Prolog-based RDF storage and retrieval

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 position paper we briefly describe our choices and results in dealing with RDFS. We also outline problems and possible directions for handling OWL.

1 Requirements

It is very hard to set requirements for handling RDF triples and the semantic web languages RDFS and OWL without an application domain in mind. Size as well as dynamic modification of the ABox and TBox as well as the reasoning performed (RDF, OWL/DL, OWL/Full or a subset thereof) have great impact on suitable technology. In our application domain, the annotation of multi-media objects using background ontologies we anticipate the use of approximately 1.5 million triples TBox and a similar amount annotations (ABox) for a realistic application. We wish to be able to run this application on hardware with 512 Mb core and reasonable application startup time. As we use several external ontologies in one application, the *meta-vocabulary* of these ontologies must be unified. This is accomplished using `rdfs:subPropertyOf` and therefore this relation must be handled efficiently.

2 Storage and indexing

After experiments with plain Prolog representations we have developed an RDF store using in-memory tables that are tightly integrated to the SWI-Prolog internals through the Prolog foreign language interface. The indexing of the in-memory store, which is implemented in the C-language to realise optimal memory use and performance, is optimised to deal with the semantics of the `rdfs:subPropertyOf` relation. *Subjects* and resource *Objects* use the immutable Prolog atom-handle as hash-key. Literal *Objects* use a case-insensitive hash to speedup case-insensitive lookup of labels, a common operation in our application domain. 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. 1. Each triple is now hashed using the root-predicate that belongs to the predicate of the triple.

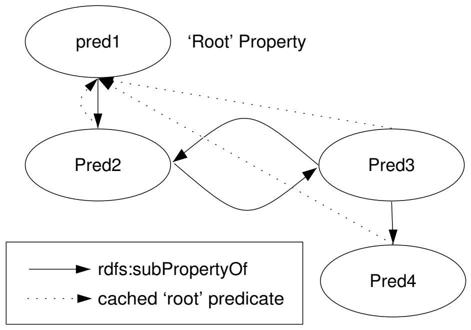

Fig. 1. All predicates are hashed on the root of the predicate hierarchy.

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 predicate 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 `subPropertyOf` relations are infrequent.

Application startup would be uncomfortably slow when reading the triples from RDF/XML. Therefore we use a binary format for caching purposes. The binary format load 22 times faster than the RDF/XML source, which is mainly caused by storing resource identifiers only once and therefore reducing the number of atom-lookup operations while loading the triples.

3 Query API

The store is designed to use Prolog as the query API, where complex queries are expressed as Prolog predicates using primitives provided by the store. A subset of the query primitive we currently provide are given in Tab. 1. This query API provides natural queries for RDFS.

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.

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.

Table 1. API summary for accessing the triple store

The main problem is that proper goal-ordering in a conjunction is important for good performance.

4 Scalability

We have tested memory requirements and performance using a test-set of 1.5 million triples originating from WordNet, [1] AAT [2] and ULAN [4] on desktop equipped with an AMD Athlon 1600+ and 2 GB memory. The key figures are summarised below.

- Memory requirements and scalability limits
  - $\pm$ 80 Mb per 10^6 triples
  - Max: $\pm 4 \times 10^7$ on 32-bit hardware
- Performance (AMD 1600+)
  - 2 μs first answer, 0.7 μs per alternative
  - Load RDF/XML: $\pm$ 12,500 triples/sec
  - Internal format: $\pm$ 300,000 triples/sec

5 Towards OWL

The current implementation support efficient handling of `rdfs:subPropertyOf` and to a lesser extend general transitive predicates. This is sufficient for RDFS, but cannot deal efficiently with reasoning anticipated in OWL. Instead of just handling transitive properties, the search involves `owl:sameAs`, `owl:inverseOf`, symmetric properties and many more. There are various options to deal with this explosion:

– Possibly some of these relations can be handled efficiently by extending the indexing of the store and/or perform some degree of forward reasoning. For example triples of symmetric predicates are hashed on the same key and therefore a search can exploit symmetric predicates at very low cost. By adjusting indexing we can make sure predicates and their inverse are hashed on the same key too. Low-level support for `owl:sameAs` is worth considering, though we anticipate that adding and deleting triples with this relation is frequent in our domain and therefore efficient incremental algorithms are needed for these operations.
– Tabling [3] is a Prolog extension to deal in a very dynamic way with normally not terminating recursive computation and avoid unnecessary recomputation. Although provided by only a few Prolog implementations, it may be worth considering.
– Constraint Logic Programming (CLP) is possible another interesting technique to improve declarativeness and performance, especially when reasoning about OWL descriptions.

References

1. G. Miller. WordNet: A lexical database for english. *Comm. ACM*, 38(11), November 1995.
2. T. Peterson. *Introduction to the Art and Architecture Thesaurus.* Oxford University Press, 1994. See also: http://www.getty.edu/research/tools/vocabulary/aat/.
3. 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.
4. ULAN: Union List of Artist Names. The Getty Foundation. URL: http://www.getty.edu/research/tools/vocabulary/ulan/, 2000.