



UvA-DARE (Digital Academic Repository)

Logic programming for knowledge-intensive interactive applications

Wielemaker, J.

Publication date
2009

[Link to publication](#)

Citation for published version (APA):

Wielemaker, J. (2009). *Logic programming for knowledge-intensive interactive applications*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

Chapter 3

Prolog-based Infrastructure for RDF: Scalability and Performance

About this chapter The core of this chapter has been published at the ISWC-03 (Wielemaker et al. 2003b). This paper has been updated with material from Wielemaker et al. 2007. We also provide an update of performance and scalability figures in section 3.6.

This paper elaborates on research question 1, investigating implementation alternatives for `rdf/3` and associated predicates. The discussion is continued in chapter 4 on query optimisation, while chapter 10 discusses its role in building a Semantic Web search tool.

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 in memory, and for elementary reasoning with triples. 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 20 million triples on 32-bit hardware or 300 million on 64-bit hardware with 64Gb main memory.

3.1 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 in the context of a project on ontology-based annotation of multi-media objects to improve annotating and querying (chapter 9, Schreiber et al. 2001), for which we use the Semantic Web languages RDF and RDFS. The annotations use a series of existing ontologies, including AAT (Peterson 1994), WordNet (Miller 1995) and ULAN (Getty 2000). 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 Parsia (2001).

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 20 million triples on 32-bit hardware. Scalability on 64-bit hardware is limited by available main memory and requires approximately 64Gb for 300 million triples. Although insufficient to represent “the whole web”, we assume 20 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 section 3.2 we describe and evaluate the Prolog-based RDF/XML parser. Section 3.3 discusses the requirements and candidate choices for a triple storage format. In section 3.4 we describe the chosen storage method and the basic query API. In section 3.5.1 we describe the API and implementation for RDFS reasoning support. This section also illustrates the mechanism for expressing higher level queries. Section 3.6 evaluates performance and scalability and compares the figures to some popular RDF stores.

3.2 Parsing RDF/XML

The RDF/XML parser is the oldest component of the system. We started our own parser because the existing (1999) Java (SiRPAC¹) and Pro Solutions Perl-based² 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 (see section 7.2) into a Prolog list of triples using the steps summarised in figure 3.1. We illustrate these steps using an example from the RDF Syntax Specification document (RDFCore WG 2003), which is translated by the SWI-Prolog XML parser into a Prolog term as described in figure 3.2.

The core of the translation is formed by the second step in figure 3.1, converting the XML DOM structure into an intermediate representation. The intermediate representation is a Prolog term that represents the RDF at a higher level, shielding details such as identifier generation for reified statements, rdf bags, blank nodes and the generation of linked lists from RDF collections from the second step. Considering the rather instable specification of RDF at the time this parser was designed, we aimed at an implementation where the code follows as closely as possible the structure of the RDF specification document.

Because the output of the XML parser is a nested term rather than a list we cannot use DCG. Instead, we designed a structure matching language in the spirit of DCGs, which we introduce with an example. Figure 3.3 shows part of the rules for the pro-

¹<http://www-db.stanford.edu/~melnik/rdf/api.html>

²<http://www.pro-solutions.com/rdfdemo/>

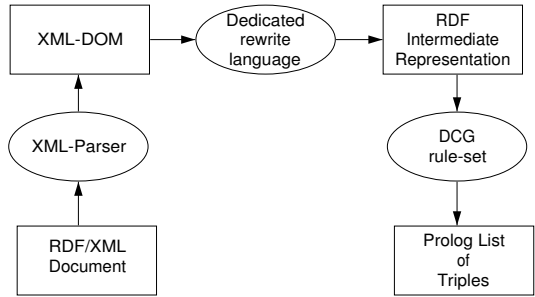


Figure 3.1: Steps converting an RDF/XML document into a Prolog list of triples.

```

<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' ])
    ]
  )
  ]
)
]
  
```

Figure 3.2: Input RDF/XML document and output of the Prolog XML Parser, illustrating the input for the RDF parser

duction *parseTypeCollectionPropertyElt*³ into a Prolog term (intermediate representation) *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 figure 3.3, the first rule (*propertyElt*) matches a term *element(Name, Attributes, Content)*, iff *Attributes* matches the attribute specification and *Content* can be matched by the *nodeElementList* rule-set.

³<http://www.w3.org/TR/rdf-syntax-grammar/#parseTypeCollectionPropertyElt>

The intermediate representation is translated into a list of `rdf(Subject, Predicate, Object)` terms using a set of traditional DCG rules.

```
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), !
    }.
```

Figure 3.3: Source code of the second step, mapping the XML-DOM structure into an intermediate representation derived from the RDF syntax specification. This fragment handles the `parseType=Collection` element

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. In this mode the RDF parser handles RDF-Descriptions one-by-one, passing the resulting triples to a user-supplied Prolog goal.

The source-code of the parser counts 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 parser passes the W3C RDF Test Cases⁴.

3.3 Storing RDF triples: requirements and alternatives

3.3.1 Requirement from integrating different ontology representations

Working with multiple ontologies created by different people and/or organisations poses some specific requirements for storing and retrieving RDF triples. We illustrate with an example from our own work on annotating images (chapter 9, Schreiber et al. 2001).

Given absence of official RDF versions of AAT and IconClass we created our own RDF representation, in which the concept hierarchy is modelled as an RDFS class hierarchy. We wanted to use these ontologies in combination with the RDF representation of WordNet created by Decker and Melnik.⁵ However, their RDF Schema for WordNet defines classes and

⁴<http://www.w3.org/TR/2003/WD-rdf-testcases-20030123/>

⁵<http://www.semanticweb.org/library/>

properties for the metamodel 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 meta property `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 metamodeling 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 way, the second statement defines that the WordNet property `hyponymOf` is a 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 is important for our applications.

3.3.2 Requirements

Based on our lessons learned from earlier annotation experiments as described in section 9.3 we state the following requirements for the RDF storage module.

Efficient subPropertyOf handling As illustrated in section 3.3.1, 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 At the time of design, we had access to 1.5 million triples in vocabularies. Together with actual annotations we aimed at storing 3 million triples on a (then) commodity notebook with 512 Mb main memory. Right now,⁶ we have access to over 30 million triples and we plan to support 300 million triples in commodity server hardware with 8 cores and 64 Gb main memory.

Fast load/save At the time of design, the RDF/XML parsing and loading time for 1.5 million triples was 108 seconds. This needs to be improved by an order of magnitude to achieve reasonable startup times, especially for interactive development.

3.3.3 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 in our annotation tool.

Our first tool (section 9.1) solved these problems by building a secondary Prolog database following the RDFS data model. 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 update 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, it is difficult to foresee for which derived relations caching is required because this depends on the structure and size of the triple set as well as the frequent query patterns.

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. Basic triple query is fast, but queries that involve an unknown predicate or need to use the predicate hierarchy (first requirement) cannot be handled efficiently. When using a native Prolog representation, we can use Prolog syntax for caching parsed RDF/XML files. Loading triples from Prolog syntax is approximately two times faster than RDF/XML, which is insufficient to satisfy our fourth requirement.

Using an external DBMS for the triple store is an alternative. Assuming an 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 optimisation

⁶November 2008

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 (1991). 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 optimisation 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 BerkeleyDB⁷ 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 RDF resource representation used in the database.

In the end we opted for a Prolog *foreign-language* extension: a module written in C to extend the functionality of Prolog.⁸ 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.

3.4 Realising an RDF store as C-extension to Prolog

3.4.1 Storage format

Triples are stored as a C-structure holding the three fields and 6 ‘next’ links, one for the linked list that represents all triples as a linear list and 5 hash-tables links. The 5 hash-tables cover all instantiation patterns with at least one field instantiated, except for all fields instantiated (+,+,+) and subject and object instantiated (+,-,+). Indexing of fully is less critical because a fully instantiated query never produces a choicepoint. Their lookup uses the (+,+,-) index. Subject and object instantiated queries use the (+,-,-) index.⁹ 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 up to 13 machine words (52 bytes on 32-bit hardware) per triple, or 149Mb 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 88Mb. The required total of 237Mb for 3 million triples fits in 512Mb.

⁷<http://www.sleepycat.com/>

⁸Extending Prolog using modules written in the C-language is provided in most today's 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 (Bagnara and Carro 2002).

⁹On our usage pattern this indexing schema performs good. Given that that database maintains statistics on the used indexing, we can consider to make existence and quality (size of the hash-table) dynamic in future versions.

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.

3.4.1.1 Indexing

Subjects and resource *Objects* use the immutable atom-handle as hash-key. The *Predicate* field needs special attention due to the requirement to handle `rdfs:subPropertyOf` efficiently. Each predicate is a first class citizen and is member of a *predicate cloud*, where each cloud represents a graph of predicates connected through `rdfs:subPropertyOf` relations. The cloud reference is used for indexing the triple. The cloud contains a reachability matrix that contains the transitive closure of the `rdfs:subPropertyOf` relations between the member predicates. The clouds are updated dynamically on assert and retract of `rdfs:subPropertyOf` triples. The system forces a re-hash of the triples if a new triple unites two clouds, both of which represent triples, or when deleting a triple splits a cloud in two non-empty clouds. As a compromise to our requirements, the storage layer must know the fully qualified resource for `rdfs:subPropertyOf` and must rebuild the predicate hierarchy and hash-tables if `rdfs:subPropertyOf` relations join or split non-empty predicate clouds. The index is re-build on the first indexable query. We assume that changes to the `rdfs:subPropertyOf` relations are infrequent.

RDF literals have been promoted to first class citizens in the database. Typed literals are supported using arbitrary Prolog terms as RDF object. All literals are kept in an AVL-tree, where

$$\text{numericliterals} < \text{stringliterals} < \text{termliterals}$$

. Numeric literals are sorted by value. String literals are sorted alphabetically, case insensitive and after removing UNICODE diacritics. String literals that are equal after discarding case and diacritics are sorted on UNICODE code-point. Other Prolog terms are sorted on Prolog standard order of terms. Sorted numeric literals are used to provide indexed search for dates. Sorted string literals are used for fast prefix search which is important for suggestions and disambiguation as-you-type with AJAX style interaction (chapter 10, Wielemaker et al. 2008). The core database provides indexed prefix lookup on the entire literals. Extended with the literal search facilities described above, it also supports indexed search on tokens and prefixes of tokens that appear in literals.

The literal search facilities are completed by means of *monitors*. Using `rdf_monitor(:Goal, +Events)` we register a predicate to be called at one or more given events. Monitors that trigger on literal creation and destruction are used to maintain a word-index for the literals as well as an index from stem to word and *metaphone* (Philips 2000) key to word.

The above representation provides fully indexed lookup using any instantiation pattern (mode) including sub-properties. Literal indexing is case insensitive and supports indexed prefix search as well as indexed search on ranges of numerical values.

3.4.2 Concurrent access

Multi-threading is supported by means of *read-write locks* and *transactions*. During normal operation, multiple readers are allowed to work concurrently. Transactions are realised using `rdf_transaction(:Goal, +Context)`. If a transaction is started, the thread waits until other transactions have finished. It then executes *Goal*, adding all write operations to an agenda. During this phase the database is not actually modified and other readers are allowed to proceed. If *Goal* succeeds, the thread waits until all readers have completed and updates the database. If *Goal* fails or throws an exception the agenda is discarded and the failure or error is returned to the caller of `rdf_transaction/2`. Note that the `rdf/3` transaction-based update behaviour differs from the multi-threaded SWI-Prolog logical update behaviour defined for dynamic predicates:

- *Prolog dynamic predicates*
In multi-threaded (SWI-)Prolog, accessing a dynamic predicate for read or write demands synchronisation only for a short time. In particular, readers with an open choice-point on the dynamic predicate allow other threads to update the same predicate. The standard Prolog logical update semantics are respected using time-stamps and keeping erased clauses around. Erased clauses are destroyed by a garbage collector that is triggered if the predicate has a sufficient number of erased clauses and is not in use.
- *RDF-DB transactions*
Multiple related modifications are bundled in a transaction. This is often desirable as many high-level (RDFS/OWL) changes involve multiple triples. Using transactions guarantees a consistent view of the database and avoids incomplete modifications if a sequence of changes is aborted.

3.4.3 Persistency and caching

Loading RDF from RDF/XML is slow, while quickly loading RDF databases is important to reduce application startup times. This is particularly important during program development. In addition, we need a persistent store to accumulate loaded RDF and updates such as human edits. We defined a binary file format that is used as a cache for loading external RDF resources and plays a role in the persistent storage.

Although attractive, storing triples using the native representation of Prolog terms (i.e., terms of the form `rdf(Subject, Predicate, Object)`) does not provide the required speedup, while the files are, mainly due to the expanded namespaces, larger than the RDF/XML source. Unpublished performance analysis of the Prolog parser indicates that most of the time is spent in parsing text to atoms used to represent RDF resources. Typically, the same resource appears in multiple triples and therefore we must use a serialisation that performs the expensive conversion from text to atom only once per unique resource. We use a simple incremental format that includes the text of the atom only the first time that the atom needs

to be saved. Later references to the same atom specify the atom as the N-th atom seen while loading this file. An atom on the file thus has two formats: $A \langle length \rangle \langle text \rangle$ or $X \langle integer \rangle$. Saving uses a hash table to keep track of already saved atoms, while loading requires an array of already-loaded atoms. The resulting representation has the same size as the RDF/XML within 10%, and loads approximately 20 times faster.

The predicate `rdf_load(+SourceURL)` can be configured to maintain a cache that is represented as a set of files in the above format.

A persistent backup is represented by a directory on the filesystem and one or two files per named graph: an optional *snapshot* and an optional *journal*. Both files are maintained through the monitor mechanism introduced in section 3.4.1.1 for providing additional indices for literals. If an initial graph is loaded from a file, a *snapshot* is saved in the persistent store using the fast load/save format described above. Subsequent addition or deletion of triples is represented by Prolog terms appended to the *journal* file. The journal optionally stores additional information passed with the transaction such as the time and user, thus maintaining a complete changelog of the graph. If needed, the current journal can be merged into the snapshot. This reduces disk usage and enhances database restore performance, but loses the history.

3.4.4 API rationale

The API is summarised in table 3.2. The predicates support the following datatypes:

- *Resource*
A fully qualified resource is represented by an atom. Prolog maps atoms representing a string uniquely to a handle and implements comparison by comparing the handles. As identity is the only operation defined on RDF resources, this mapping is perfect. We elaborate on namespace handling later in this section.
- `literal(Literal)`
A literal is embedded in a term `literal/1`. The literal itself can be any Prolog term. Two terms have a special meaning: `lang(LangID, Text)` and `type(TypeIRI, Value)`.

The central predicate is `rdf/3`, which realises a *pure*¹⁰ Prolog predicate that matches an edge in the RDF graph. In many cases it is desirable to provide a match specification other than simple exact match for literals, which is realised using a term `literal(+Query, -Value)` (see `rdf/3` in table 3.2). RDFS `subPropertyOf` entailment (requirement one of section 3.3.2) is realised by `rdf_has/4`.

¹⁰A pure predicate is a predicate that behaves consistently, regardless of the instantiation pattern. Conjunctions of pure predicate can be ordered freely without affecting the semantics.

Declarativeness Many types of reasoning involve transitive relations such as `rdfs:subClassOf` which are allowed to contain cycles. Normal Prolog non-recursion to express the transitive closure of a relation as illustrated in figure 3.4 (top) does not terminate if the relation contains cycles. Simple cases can be fixed easily by maintaining a set of visited resources. Complicated mutual recursive definitions can be handled transparently in Prolog systems that provide tabling (Ramakrishnan et al. 1995). Currently, we provide `rdf_reachable/3` which provides cycle-safe breadth-first search to simplify some of the coding (figure 3.4, bottom).

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

```
rdfs_subclass_of(Class, Super) :-
    rdf_reachable(Class, rdfs:subClassOf, Super).
```

Figure 3.4: Coding a transitive relation using standard Prolog recursion does not terminate (top) because most relations may contain cycles. The predicate `rdf_reachable/3` (bottom) explores transitive relations safely.

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 fast.

We unite the advantage of fully qualified atoms with the compactness in the source of `<NS>:<Identifier>` using macro-expansion based on Prolog `goal_expansion/2` rules. For each of the arguments that can receive a resource, a term of the format `<NS>:<Identifier>`, where `<NS>` is a registered abbreviation of a namespace and `<Identifier>` is a local name, is mapped to the fully qualified resource.¹¹ The predicate `rdf_db:ns/2` maps registered short local namespace identifiers to the fully qualified namespaces. The initial definition contains the well-known abbreviations used in the context of the Semantic Web. See table 3.1. The mapping can be extended using `rdf_register_ns/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').
```

¹¹In a prototype of this library 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 3.1: Initial registered namespace abbreviations

3.5 Querying the RDF store

3.5.1 RDFS queries

Queries at the RDFS level are provided as a library implemented using Prolog rules exploiting the API primitives in table 3.2. For example the code in figure 3.5 realises testing and generating of 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, _)).

```

Figure 3.5: Implementation of rdfs_individual_of/2

rdf (?Subject, ?Predicate, ?Object)	Elementary query for triples. <i>Subject</i> and <i>Predicate</i> are atoms representing the fully qualified URL of the resource. <i>Object</i> is either an atom representing a resource or <code>literal(Text)</code> if the object is a literal value. For querying purposes, <i>Object</i> can be of the form <code>literal(+Query, -Value)</code> , where <i>Query</i> is one of
exact (+Text)	Perform exact, but case-insensitive match. This query is fully indexed.
substring (+Text)	Match any literal that contains <i>Text</i> as a case-insensitive substring.
word (+Text)	Match any literal that contains <i>Text</i> as a 'whole word'.
prefix (+Text)	Match any literal that starts with <i>Text</i> .
rdf_has (?Subject, ?Predicate, ?Object, -TriplePred)	This query exploits the <code>rdfs:subPropertyOf</code> relation. It returns any triple whose stored predicate equals <i>Predicate</i> or can reach this by following the transitive <code>rdfs:subPropertyOf</code> relation. The actual stored predicate is returned in <i>TriplePred</i> .
rdf_reachable (?Subject, +Predicate, ?Object)	True if <i>Object</i> is, or can be reached following the transitive property <i>Predicate</i> from <i>Subject</i> . Either <i>Subject</i> or <i>Object</i> or both must be specified. If one of <i>Subject</i> or <i>Object</i> is unbound this predicate generates solutions in breadth-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 <i>without duplicates</i> as one would get using <code>rdf(Subject, → .)</code> . The storage layer ensures the first triple with a specified <i>Subject</i> is flagged as such.
rdf_transaction (:Goal, +Context)	Run <i>Goal</i> , recording all database modifying operations. Commit the operations if <i>Goal</i> succeeds, discard them otherwise. <i>Context</i> may associate additional information for the persistency layer as well as providing feedback to the user in interactive applications. See section 3.4.2.
rdf_assert (+Subject, +Predicate, +Object)	Assert a new triple into the database. <i>Subject</i> and <i>Predicate</i> are resources. <i>Object</i> is either a resource or a term <code>literal(Value)</code> .
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 <i>Action</i> :
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 <code>literal(Value)</code> .

Table 3.2: API summary for accessing the triple store

3.5.2 Application queries

We study the Prolog implementation of some queries on WordNet 1.6 (Miller 1995) in RDF as converted by Decker and Melnik to get some insight in how queries are formulated and what performance considerations must be taken into account. Timing is performed on an AMD 1600+ processor. Consider the question ‘Give me an individual of WordNet ‘Noun’ labelled *right*’. This non-deterministic query can be coded in two ways, which are semantically equivalent:

```
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 0.5 million triples with label ‘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 460× speedup. Query optimisation by ordering goals in a conjunction is required for good performance. Automatic reordering of conjunctions is discussed in chapter 4 (Wielemaker 2005) in the context of implementing the Semantic Web query languages `seRQL` (Broekstra et al. 2002) and `SPARQL` (Prud’hommeaux and Seaborne 2008) on top of this library.

3.5.3 Performance of `rdf/3` and `rdf_has/4`

The most direct way to describe the query performance is to describe the metrics of the core API functions: `rdf/3` and `rdf_has/4`. Other reasoning must be defined on top of these primitives and as we are faced with a large variety of potential tasks that can be fulfilled using a large variety of algorithms that are not provided with the `RDF-DB` library it is beyond the scope of this paper to comment on the performance for high-level queries.

We examined two queries using WordNet 1.6, executed on an AMD 1600+ CPU with 2Gb main memory. First we generated all solutions for `rdf(X, rdf:type, wns:'Noun')`. The 66,025 nouns are generated in 0.046 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. In summary, the time to setup a query and come with the first answer is approximately $2\mu\text{s}$ and the time to generate subsequent solutions is approximately $0.7\mu\text{s}$ per solution.

3.6 Performance and scalability comparison

Comparing the performance of RDF stores has several aspects. First, there is the data. Here we see a number of reference datasets,¹² such as the Lehigh University Benchmark (Guo et al. 2004). Second, there are the queries, which are closely related to the tasks for which we wish to use the RDF store. Examples of tasks are graph-exploration algorithms such as discussed in chapter 10, graph pattern matching as supported by SPARQL and (partial) OWL reasoning as supported by OWLIM (Kiryakov et al. 2005). Third, there is the different basic design which depends in part on the intended usage. We identified six design dimensions as listed below. This list has some overlap with section 3.3.3, but where the discussion in section 3.3.3 is already focused because of the requirements (section 3.3.2) and the choice for Prolog, the list below covers design choices made by a wider range of RDF stores which we can consider to include into our comparison.

- *Memory vs. disk-based*
Disk-based storage allows for much larger amounts of triples and can ensure short application startup times. However, access is several orders of magnitude slower. This can be compensated for by using caching, pre-computing results (forward chaining) and additional indexes. Because our aim is to provide a platform where we can prototype different reasoning mechanisms and support updates to the database, pre-computing is unattractive. Many RDF stores can be configured for either disk-based or memory-based operation. Because of the totally different performance curve and storage limits there is not much point comparing our memory-based infrastructure with disk-based implementations.
- *Node vs. relational model*
RDF can be stored as a graph in memory, where nodes link directly to their neighbours or as a table of RDF triples. Both these models can be used on disk and in memory. The SWI-Prolog RDF store uses the relational model. We have no information on the other stores.
- *Read-only vs. read/write*
Systems differ widely in the possibility to update the database. Some offer no update at all (e.g., BRAHMS, Janik and Kochut 2005); systems using forward reasoning provide much slower updates than systems doing backward reasoning only, notably on delete operations. Because annotation is one of our use cases, changes are not very frequent but must be supported. Read-only databases are generally faster and provide more compact storage because they can use fixed-size data structures (e.g., arrays) and require no locking for concurrent access.
- *Forward vs. backward reasoning*
Forward reasoning systems pre-compute (part of) the entailment and can therefore

¹²See also <http://esw.w3.org/topic/RdfStoreBenchmarking>

answer questions that require that entailment instantaneously. The price is a higher memory footprint, slower loading, slow updates and more difficult (or lack of) support to enable/disable specific entailment (see section 10.5.0.5). This model fits poorly with our use cases and the Prolog backward reasoning bias.

- *Built-in support for RDFS, OWL, etc.*
Especially stores using backward reasoning may have built-in support for specific inferences. Our store has special support for `rdfs:subPropertyOf` (`rdf.has/4`) and transitive properties (`rdf.reachable/3`). Others provide support for identity mapping using `owl:sameAs`.
- *Direct access to the triple store vs. query-language only*
Some stores can be deployed as a library which provided direct access to the triple store. Typically these stores provide an *iterator* that can be used to iterate over all triples in the store that match some pattern. This is comparable to our `rdf/3` predicate that iterates on backtracking over matching triples. Other stores only provide access through a query language such as SPARQL and some can be used in both modes.

Each value on each of the above six dimensions have advantages and disadvantages for certain classes of RDF-based applications, and it should therefore not come as a surprise that it is hard to compare RDF stores.

In 2005, Janik and Kochut (2005) compared the main-memory RDF store BRAHMS on load time, memory usage and two graph-exploration tasks to several RDF stores that provide a direct API to the triple store from the language in which they are implemented (see last bullet above). A direct API is important for their use case: searching RDF graphs for long associative relations. This use case compares well to the general purpose graph-exploration we aim at (see section 10.3) and therefore we decided to use the same RDF stores and datasets. The data (except for the ‘small synthetic’ mentioned in the paper) is all publically available. For the RDF stores we downloaded the latest stable version. We added SwiftOWLIM (OWLIM using main memory storage) because it is claimed¹³ to be a very fast and memory efficient store. Data, stores and versions are summarised in table 3.3.

Our hypothesis was that the hardware used Janik and Kochu (dual Intel Xeon@3.06Ghz) is comparable to ours (Intel X6800@2.93Ghz) and we could add our figures to the tables presented in their paper. After installing BRAHMS and re-running some of the tests we concluded this hypothesis to be false. For example, according to Janik and Kochut (2005), BRAHMS required 363 seconds to load the ‘big SWETO’ dataset while our timing is 37 seconds. BRAHMS search tests run 2-4 times faster in our setup with a significant variation. Possible explanations for these differences are CPU, compiler (gcc 3.2.2 vs. gcc 4.3.1), 32-bit vs. 64-bit version and system load. Dependency on system load cannot be excluded because examining the source for the timings on BRAHMS learned us that the reported time is wall time. Possible explanations for the much larger differences in load time are enhancements to the Raptor parser (the version used in the BRAHMS paper is unknown) and I/O

¹³<http://www.ontotext.com/owlim/OWLIMPres.pdf>

System	Remarks	
Jena version 2.5.6 (McBride 2001)		
Sesame version 2.2.1 (Broekstra et al. 2002)	Storage (<i>sail</i>): built-in memory	
Sesame version 2.2.1 (Broekstra et al. 2002)	Storage (<i>sail</i>): OWLIM 3.0beta	
Redland version 1.0.8 (Beckett 2002)	Parser: Raptor 1.4.18; in-memory trees	
BRAHMS version 1.0.1 (Janik and Kochut 2005)	Parser: Raptor 1.4.18	
SWI-Prolog version 5.7.2		
Dataset	file size	#triples
Small SWETO (Aleman-Meza et al. 2004)	14Mb	187,505
Big SWETO	244Mb	3,196,692
Univ(50,0) (Guo et al. 2004)	533Mb	6,890,961

Table 3.3: Systems and datasets used to evaluate load time and memory usage. The Raptor parser is part of the Redland suite described in the cited paper.

speed. Notably because time was measured as wall time differences between, for example local disk and network disk can be important.

Given availability of software and expertise to re-run the load time and memory usage experiments, we decided to do so using the currently stable version of all software and using the same platform for all tests. Section 3.6.1 describes this experiment.

All measurements are executed on an Intel core duo X6800@2.93Ghz, equipped with 8Gb main memory and running SuSE Linux 11.0. C-based systems where compiled with gcc 4.3.1 using optimisation settings as suggested by the package configure program and in 64-bit mode (unless stated otherwise); Java-based systems where executed using SUN Java 1.6.0 (64-bit, used -Xmx7G to specify 7Gb heap limit).

3.6.1 Load time and memory usage

Load time and memory usage put a limit on the scalability of main-memory-based RDF stores. Load time because it dictates the startup time of the application and memory usage because it puts a physical limit on the number of triples that can be stored. The amount of memory available to 32-bit applications is nowadays dictated by the address space granted to user processes by the OS and varies between 2Gb and 3.5Gb. On 64-bit machines it is limited by the amount of physical memory. In 2008, commodity server hardware scales to 64Gb; higher amounts are only available in expensive high-end hardware.

Table 3.4 shows both the load times for loading RDF/XML and for loading the proprietary cache/images formats. Redland was not capable of processing the Univ(50,0) dataset due to a ‘fatal error adding statements’. We conclude that our parser is the slowest of the tested ones; only by a small margin compared to Jena and upto five times slower compared to Raptor+BRAHMS. Based on analysing the SWI-Prolog profiler output we conclude that

the interpreted translation from XML to RDF triples (section 3.2) and especially the translation of resources to their final IRI format provides significant room for improvement. Janik and Kochut 2005 show extremely long load times for Univ(50,0) for both Jena and Sesame (Sesame: 2820 seconds for Univ(50,0) vs. 158 for big SWETO; nearly 18 times longer for loading only a bit over 2 times more data). A likely cause is garbage collection time because both systems were operating close to their memory limit on 32-bit hardware. We used a comfortable 7Gb Java heap limit on 64-bit hardware and did not observe the huge slowdown reported by Janik and Kochu. SWI-Prolog's infrastructure for loading RDF/XML is not subject to garbage collection because the parser operates in *streaming* mode and the RDF store provides its own memory management.

The SWI-Prolog RDF-DB cache format loads slower than BRAHMS images. One explanation is that BRAHMS internalises RDF resources as integers that are local to the image. In SWI-Prolog, RDF resources are internalised as Prolog atoms that must be resolved at load time. BRAHMS can only load exactly one image, while the SWI-Prolog RDF-DB can load any mixture of RDF/XML and cache files in any order.

Table 3.5 shows the memory usage each of the tested RDF stores. Results compare well to Janik and Kochu when considering the 32-bit vs. 64-bit versions of the used software. SWI-Prolog RDF-DB uses slightly more memory than BRAHMS, which is to be expected as BRAHMS is a read-only database, while SWI-Prolog's RDF-DB is designed to allow for updates. It is not clear to us why the other systems use so much more memory, particularly so because the forward reasoning of all systems has been disabled. Table 3.5 also provides the figures for the 32-bit version of SWI-Prolog. As all memory except for some flag fields and the resource string consists of pointers, memory usage is almost 50% lower.

	Small SWETO	Big SWETO	Univ(50,0)
Jena	7.4	120	180
Sesame	5.7	88	127
Sesame – OWLIM	4.1	63	104
Redland	3.1	66	–
BRAHMS – create image	1.8	37	69
BRAHMS – load image	0.1	1	1
SWI – initial	8.2	161	207
SWI – load cache	0.4	11	19
SWI – Triples/second	22,866	19,887	29,701

Table 3.4: RDF File load time from local disk in seconds. Reported time is CPU time, except for the BRAHMS case, which is wall time because we kept the original instrumentation. System load was low.

	Small SWETO	Big SWETO	Univ(50,0)
Jena	304	2669	2793
Sesame	141	2033	3350
Sesame – OWLIM	143	1321	2597
Redland	96	1514	–
BRAHMS	31	461	714
SWI	36	608	988
SWI – 32-bit	24	365	588

Table 3.5: Memory usage for RDF file load in Mb. Except for the last row, all systems were used in 64-bit mode.

3.6.2 Query performance on association search

Janik and Kochut 2005 evaluate the performance by searching all paths between two given resources, without considering paths to schema information, in an RDF graph. They use two algorithms for this: depth-first search (DFS) and bi-directional breadth first search (bi-BFS). We implemented both algorithms in Prolog on top of the SWI-Prolog RDF-DB store. Implementing DFS is straightforward, but the implementation of bi-BFS leaves more options. We implemented a faithful translation of the C++ bi-BFS implementation that is part of the examples distributed with BRAHMS.

DFS We replicated the DFS experiment with BRAHMS and SWI-Prolog using the software and hardware described on page 43. The results for the other stores are the values reported by Janik and Kochu, multiplied by the performance difference found between their report on BRAHMS and our measurements ($3.3\times$). We are aware that these figures are only a rough approximations. In the Prolog version, `rdf/3` is responsible for almost 50% of the CPU time. Memory requirements for DFS are low and the algorithm does not involve garbage collection. BRAHMS performs best on this task. Read-only access, no support for concurrent access and indexing that is optimised for this task are a few obvious reasons. For the other stores we see a large variation with SWI-Prolog on a second position after Sesame.

bi-BFS Similar to the load test and unlike the DFS test, the figures in Janik and Kochut 2005 on bi-BFS performance show large and hard-to-explain differences. Garbage collection and swapping¹⁴ could explain some of these, but we have no information on these details. We decided against replicating the association search experiment for all stores on the observation that for bi-BFS, only 5% of the time is spent on `rdf/3`. In other words, the bi-BFS tests are actually about comparing the implementation of bi-BFS in 4 different languages (C, C++,

¹⁴If all measurements are wall time. We confirmed this for BRAHMS by examining the source code but have no access to the source code used to test the other stores.

Association length	9	10	11	12
	<i>Measured</i>			
DFS SWI	1.6	2.1	7	28
DFS BRAHMS	0.1	0.1	0.9	2.2
	<i>Scaled down 3.3×</i>			
DFS Jena	23	32	53	–
DFS Sesame	0.6	0.6	3	16
DFS Redland	5	8	16	62
Found paths	47	61	61	61

Table 3.6: Search time on small SWETO in seconds. The numbers in the top half of the table are measured on the hardware described on page 43. The numbers on the bottom half of the table are scaled versions of the measurements by Janik and Kochu.

Java and Prolog). Implementation details do matter: our final version of bi-BFS is an order of magnitude faster than the initial version. Such optimisations are easily missed by non-experts in the programming language and/or the RDF store’s API.

We executed all bi-BFS experiments on SWI-Prolog and make some qualitative comments. All tests by Janik and Kochu were executed on the 32-bit versions of the RDF stores and many of the tests could not be executed by Jena, Sesame or Redland due to memory exhaustion. As SWI-Prolog memory usage is comparable to BRAHMS, even the 32-bit version can complete all tests. On smaller tests, SWI-Prolog is approximately 10 times slower than BRAHMS. In addition to the argument for DFS, an additional explanation for SWI-Prolog being slower than BRAHMS on bi-BFS can be found in sorting the bi-BFS agendas before joining the paths from both directions, a task on which our implementation spends 10-30% of the time. This step implies sorting integers (representing resources) for BRAHMS and sorting atoms for Prolog. Built-in alphabetical comparison of atoms is considerably slower, especially so because resources only differ after comparing the often equal XML namespace prefix. As the number of paths grows, SWI-Prolog slows down due to excessive garbage collection, particularly so because the current version of SWI-Prolog lacks an incremental garbage collector.

After applying the scale factor of 3.3 found with the DFS experiment, SWI-Prolog is significantly faster than the other systems in the comparison except for BRAHMS. For a fair experiment, the search algorithms must be implemented with involvement of experts in the programming language and the API of the store. Considering the low percentage of the time spent in fetching neighbouring nodes in the RDF graph, this would still largely be a cross-language and programming-skill competition rather than a competition between RDF stores.

3.7 Conclusions

We have outlined different ways to implement an RDF store in Prolog before describing our final implementation as a C library that extends Prolog. The library scales to approximately 20M triples on 32-bit hardware or 300M triples on a 64-bit machine with 64Gb main memory. The 300M triples can be loaded from the internal cache format in approximately 30 minutes, including concurrent rebuilding of the full text search index. The performance of indexed queries is constant with regard to the size of the triple set. The time required for not-indexed queries and re-indexing due to changes in the property hierarchy is proportional to the size of the triple set.

We compared our RDF store on load time, memory footprint and depth-first search for paths that link two resources with four other main memory RDF stores: BRAHMS, Jena, Sesame and Redland. BRAHMS performs best in all these tests, which can be explained because it is designed specially for this job. In particular, BRAHMS is read-only and single-threaded. Compared to Sesame, Jena and Redland, SWI-Prolog's RDF infrastructure has a slower parser (1.1 to 5 times), requires significantly less memory (3 to 8 times) and has competitive search performance (2 times slower to 7 times faster). Profiling indicates that there is significant room for improving the parser. Replacing our parser with Raptor (currently the fastest parser in our test-set) is also an option.

Experience has indicated that the queries required for our annotation and search process (see chapter 9 and 10) can be expressed concisely in the Prolog language. In chapter 4 (Wielemaker 2005) we show how conjunctions of RDF statements can be optimised for optimal performance as part of the implementation of standard RDF query languages (SeRQL and SPARQL) and making these accessible through compliant HTTP APIs. This optimisation is in part based on metrics about the triple set that is maintained by this library.

Acknowledgements

We would like to thank Maciej Janik of the BRAHMS project for his help in getting the datasets and clarifying the details of their bi-BFS implementation and Eyal Oren for his help with configuring Sesame and Jena for the evaluation. This work was supported by the ICES-KIS project "Multimedia Information Analysis" funded by the Dutch government and the MultimediaN¹⁵ project funded through the BSIK programme of the Dutch Government.

¹⁵www.multimedien.nl

