Prolog-based Infrastructure for RDF: Scalability and Performance

Jan Wielemaker[1], Guus Schreiber[2], and Bob Wielinga[1]

[1] University of Amsterdam
Social Science Informatics (SWI)
Roetersstraat 15, 1018 WB Amsterdam, The Netherlands
{jan,wielinga}@swi.psy.uva.nl
[2] Vrije Universiteit Amsterdam
Department of Computer Science
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
schreiber@cs.vu.nl

Abstract. The semantic web is a promising application-area for the Prolog programming language for its non-determinism and pattern-matching. In this paper we outline an infrastructure for loading and saving RDF/XML, storing triples, elementary reasoning with triples and visualization. A predecessor of the infrastructure described here has been used in various applications for ontology-based annotation of multimedia objects using semantic web languages. Our library aims at fast parsing, fast access and scalability for fairly large but not unbounded applications upto 40 million triples.

The RDF parser is distributed with SWI-Prolog under the LGPL Free Software licence. The other components will be added to the distribution as they become stable and documented.

1   Introduction

Semantic-web applications will require multiple large ontologies for indexing and querying. In this paper we describe an infrastructure for handling such large ontologies, This work was done on the context of a project on ontology-based annotation of multi-media objects to improve annotations and querying [13], for which we use the semantic-web languages RDF and RDFS. The annotations use a series of existing ontologies, including AAT [10], WordNet [8] and ULAN [14]. To facilitate this research we require an RDF toolkit capable of handling approximately 3 million triples efficiently on current desktop hardware. This paper describes the parser, storage and basic query interface for this Prolog-based RDF infrastructure. A practical overview using an older version of this infrastructure is in an XML.com article [9].

We have opted for a purely memory-based infrastructure for optimal speed. Our tool set can handle the 3 million triple target with approximately 300 Mb. of memory and scales to approximately 40 million triples on fully equipped 32-bit hardware. Although insufficient to represent "the whole web", we assume 40

million triples is sufficient for applications operating in a restricted domain such as annotations for a set of cultural-heritage collections.

This document is organised as follows. In Sect. 2 we describe and evaluate the Prolog-based RDF/XML parser. Section 3 discusses the requirements and candidate choices for a triple storage format. In Sect. 4 we describe the chosen storage method and the basic query engine. In Sect. 5 we describe the API and implementation for RDFS reasoning support. This section also illustrates the mechanism for expressing higher level queries. Section 7 describes visualisation tools to examine the contents of the database. Finally, Sect. 8 describes some related work.

Throughout the document we present metrics on time and memory resources required by our toolkit. Unless specified otherwise these are collected on a dual AMD 1600+ (approx. Pentium-IV 1600) machine with 2GB memory running SuSE Linux 8.1, gcc 3.2 and multi-threaded SWI-Prolog 5.1.11.[3] The software is widely portable to other platforms, including most Unix dialects, MS-Windows and MacOS X. Timing tests are executed on our reference data consisting of 1.5 million triples from WordNet, AAT and ULAN.

2   Parsing RDF/XML

The RDF/XML parser is the oldest component of the system. We started our own parser because the existing (1999) Java (SiRPAC[4]) and Pro Solutions Perl-based[5] parsers did not provide the performance required and we did not wish to enlarge the footprint and complicate the system by introducing Java or Perl components. The RDF/XML parser translates the output of the SWI-Prolog SGML/XML parser[6] into a Prolog list of triples using the steps summarised in Fig. 1. We illustrate these steps using an example from the RDF Syntax Specification document [5], which is translated by the SWI-Prolog XML parser into a Prolog term as described in Fig. 2.

We wanted our parser from the XML parse-tree to triples to reflect as closely as possible the RDF Syntax Specification to improve maintainability and simplify the implementation. The first parser-step uses a variation of DCG (Definite Clause Grammar), each ruleset translating a production from the Syntax Specification. Figure 3 shows part of the rules for *Production parseTypeCollection-PropertyElt*[7] into a Prolog term **collection**(*Elements*), where *Elements* holds an intermediate representation for the collection-elements. The body of the rules guiding this process consists of the term that must be matched, optionally followed by raw Prolog code between {. . . }, similar to DCG. The matched term can call rule-sets to translate a sub-term using a \ escape-sequence. In Fig. 3, the first rule (propertyElt) matches a term **element**(*Name, Attributes, Content*), iff
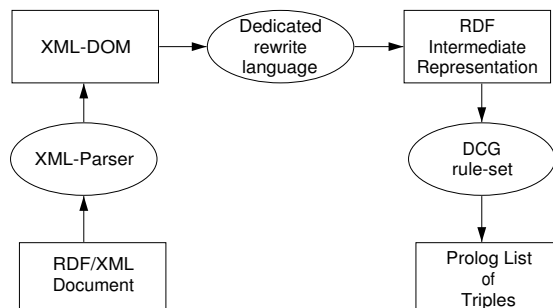
---

[3] http://www.swi-prolog.org
[4] http://www-db.stanford.edu/~melnik/rdf/api.html
[5] http://www.pro-solutions.com/rdfdemo/
[6] http://www.swi-prolog.org/packages/sgml2pl.html
[7] http://www.w3.org/TR/rdf-syntax-grammar/#parseTypeCollectionPropertyElt

Fig. 1. Steps converting an RDF/XML document into a Prolog list of triples.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:s="http://description.org/schema/">
  <rdf:Description about="http://www.w3.org/Home/Lassila">
     <s:Creator>Ora Lassila</s:Creator>
  </rdf:Description>
</rdf:RDF>
```

```
[ element('http://www.w3.org/1999/02/22-rdf-syntax-ns#':'RDF',
          [ xmlns:rdf = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#',
            xmlns:s = 'http://description.org/schema/'
          ],
          [ element('http://www.w3.org/1999/02/22-rdf-syntax-ns#':'Description',
                    [ about = 'http://www.w3.org/Home/Lassila'
                    ],
                    [ element('http://description.org/schema/':'Creator',
                              [],
                              [ 'Ora Lassila'
                              ])
                    ])
          ])
]
```

Fig. 2. Input RDF/XML document and output of the Prolog XML Parser, illustrating the input for the RDF parser

Attributes matches the attribute specification and *Content* can be matched by the `nodeElementList` rule-set. The final step uses traditional DCG to translate the intermediate structure into a flat list of triples.

```
propertyElt(Id, Name, collection(Elements), Base) ::=
        element(Name,
                \attrs([ \parseCollection,
                         \?idAttr(Id, Base)
                       ]),
                \nodeElementList(Elements, Base)).

parseCollection ::=
        \rdf_or_unqualified(parseType) = 'Collection'.

rdf_or_unqualified(Tag) ::=
        Tag.
rdf_or_unqualified(Tag) ::=
        NS:Tag,
        { rdf_name_space(NS), !
        }.
```

Fig. 3. Source code of the first phase, mapping the XML-DOM structure into a term derived from the RDF syntax specification. This fragment handles the `parseType=Collection` element

In the last phase the intermediate terms (e.g. **collection**(*Elements*) above) is translated into a list of triples using DCG rules.

Long documents cannot be handled this way as both the entire XML structure and the resulting list of RDF triples must fit on the Prolog stacks. To avoid this problem the XML parser can be operated in *streaming* mode and based on this mode the RDF parser handles RDF-Descriptions one-by-one, passing the resulting triples to a user-supplied Prolog goal.

2.1   Metrics and Evaluation

The source-code of the parser is 1170 lines, 564 for the first pass creating the intermediate state, 341 for the generating the triples and 265 for the driver putting it all together. The time to parse the WordNet sources are given in Tab. 1.

The parser passes the W3C RDF Test Cases[8]. In the current implementation however it does not handle the `xml:lang` tag nor RDF typed literals using `rdf:datatype`.

----

[8] http://www.w3.org/TR/2003/WD-rdf-testcases-20030123/

File	Size (Kb)	Time (sec)	Triples	Triples/Sec.
wordnet-20000620.rdfs	3	0.00	37	–
wordnet_glossary-20010201.rdf	14,806	10.64	99,642	9,365
wordnet_hyponyms-20010201.rdf	8,064	10.22	78,445	7,676
wordnet_nouns-20010201.rdf	9,659	13.84	273,644	19,772
wordnet_similar-20010201.rdf	1,763	2.36	21,858	9,262
Total	34,295	37.06	473,626	12,780

Table 1. Statistics loading WordNet

3   Storing RDF triples: requirements and alternatives

3.1   Requirement from integrating different ontology representations

Working with multiple ontologies created by different people and/or organizations poses some specific requirements for storing and retrieving RDF triples. We illustrate with an example from our own work on annotating images [12].

Given absence of official RDF versions of AAT and IconClass we created our own RDF representation, in which the concept hierarchy is modeled as an RDFS class hierarchy. We wanted to use these ontologies in combination with the RDF representation of WordNet created by Decker and Melnik[9]. However, their RDF Schema for WordNet defines classes and properties for the meta-model of WordNet. This means that WordNet *synsets* (the basic WordNet concepts) are represented as instances of the (meta)class `LexicalConcept` and that the WordNet hyponym relations (the subclass relations in WordNet) are represented as tuples of the metaproperty `hyponymOf` relation between instances of `wns:LexicalConcept`. This leads to a representational mismatch, as we are now unable to treat WordNet concepts as classes and WordNet hyponym relations as subclass relations.

Fortunately, RDFS provides metamodelling primitives for coping with this. Consider the following two RDF descriptions:

```
<rdf:Description rdf:about="&wns;LexicalConcept">
  <rdfs:subClassOf rdf:resource="&rdfs;Class"/>
</rdf:Description>

<rdf:Description rdf:about="&wns;hyponymOf">
  <rdfs:subPropertyOf rdf:resource="&rdfs;subClassOf"/>
</rdf:Description>
```

The first statement specifies that the class `LexicalConcept` is a subclass of the built-in RDFS metaclass `Class`, the instances of which are classes. This means that now all instances of `LexicalConcept` are also classes. In a similar vein, the second statement defines that the WordNet property `hyponymOf` is a

---
[9] http://www.semanticweb.org/library/

subproperty of the RDFS subclass-of relation. This enables us to interpret the instances of `hyponymOf` as subclass links.

We expect representational mismatches to occur frequently in any realistic semantic-web setting. RDF mechanisms similar to the ones above can be employed to handle this. However, this poses the requirement on the toolkit that the infrastructure is able to interpret subtypes of `rdfs:Class` and `rdfs:subPropertyOf`. In particular the latter was important for our applications, e.g., to be able to reason with WordNet hyponym relations as subclass relations or to visualize WordNet as a class hierarchy (cf. Fig. 9).

3.2   Requirements

Based on experiences we stated the following requirements for the RDF storage formate.

Efficient subPropertyOf handling As illustrated in Sect. 3.1, ontology-based annotation requires the re-use of multiple external ontologies. The `subPropertyOf` relation provides an ideal mechanism to re-interpret an existing RDF dataset.

Avoid frequent cache updates In our first prototype we used secondary store based on the RDFS data model to speedup RDFS queries. The mapping from triples to this model is not suitable for incremental update, resulting in frequent slow re-computation of the derived model from the triples as the triple set changes.

Scalability We anticipate the use of at least AAT, WordNet and ULAN in the next generation annotation tools. Together these require 1.5 million triples in their current form. We would like to be able to handle 3 million triples on a state-of-the-art notebook (512 MB).

Fast load/save RDF/XML parsing and loading time for the above ontologies is 108 seconds. This should be reduced using an internal format.

3.3   Storage options

The most natural way to store RDF triples is using facts of the format **rdf**(*Subject, Predicate, Object*) and this is, except for a thin wrapper improving namespace handling, the representation used in our first prototype. As standard Prolog systems only provide indexing on the first argument this implies that asking for properties of a subject is indexed, but asking about inverse relations is slow. Many queries involve reverse relations: "what are the sub-classes of X?". "what instances does Y have?", "what subjects have label L?" are queries commonly used on our annotation tool.

Our first tool solved these problems by building a secondary database following the RDFS datamodel. The cached relations included **rdfs_class**(*Class, Super, Meta*). **rdfs_property**(*Class, Property, Facet*), **rdf_instance**(*Resource, Class*) and **rdfs_label**(*Resource, Label*). These relations can be accessed quickly in any direction. This approach has a number of drawbacks. First of all, the

implications of even adding or deleting a single triple are potentially enormous, leaving the choice between complicated incremental synchronisation of the cache with the triple set or frequent slow total recompute of the cache. Second, storing the cache requires considerable memory resources and third there are many more relations that could profit from caching.

Using an external DBMS for the triple store is an alternative. Assuming some SQL database, there are three possible designs. The simplest one is to use Prolog reasoning and simple SELECT statements to query the DB. This approach does not exploit query optimization and causes many requests involving large amounts of data. Alternatively, one could either write a mixture of Prolog and SQL or automate part of this process, as covered by the Prolog to SQL converter of Draxler [3]. Our own (unpublished) experiences indicate a simple database query is at best 100 and in practice often over 1,000 times slower than using the internal Prolog database. Query optimization is likely to be of limited effect due to poor handling of transitive relations in SQL. Many queries involve rdfs:subClassOf, rdfs:subPropertyOf and other transitive relations. Using an embedded database such as BerkeleyDB[10] provides much faster simple queries, but still imposes a serious efficiency penalty. This is due to both the overhead of the formal database API and to the mapping between the in-memory Prolog atom handles and the resource representation used in the database.

In another attempt we used **Predicate**(*Subject, Object*) as database representation and stored the inverse relation as well in **InversePred**(*Object, Subject*) with a wrapper to call the 'best' version depending on the runtime instantiation. This approach, using native Prolog syntax for fast load/safe satisfies the requirements with minor drawbacks. The 3 million triples, the software and OS together require about 600MB of memory. Save/load using Prolog native syntax is, despite the fast SWI-Prolog parser, only twice as fast as parsing the RDF/XML.

In the end we opted for a Prolog *foreign-language* extension: a module written in C to extend the functionality of Prolog.[11] A significant advantage using an extension to Prolog rather than a language independent storage module separated by a formal API is that the extension can use native Prolog atoms, significantly reducing memory requirements and access time.

4   Realising an RDF store as C-extension to Prolog

4.1   Storage format

Triples are stored as a C-structure holding the three fields and 7 hash-table links for index access on all 7 possible instantiation patterns with at least one-field

---

[10] http://www.sleepycat.com/
[11] Extending Prolog using modules written in the C-language is provided in most to-days Prolog systems although there is no established standard foreign interface and therefore the connection between the extension and Prolog needs to be rewritten when porting to other implementation of the Prolog language [1].

instantiated. The size of the hash-tables is automatically increased as the triple set grows. In addition, each triple is associated with a *source-reference* consisting of an atom (normally the filename) and an integer (normally the line-number) and a general-purpose set of flags, adding to 13 machine words (52 bytes on 32-bit hardware) per triple, or 149 Mbytes for the intended 3 million triples. Our reference-set of 1.5 million triples uses 890,000 atoms. In SWI-Prolog an atom requires 7 machine words overhead excluding the represented string. If we estimate the average length of an atom representing a fully qualified resource at 30 characters the atom-space required for the 1.8 million atoms in 3 million triples is about 88 Mbytes. The required total of 237 Mbytes for 3 million triples fits easily in 512 Mbytes.

To accommodate active queries safely, deletion of triples is realised by flagging them as *erased*. Garbage collection can be invoked if no queries are active.

Indexing Subjects and resource *Objects* use the immutable atom-handle as hash-key. Literal *Objects* use a case-insensitive hash to speedup case-insensitive lookup of labels, a common operation in our annotation tool. The *Predicate* field needs special attention due to the requirement to handle `subPropertyOf` efficiently. The storage layer has an explicit representation for all known predicates which are linked directly in a hierarchy built using the `subPropertyOf` relation. Each predicate has a direct pointer to the *root* predicate: the topmost predicate in the hierarchy. If the top is formed by a cycle an arbitrary node of the cycle is flagged as the root, but all predicates in the hierarchy point to the same root as illustrated in Fig. 4. Each triple is now hashed using the root-predicate that belongs to the predicate of the triple.
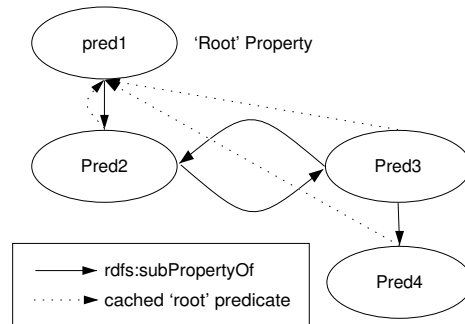
Fig. 4. All predicates are hashed on the root of the predicate hierarchy.

The above representation provides fully indexed lookup of any instantiation pattern, case insensitive on literals and including sub-properties. As a compromise to our requirements, the storage layer must know the fully qualified resource for `subPropertyOf` and must rebuild the predicate hierarchy and hash-tables if `subPropertyOf` relations are added to or deleted from the triple store. The pred-

icate hierarchy and index are invalidated if such a triple is added or deleted. The index is re-build on the first indexable query. We assume that changes to the conststubPropertyOf relations are infrequent.

4.2   Fast save/load format

Although attractive, the Prolog-only prototype has indicated that storing triples using the native representation of Prolog terms does not provide the required speedup, while the files are, mainly due to the expanded namespaces, larger than the RDF/XML source. An efficient format can be realised by storing the atom-text only the first time. Later references to the same atom simply store this as the N-th atom. A hash-table is used to keep track of the atoms already seen. An atom on the file thus has two formats: X $\langle integer \rangle$ or A $\langle length \rangle$ $\langle text \rangle$. Loading requires an array of already-loaded atoms. The resulting representation has the same size as the RDF/XML within 10%, and our reference dataset of 1.5 million triples is loaded 22 times faster, or 5 seconds.

4.3   Namespace handling

Fully qualified resources are long, hard to read and difficult to maintain in application source-code. On the other hand, representing resources as atoms holding the fully qualified resource is attractive because it is compact and compares very fast: the only test between two atoms as well as two resources is the equivalence test. Prolog optimises this test by ensuring there are no two atoms representing the same characters and therefore comparing atom-handles decides on equivalence.

   To merge as much as possible of the advantages the API described in Tab. 3 is encapsulated in a macro-expansion mechanism based on Prolog **goal_expansion/2** rules. For each of the arguments that can receive a resource a term of the format $\langle NS \rangle$:$\langle Identifier \rangle$, where $\langle NS \rangle$ is a registered abbreviation of a namespace and $\langle Identifier \rangle$ is a local name, is mapped to the fully qualified resource.[12] The predicate rdf_db:ns/2 maps registered short local namespace identifiers to the fully qualified namespaces. Declared as multifile, this predicate can be extended by the user. The initial definition contains the well-known abbreviations used in the context of the sematic web. See Tab. 2.

   With these declarations, we can write the following to get all individuals of `http://www.w3.org/2000/01/rdf-schema#Class` on backtracking:

   rdf(*X, rdf:type, rdfs:'Class'*)

---

[12] In our original prototype we provided a more powerful version of this mapping at runtime. In this version, output-arguments could be split into their namespace and local name as well. After examining actual use of this extra facility in the prototype and performance we concluded a limited compile-time alternative is more attractive.

rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
owl	http://www.w3.org/2002/7/owl#
xsd	http://www.w3.org/2000/10/XMLSchema#
dc	http://purl.org/dc/elements/1.1/
eor	http://dublincore.org/2000/03/13/eor#

Table 2. Initial registered namespace abbreviations

4.4   Performance evaluation

We studied two queries using our reference set. First we generated all solutions for **rdf**(*X, rdf:type, wns:'Noun'*). The 66025 nouns are generated in 0.0464 seconds (1.4 million alternatives/second). Second we asked for the type of randomly generated nouns. This deterministic query is executed at 526,000 queries/second. Tests comparing **rdf/3** with **rdf_has/4**, which exploits the `rdfs:subPropertyOf` relation show no significant difference in performance.

5   Querying and RDFS

Queries at the RDFS level are implemented using trivial Prolog rules exploiting the primitives in Tab. 3. For example, Fig. 5 realises testing and generating individuals. The first rule tests whether an individual belongs to a given class or generates all classes the individual belongs to. The second rule generates all individuals that belong to a specified class. The last rule is called in the unbound condition. There is not much point generating all classes and all individuals that have a type that is equal to or a subclass of the generated class and therefore we generate a standard Prolog exception.

```
rdfs_individual_of(Resource, Class) :-
        nonvar(Resource), !,
        rdf_has(Resource, rdf:type, MyClass),
        rdfs_subclass_of(MyClass, Class).
rdfs_individual_of(Resource, Class) :-
        nonvar(Class), !,
        rdfs_subclass_of(SubClass, Class),
        rdf_has(Resource, rdf:type, SubClass).
rdfs_individual_of(_Resource, _Class) :-
        throw(error(instantiation_error, _)).
```

Fig. 5. Implementation of **rdfs_individual_of/2**

rdf(*?Subject, ?Predicate, ?Object***)**

    Elementary query for triples. *Subject* and *Predicate* are atoms representing the fully qualified URL of the resource. *Object* is either an atom representing a resource or **literal(***Text***)** if the object is a literal value. For querying purposes, *Object* can be of the form **literal(***+Query, -Value***)**, where *Query* is one of

    exact(*+Text***)**

        Perform exact, but case-insensitive match. This query is fully indexed.

    substring(*+Text***)**

        Match any literal that contains *Text* as a case-insensitive substring.

    word(*+Text***)**

        Match any literal that contains *Text* as a 'whole word'.

    prefix(*+Text***)**

        Match any literal that starts with *Text*.

rdf_has(*?Subject, ?Predicate, ?Object, -TriplePred***)**

    This query exploits the `rdfs:subPropertyOf` relation. It returns any triple whose stored predicate equals *Predicate* or can reach this by following the transitive `rdfs:subPropertyOf` relation. The actual stored predicate is returned in *TriplePred*.

rdf_reachable(*?Subject, +Predicate, ?Object***)**

    True if *Object* is, or can be reached following the transitive property *Predicate* from *Subject*. Either *Subject* or *Object* or both must be specified. If one of *Subject* or *Object* is unbound this predicate generates solutions in breath-first search order. It maintains a table of visited resources, never generates the same resource twice and is robust against cycles in the transitive relation.

rdf_subject(*?Subject***)**

    Enumerate resources appearing as a subject in a triple. The reason for this predicate is to generate the known subjects *without duplicates* as one would get using **rdf(***Subject, _, _***)**. The storage layer ensures the first triple with a specified *Subject* is flagged as such.

rdf_assert(*+Subject, +Predicate, +Object***)**

    Assert a new triple into the database. *Subject* and *Predicate* are resources. *Object* is either a resource or a term **literal(***Value***)**.

rdf_retractall(*?Subject, ?Predicate, ?Object***)**

    Removes all matching triples from the database.

rdf_update(*+Subject, +Predicate, +Object, +Action***)**

    Replaces one of the three fields on the matching triples depending on *Action*:

    subject(*Resource***)**

        Changes the first field of the triple.

    predicate(*Resource***)**

        Changes the second field of the triple.

    object(*Object***)**

        Changes the last field of the triple to the given resource or **literal(***Value***)**.

Table 3. API summary for accessing the triple store

5.1   A few User-queries

Let us study the question 'Give me an individual of WordNet 'Noun' labeled *right*'. This non-deterministic query can be coded in two ways:

```
right_noun_1(R) :-
        rdfs_individual_of(R, wns:'Noun'),
        rdf_has(R, rdfs:label, literal(right)).

right_noun_2(R) :-
        rdf_has(R, rdfs:label, literal(right)),
        rdfs_individual_of(R, wns:'Noun').
```

The first query enumerates the subclasses of `wns:Noun`, generates their 66025 individuals and tests each for having the literal 'right' as label. The second generates the 8 resources in the 1.5 million triple set labeled 'right' and tests them to belong to `wns:Noun`. The first query requires 0.17 seconds and the second 0.37 milli-seconds to generate all alternatives.

A more interesting question is 'Give me a WordNet word that belongs to multiple lexical categories'. The program is shown in Fig. 6. The first **setof/3** generates the 123497 labels (a subproperty of `wns:wordForm`) defined in this WordNet version. Next we examine the labels one by one, generating the lexical categories and selecting the 6584 words that belongs to multiple categories. The query completes in 9.33 seconds after 2.27 million calls on **rdf_has/4** and **rdf_reachable/3**.

```
multi_cat(Label, CatList) :-
        setof(Label, wn_label(Label), Labels),
        member(Label, Labels),
        setof(Cat, lexical_category(Label, Cat), CatList),
        CatList = [_,_|_].

lexical_category(Label, Category) :-
        rdf_has(SynSet, rdfs:label, literal(Label)),
        rdfs_individual_of(SynSet, Category),
        rdf_has(Category, rdfs:subClassOf, wns:'LexicalConcept').

wn_label(Label) :-
        rdfs_individual_of(SynSet,  wns:'LexicalConcept'),
        rdf_has(SynSet, rdfs:label, literal(Label)).
```

Fig. 6. Finding all words that belong to multiple lexical categories

6   Declarativeness and optimisation

As we have seen above, query optimisation by ordering goals in a conjunction is required for good performance. Future versions could perform reordering of conjunctions based on the instantiation pattern and cardinality statistics of the predicates.

Many types of reasoning involve transitive relations such as `rdfs:subClassOf` which are allowed to contain cycles. Using normal Prolog non-determinism to follow a transitive chain as illustrated in Fig. 7 will not terminate under these circumstances. This problem can be handled transparently in Prolog systems that provide tabling [11]. We have choosen for a dedicated solution with **rdf_reachable/3** described in Tab. 3 to stay within standard Prolog.

```
is_subclass_of(Class, Class).
is_subclass_of(Class, Super) :-
        rdf_has(Class, rdfs:subClassOf, Super0),
        is_subclass_of(Super0, Super).
```

Fig. 7. Coding a transitive relation

7   Visualisation

For our annotation application we developed interactive editors. We are reorganising these into a clean modular design for building RDF/RDFS and OWL tools. The current toolkit provides a hierarchical browser with instance and class-view on resources and a tool to generate classical RDF diagrams. Both tools provide menus that exploit the registered source-information to view the origin of a triple in a text-editor. Currently these tools help developers to examine the content of the database. Figure 8 and Fig. 9 visualise the WordNet resource labeled *right* in one of its many meanings.

8   Related Work

Protege [4] is a modular Java-based ontology editor that can be extended using plugins. We regard Protege as complementary, providing interactive editing where we only provide simple interactive browsing. The Protege ontology language does not map one-to-one to RDFS, providing both extensions (e.g. cardinality) and limitations (notably in handling `subPropertyOf`). New versions of Protege and the introduction of OWL reduce this mismatch.
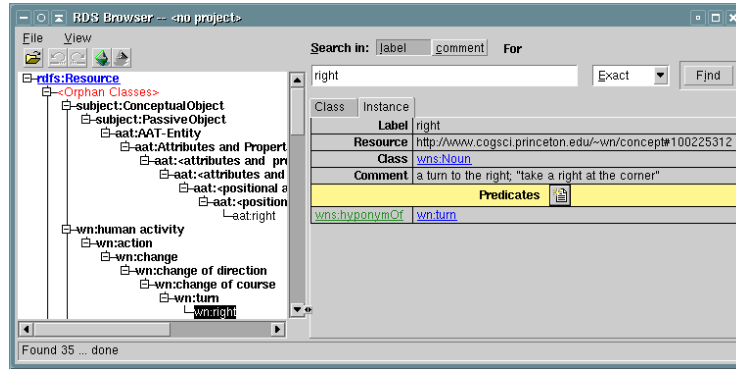
Fig. 8. The RDF browser after searching for *right* and selecting this term as a refinement of *turn*. The right tabbed-window can show a resource from various different viewpoints. This resource can be visualised as a generic resource or as a class.
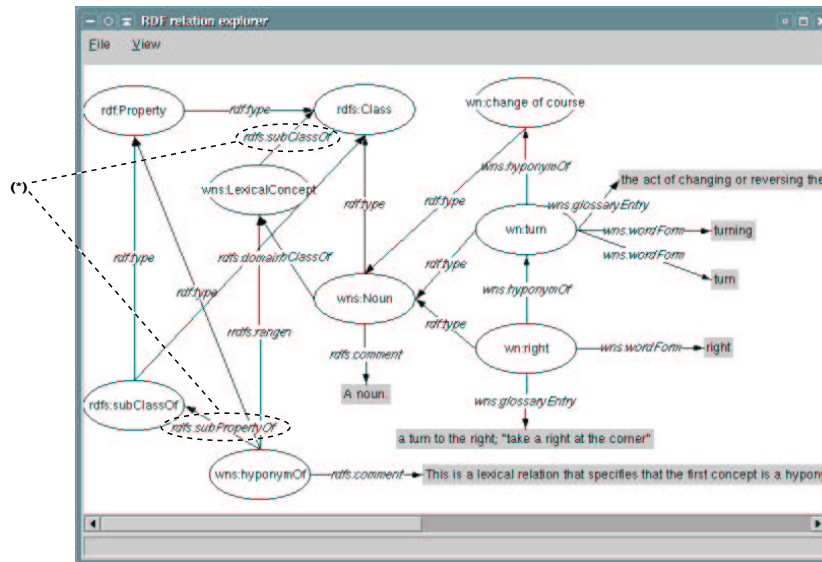

Fig. 9. From the browser we selected the *Diagram* option and expanded a few relations. The grey boxes represent literal values. The two marked relations turn WordNet into an RDFS class-hierarchy as explained in Sect. 3.1.

Jena [7] is a Java implementation for basic RDF handling. It aims at standard compliance and a friendly access from Java. Although its focus and coverage are slightly different the main difference is the choice of language.

Sesame [2] is an extensible Java-based architecture realising load/save of RDF/XML, modify the triple model and RQL [6] queries. It stresses a modular design where notably the storage module can be replaced. Although scalable, the modular approach with generic DBMS performs poorly (section 6.5 of [2]: Scalability Issues).

9 Discussion and Conclusions

We have outlined alternatives and an existing implementation of a library for handling semantic web languages in the Prolog language. We have demonstrated that this library can handle moderately large RDF triple sets (3 million) using 237 MB memory, ranging upto 40 million on 32-bit hardware providing a 3.5 GB address-space to applications. Further scaling either requires complicated segmentation of the store or hardware providing a larger (e.g. 64-bit) address-space. The library requires approx. 220 sec. to read 3 million triples from RDF/XML and 10 sec. from its proprietary file-format. Updating the `subPropertyOf` cache requires 3.3 sec. on this data-set. The library requires approx. 2 μs for the first answer and 0.7 μs for providing alternatives from the result-set through Prolog backtracking. All measurements on AMD Athlon 1600+ with 2 GB memory. The perfermance of indexed queries is constant with regard to the size of the triple set. The time required for not-indexed queries and cache-updates is proportional with the size of the triple set.

Declarativeness and optimisation by automatic reordering of conjunctions as discussed in Sect. 6 are omissions in the current design. Declarativeness can be achieved using tabling [11], a direction for which we should compare scalability and useability to dedicated solutions such as **rdf_reachable/3**. The use of an external database could provide query optimisation, but the lack of support for transitive relations is likely to harm optimisation and having two languages (Prolog and SQL) is a clear disadvantage. Automatic reordering of conjunctions of primitive RDF queries is a promising direction.

Experience with our first prototype has indicated that the queries required for our annotation and search process are expressed easily and concise in the Prolog language. We anticipate this infra structure is also suitable for the prototyping and implementation of end-user query langages.

References

1. Roberto Bagnara and Manuel Carro. Foreign language interfaces for Prolog: A terse survey. *ALP newsletter*, Mey 2002.

2. Jeen Broekstra and Arjohn Kampman. Sesame: A generic architecture for storing and querying RDF and RDF Schema. Technical Report OTK-del-10, Aidministrator Nederland bv, October 2001. URL: http://sesame.aidministrator.nl/publications/del10.pdf.

3. C. Draxler. Accessing relational and NF^2 databases through database set predicates. In Geraint A. Wiggins, Chris Mellish, and Tim Duncan, editors, *ALPUK91: Proceedings of the 3rd UK Annual Conference on Logic Programming, Edinburgh 1991*, Workshops in Computing, pages 156–173. Springer-Verlag, 1991.

4. W. E. Grosso, H. Eriksson, R. W. Fergerson, J. H. Gennari, S. W. Tu, and M. A. Musen. Knowledge modeling at the millennium: The design and evolution of Protégé-2000. In *12th Banff Workshop on Knowledge Acquisition, Modeling, and Management. Banff, Alberta*, 1999. URL: http://smi.stanford.edu/projects/protege (access date: 18 December 2000).

5. RDFCore Working Group. RDF/XML Syntax Specification (Revised)a. W3C Working Draft, World Wide Web Consortium, February 2003. http://www.w3.org/TR/rdf-syntax-grammar/.

6. G. Karvounarakis, V. Christophides, D. Plexousakis, and S. Alexaki. Querying community web portals. URL: http://www.ics.forth.gr/proj/isst/RDF/RQL/rql.html.

7. Brian McBride. Jena: Implementing the rdf model and syntax specification. 2001.

8. G. Miller. WordNet: A lexical database for english. *Comm. ACM*, 38(11), November 1995.

9. Bijan Parsia. RDF applications with Prolog. O'Reilly XML.com, 2001. http://www.xml.com/pub/a/2001/07/25/prologrdf.html.

10. T. Peterson. *Introduction to the Art and Architecture Thesaurus*. Oxford University Press, 1994. See also: http://www.getty.edu/research/tools/vocabulary/aat/.

11. I. V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S. Warren. Efficient tabling mechanisms for logic programs. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 697–714, Cambridge, June 13–18 1995. MIT Press.

12. A. Th. Schreiber. The web is not well-formed. *IEEE Intelligent Systems*, March/April 2002.

13. A. Th. Schreiber, B. Dubbeldam, J. Wielemaker, and B. J. Wielinga. Ontology-based photo annotation. *IEEE Intelligent Systems*, 16(3):66–74, May/June 2001.

14. ULAN: Union List of Artist Names. The Getty Foundation. URL: http://www.getty.edu/research/tools/vocabulary/ulan/, 2000.