mapping dictionary. The system supports both individual update operations and entire batches updates.

✓ *Hexastore*

Hexastore [Weiss et al, 2008] takes also a similar approach to YARS. The framework is based on the idea of main-memory indexing of RDF data in a multiple-index framework. The RDF data is indexed in six possible ways, one for each possible ordering of the three RDF elements by individual columns. The representation is based on any order of significance of RDF resources and properties and can be seen as a combination of vertical partitioning and multiple indexing approaches. Two vectors are associated with each RDF element, one for each of the others two RDF elements (e.g., [subject, property] and [subject, object]). Moreover, lists of the third RDF element are appended to the elements in these vectors. Hence, a sextuple indexing schema is created. As [Weiss et al, 2008] point out in, the values for O in PSO and SPO are the same. So in reality, even though six tables are created only *five copies* of the data are really computed, since the object columns are duplicated. To limit the amount of storage needed for the URIs, Hexastore uses the typical dictionary encoding of the URIs and the literals, i.e. every URI and literal is assigned *a unique numerical identifier*. Hexastore provides efficient single triple pattern lookups, and also allows fast merge-joins for any pair of two triple patterns. However, space requirement of Hexastore is five times the space required for storing statement in a triples table. Hexastore favors query performance over insertion time passing over applications that requires efficient statement insertion. Updates and insertions operations affect all six indices, hence can be slow. Hexastore does not provide inference support. [Weiss et al, 2008] proposed an on-disk index structure/storage layout so that Hexastore performance advantages can be preserved. Additionally to their experimental evaluations, they show empirically that, in the context of RDF storage, their *vector storage schema provides significantly lower data retrieval times compared to B trees*.

✓ *RDFCube*

The system RDFCube [Matono et al, 2007] is a three-dimensional hash index designed for RDFPeers [Cai & Frank, 2004] which is a distributed RDF repository that efficiently search RDF triples. Each triple is stored by specifying its subject, predicate, or object as a key. The RDFCube storage schema consists of set of cubes of the same size called cells. Each of these cells contains a bit called existence flag indicating the presence or absence of triples mapped into the cell. During the processing of a query, by checking the existence flags of cells into which candidate answer triples are mapped, it is possible to know the existence of the triples before actually accessing remote nodes where the candidate answer triples are stored. This information helps reducing the amount of data that is transferred among nodes when processing a join query since it is possible to narrow down the candidate triples by using AND operator between existence flags bits and transfer only the actual

present candidate triples. However, using a DHT (Distributed Hash Table) for the indexation suffers from some problems such as freshness of data and security [Faye et al, 2012].

✓ ***BitMat***

BitMat [Atre et al, 2009] is a main-memory based bit-matrix structure for representing a large set of RDF triples with the idea to make the representation compact. Each RDF triple is considered as a 3-dimensional entity which conceptually gives rise to a single universal table holding all RDF triples. This last can be horizontally partitioned into multiple fragments based on the usage requirements. BitMat can be viewed as a 3-dimensional bit-cube, in which each cell is a bit representing a unique triple and denoting the presence or absence of that triple. For representing the bit-cube in memory, it is flattened in a 2-dimensional bit matrix. There are six ways of flattening a bit-cube into a BitMat. Each structure contributes to more efficient particular set of single-join queries. To deal with the inherent sparsity of BitMat, this latter is maintained as an array of bit-rows, where each row is a collection of all the triples having the same subject. The underlying goal is to represent large RDF triple-sets with a compact in-memory representation and supporting a scalable multi-join query execution. These queries are processed using bitwise AND, OR operations on the BitMat rows and the resulting triples are returned as another BitMat. *BitMat is designed to be mainly a read-only RDF triple storing system.* Dynamic insertion or deletion of RDF triples is not supported at present.

✓ ***Parliament***

Parliament [Kolas et al, 2009] is an open source triple store that is an improved version of DAMLDB. Parliament takes linked list style of approach. It uses BerkeleyDB for storing the URI values and then stores the triple of references in a linked list.

Parliament describes storage and indexing schema based on linked lists and memory-mapped files with a storage structure composed of three parts: the resource table, the statement table, and the resource dictionary.

The resource table is a single file of fixed-length records (sequentially numbered with numbers serving as ID of the corresponding resources), each of which representing a single resource or literal. This allows direct access to a record given its ID via simple array indexing. Each record has eight components:

- Three statement ID fields representing the first statements that contain this resource as a subject, predicate, and object, respectively;
- Three count fields containing the number of statements using this resource as a subject, predicate, and object, respectively;
- An offset used to retrieve the string representation of the resource;
- Bit-field flags encoding various attributes of the resource.

➢ *B3.3. Storage characteristics of outlined RDF triple stores*

Storage characteristics of outlined RDF triple stores are presented in Table 77.

Table 77. Storage characteristics of outlined RDF triple stores

Store	Triple Table	Property Table	Multi-indexing	DBMS	File	RAM	Update Support
3store [Harris & Gibbins, 2003]	√			√			
Jena [Jena2, 2012; Wilkinson et al, 2003]		√		√		√	√
RDFSuite [Alexaki et al, 2001]		√		√			√
Sesame [Sesame, 2012; Broekstra et al, 2002]		√		√	√	√	√
4store [Harris et al, 2009]		√		√			√
Oracle [Oracle, 2013]				√			
YARS [YARS, 2013]			√		√	√	√
Kowari [Wood et al, 2005]			√	√			
Virtuoso [Erling & Mikhailov, 2007]			√	√			√
RDF-3X [Neumann & Weikum, 2008]			√	√			√
Hexastore [Weiss et al, 2008]			√	√		√	
RDFCube [Matono et al, 2007]			√				
BitMat [Atre et al, 2009]			√			√	
Parliament [Kolas et al, 2009]			√		√		√