SWI-Prolog/XPCE Semantic Web Library

Jan Wielemaker
HCS,
University of Amsterdam
The Netherlands
E-mail: wielemak@science.uva.nl

March 14, 2006

Abstract

This document describes a library for dealing with standards from the W3C standard for the *Semantic Web*. Like the standards themselves (RDF, RDFS and OWL) this infrastructure is modular. It consists of Prolog packages for reading, querying and storing semantic web documents as well as XPCE libraries that provide visualisation and editing. The Prolog libraries can be used without the XPCE GUI modules. The library has been actively used with upto 10 million triples, using approximately 1GB of memory. Its scalability is limited by memory only. The library can be used both on 32-bit and 64-bit platforms.

Contents

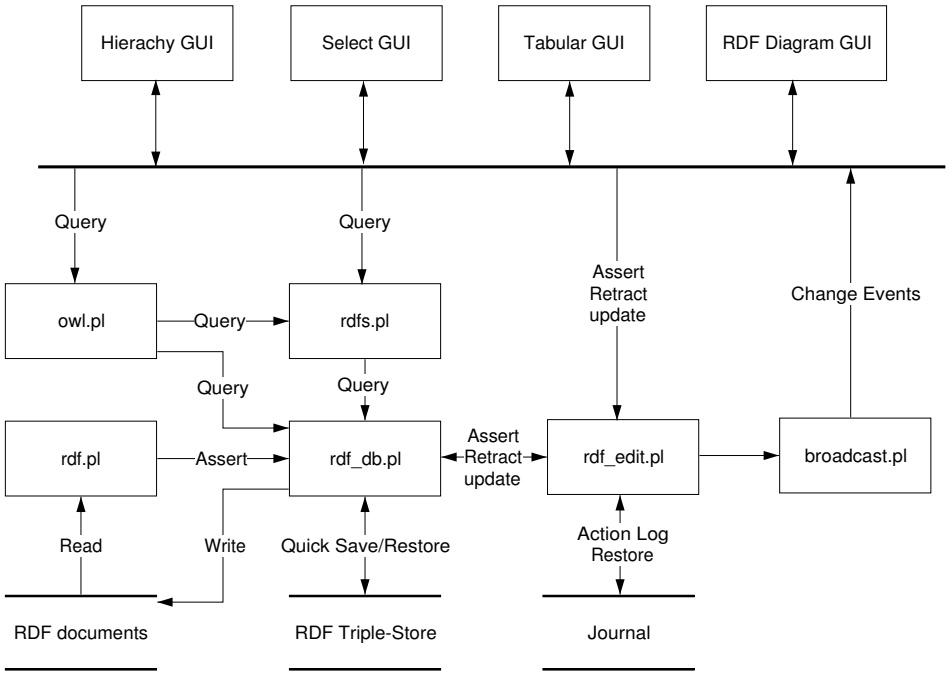

Figure 1: Modules for the Semantic Web library

1 Introduction

SWI-Prolog has started support for web-documents with the development of a small and fast SGML/XML parser, followed by an RDF parser (early 2000). With the `semweb` library we provide more high level support for manipulating semantic web documents. The semantic web is the likely point of orientation for knowledge representation in the future, making a library designed in its spirit promising.

2 Modules

Central to this library is the module `rdf_db.pl`, providing storage and basic querying for RDF triples. This triple store is filled using the RDF parser realised by `rdf.pl`. The storage module can quickly save and load (partial) databases. The modules `rdfs.pl` and `owl.pl` add querying in terms of the more powerful RDFS and OWL languages. Module `rdf_edit.pl` adds editing, undo, journaling and change-forwarding. Finally, a variety of XPCE modules visualise and edit the database. Figure figure 1 summarised the modular design.

3 Module rdf_db

The central module is called `rdf_db`. It provides storage and indexed querying of RDF triples. Triples are stored as a quintuple. The first three elements denote the RDF triple. *File* and *Line* provide information about the origin of the triple.

{*Subject Predicate Object File Line*}

The actual storage is provided by the *foreign language (C)* module `rdf_db.c`. Using a dedicated C-based implementation we can reduced memory usage and improve indexing capabilities.[1] Currently the following indexing is provided.

- Any of the 3 fields of the triple
- *Subject* + *Predicate* and *Predicate* + *Object*
- *Predicates* are indexed on the *highest property*. In other words, if predicates are related through `subPropertyOf` predicates indexing happens on the most abstract predicate. This makes calls to `rdf_has/4` very efficient.
- String literal *Objects* are indexed case-insensitive to make case-insensitive queries fully indexed. See `rdf/3`.

3.1 Query the RDF database

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`(*Value*) if the object is a literal value. If a value of the form *NameSpaceID* : *LocalName* is provided it is expanded to a ground atom using `expand_goal/2`. This implies you can use this construct in compiled code without paying a preformance penalty. See also section 3.5. Literal values take one of the following forms:

Atom

If the value is a simple atom it is the textual representation of a string literal without explicit type or language (`xml:lang`) qualifier.

lang(*LangID, Atom*)

Atom represents the text of a string literal qualified with the given language.

type(*TypeID, Value*)

Used for attributes qualified using the `rdf:datatype` *TypeID*. The *Value* is either the textual representation or a natural Prolog representation. See the option `convert_typed_literal`(*:Convertor*) of the parser. The storage layer provides efficient handling of atoms, integers (64-bit) and floats (native C-doubles). All other data is represented as a Prolog record.

For string querying purposes, *Object* can be of the form `literal`(+*Query, -Value*), where *Query* is one of the terms below. Details of literal matching and indexing are described in section 3.1.1.

exact(+*Text*)

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

[1] The orginal implementation was in Prolog. This version was implemented in 3 hours, where the C-based implementation costed a full week. The C-based implementation requires about half the memory and provides about twice the performance.

substring(+*Text*)
: Match any literal that contains *Text* as a case-insensitive substring. The query is not indexed on *Object*.

word(+*Text*)
: Match any literal that contains *Text* delimited by a non alpha-numeric character, the start or end of the string. The query is not indexed on *Object*.

prefix(+*Text*)
: Match any literal that starts with *Text*. This call is intended for *completion*. The query is not indexed on *Object*.

like(+*Pattern*)
: Match any literal that matches *Pattern* case insensitively, where the '*' character in *Pattern* matches zero or more characters.

Backtracking never returns duplicate triples. Duplicates can be retrieved using `rdf/4`.

rdf(*?Subject, ?Predicate, ?Object, ?Source*)
: As `rdf/3` but in addition return the source-location of the triple. The source is either a plain atom or a term of the format *Atom* : *Integer* where *Atom* is intended to be used as filename or URL and *Integer* for representing the line-number. Unlike `rdf/3`, this predicate does not remove duplicates from the result set.

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 recursive *subPropertyOf* relation. The actual stored predicate is returned in *TriplePred*. The example below gets all subclasses of an RDFS (or OWL) class, even if the relation used is not `rdfs:subClassOf`, but a user-defined sub-property thereof.[2]

```
subclasses(Class, SubClasses) :-
        findall(S, rdf_has(S, rdfs:subClassOf, Class), SubClasses).
```

Note that `rdf_has/4` and `rdf_has/3` can return duplicate answers if they use a different *TriplePred*.

rdf_has(*?Subject, ?Predicate, ?Object*)
: Same as `rdf_has`(*Subject, Predicate, Object,* _).

rdf_reachable(*?Subject, +Predicate, ?Object*)
: Is true if *Object* can be reached from *Subject* following the transitive predicate *Predicate* or a sub-property thereof. When used with either *Subject* or *Object* unbound, it first returns the origin, followed by the reachable nodes in breath-first search-order. It never generates the same node twice and is robust against cycles in the transitive relation. With all arguments instantiated it succeeds deterministically of the relation if a path can be found from *Subject* to *Object*. Searching starts at *Subject*, assuming the branching factor is normally lower. A call with both *Subject* and *Object* unbound raises an instantiation error. The following example generates all subclasses of `rdfs:Resource`:

[2]This predicate realises semantics defined in RDF-Schema rather than RDF. It is part of the `rdf_db` module because the indexing of this module incorporates the `rdfs:subClassOf` predicate.

```
?- rdf_reachable(X, rdfs:subClassOf, rdfs:'Resource').

X = 'http://www.w3.org/2000/01/rdf-schema#Resource' ;

X = 'http://www.w3.org/2000/01/rdf-schema#Class' ;

X = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#Property' ;

...
```

rdf_subject(*?Subject*)

Enumerate resources appearing as a subject in a triple. The main reason for this predicate is to generate the known subjects *without duplicates* as one gets using `rdf`(*Subject, _, _*).

3.1.1 Literal matching and indexing

Starting with version 2.5.0 of this library, literal values are ordered and indexed using a balanced binary tree (AVL tree). The aim of this index is threefold.

- Unlike hash-tables, binary trees allow for efficient *prefix* matching. Prefix matching is very useful in interactive applications to provide feedback while typing such as auto-completion.
- Having a table of unique literals we generate creation and destruction events (see `rdf_monitor/2`). These events can be used to maintain additional indexing on literals, such as 'by word'.
- A binary table allow for fast interval matching on typed numeric literals.[3]

As string literal matching is most frequently used for searching purposes, the match is executed case-insensitive and after removal of diacritics. Case matching and diacritics removal is based on Unicode character properties and independent from the current locale. Case conversion is based on the 'simple uppercase mapping' defined by Unicode and diacritic removal on the 'decomposition type'. The approach is lightweight, but somewhat simpleminded for some languages. The tables are generated for Unicode characters upto 0x7fff. For more information, please check the source-code of the mapping-table generator `unicode_map.pl` available in the sources of this package.

Currently the total order of literals is first based on the type of literal using the ordering

$$numeric < string < term$$

Numeric values (integer and float) are ordered by value, integers preceed floats if they represent the same value. strings are sorted alphabetically after case-mapping and diacritic removal as described above. If they match equal, uppercase preceeds lowercase and diacritics are ordered on their unicode value. If they still compare equal literals without any qualifier preceeds literals with a type qualifier which preceeds literals with a language qualifier. Same qualifiers (both type or both language) are sorted alphabetically.[4]

The ordered tree is used for indexed execution of `literal`(`prefix`*(Prefix), Literal*) as well as `literal`(`like`*(Like), Literal*) if *Like* does not start with a '*'. Note that results of quereies that are use the tree index are returned in alphabetical order.

[3] Not yet implemented

[4] The ordering defined above may change in future versions to deal with new queries for literals.

3.2 Predicate properties

The predicates below form an experimental interface to provide more reasoning inside the kernel of the rdb_db engine. Note that `symetric`, `inverse_of` and `transitive` are not yet supported by the rest of the engine.

rdf_current_predicate(*?Predicate*)
Enumerate all defined predicates. Behaves as the code below, but much more efficient.

```
rdf_current_predicate(Predicate) :-
        findall(P, rdf(_,P,_), Ps),
        sort(Ps, S),
        member(Predicate, S).
```

rdf_set_predicate(+*Predicate, +Property*)
Define a property of the predicate. Defined properties are listed with `rdf_predicate_property/2`.

rdf_predicate_property(*?Predicate, -Property*)
Query properties of a defined predicate. Currently defined properties are given below.

symmetric(*Bool*)
True if the predicate is defined to be symetric. I.e. {A} P {B} implies {B} P {A}.

inverse_of(*Inverse*)
True if this predicate is the inverse of *Inverse*.

transitive(*Bool*)
True if this predicate is transitive.

triples(*Triples*)
Unify *Triples* with the number of existing triples using this predicate as second argument. Reporting the number of triples is intended to support query optimization.

rdf_subject_branch_factor(*-Float*)
Unify *Float* with the average number of triples associated with each unique value for the subject-side of this relation. If there are no triples the value 0.0 is returned. This value is cached with the predicate and recomputed only after substantial changes to the triple set associated to this relation. This property is indented for path optimalisation when solving conjunctions of `rdf/3` goals.

rdf_object_branch_factor(*-Float*)
Unify *Float* with the average number of triples associated with each unique value for the object-side of this relation. In addition to the comments with the subject_branch_factor property, uniqueness of the object value is computed from the hash key rather than the actual values.

rdfs_subject_branch_factor(*-Float*)
Same as `rdf_subject_branch_factor/1`, but also considering triples of 'subPropertyOf' this relation. See also `rdf_has/3`.

rdfs_object_branch_factor(*-Float*)
Same as `rdf_object_branch_factor/1`, but also considering triples of 'subPropertyOf' this relation. See also `rdf_has/3`.

3.3 Modifying the database

As depicted in figure 1, there are two levels of modification. The `rdf_db` module simply modifies, where the `rdf_edit` library provides transactions and undo on top of this. Applications that wish to use the `rdf_edit` layer must *never* use the predicates from this section directly.

3.3.1 Modifying predicates

rdf_assert(+*Subject, +Predicate, +Object*)
Assert a new triple into the database. This is equivalent to `rdf_assert/4` using *SourceRef* `user`. *Subject* and *Predicate* are resources. *Object* is either a resource or a term `literal`(*Value*). See `rdf/3` for an explanation of *Value* for typed and language qualified literals. All arguments are subject to name-space expansion (see section 3.5).

rdf_assert(+*Subject, +Predicate, +Object, +SourceRef*)
As `rdf_assert/3`, adding *SourceRef* to specify the orgin of the triple. *SourceRef* is either an atom or a term of the format *Atom*:*Int* where *Atom* normally refers to a filename and *Int* to the line-number where the description starts.

rdf_retractall(*?Subject, ?Predicate, ?Object*)
Removes all matching triples from the database. Previous Prolog implementations also provided a backtracking `rdf_retract/3`, but this proved to be rarely used and could always be replaced with `rdf_retractall/3`. As `rdf_retractall/4` using an unbound *SourceRef*.

rdf_retractall(*?Subject, ?Predicate, ?Object, ?SourceRef*)
As `rdf_retractall/4`, also matching on the *SourceRef*. This is particulary useful to update all triples coming from a loaded file.

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*).

source(*Source*)
Changes the source location (*payload*). Note that updating the source has no consequences for the semantics and therefore the *generation* (see `rdf_generation/1`) is *not* updated.

rdf_update(+*Subject, +Predicate, +Object, +Source,+Action*)
As `rdf_update/4` but allows for specifying the source.

3.3.2 Transactions

The predicates from section 3.3.1 perform immediate and atomic modifications to the database. There are two cases where this is not desirable:

1. If the database is modified using information based on reading the same database. A typical case is a forward reasoner examining the database and asserting new triples that can be deduced from the already existing ones. For example, *if* $length(X) > 2$ *then size(X) is large*:

```
(   rdf(X, length, literal(L)),
    atom_number(L, IL),
    IL > 2,
    rdf_assert(X, size, large),
    fail
;   true
).
```

   Running this code without precautions causes an error because `rdf_assert/3` tries to get a write lock on the database which has an a read operation (rdf/3 has choicepoints) in progress.

2. Multi-threaded access making multiple changes to the database that must be handled as a unit.

Where the second case is probably obvious, the first case is less so. The storage layer may require reindexing after adding or deleting triples. Such reindexing operatations however are not possible while there are active read operations in other threads or from choicepoints that can be in the same thread. For this reason we added `rdf_transaction/2`. Note that, like the predicates from section 3.3.1, `rdf_transaction/2` raises a permission error exception if the calling thread has active choicepoints on the database. The problem is illustrated below. The `rdf/3` call leaves a choicepoint and as the read lock originates from the calling thread itself the system will deadlock if it would not generate an exception.

```
1 ?- rdf_assert(a,b,c).

Yes
2 ?- rdf_assert(a,b,d).

Yes
3 ?- rdf(a,b,X), rdf_transaction(rdf_assert(a,b,e)).
ERROR: No permission to write rdf_db 'default' (Operation would deadlock)
^  Exception: (8) rdf_db:rdf_transaction(rdf_assert(a, b, e)) ? no debug
4 ?-
```

rdf_transaction(*:Goal*)
   Same as

rdf_transaction(*Goal,* `user`)
   .

rdf_transaction(*:Goal, +Id*)
   After starting a transaction, all predicates from section 3.3.1 append their operation to the *transaction* instead of modifying the database. If *Goal* succeeds rdf_transaction cuts all choicepoints in *Goal* and executes all recorded operations. If *Goal* fails or throws an exception, all recorded operations are discarded and `rdf_transaction/1` fails or re-throws the exception.

On entry, `rdf_transaction/1` gains exclusive access to the database, but does allow readers to come in from all threads. After the successful completion of *Goal* `rdf_transaction/1` gains completely exclusive access while performing the database updates.

Transactions may be nested. Committing a nested transactions merges its change records into the outer transaction, while discarding a nested transaction simply destroys the change records belonging to the nested transaction.

The *Id* argument may be used to identify the transaction. It is passed to the begin/end events posted to hooks registered with `rdf_monitor/2`.

3.4 Loading and saving to file

The `rdf_db` module can read and write RDF-XML for import and export as well as a binary format built for quick load and save described in section 3.4.2. Here are the predicates for portable RDF load and save.

rdf_load(+*InOrList*)
Load triples from *In*, which is either a stream opened for reading, an atom specifying a filename or a list of valid inputs. This predicate calls `process_rdf/3` to read the source one description at a time, avoiding limits to the size of the input. If *In* is a file, `rdf_load/1` provides for caching the results for quick-load using `rdf_load_db/1` described below. Caching is activated by creating a directory `.cache` (or `_cache` on Windows) in the directory holding the `.rdf` files. Cached RDF files are loaded at approx. 25 times the speed of RDF-XML files.

rdf_load(+*FileOrList, +Options*)
As `rdf_load/1`, providing additional options. The options are handed to the RDF parser as implemented by `process_rdf/3`.

rdf_unload(+*Spec*)
Remove all triples loaded from *Spec*. *Spec* is either the name of a loaded database (see `rdf_source/1`) or a file specification. If *Spec* does not refer to a loaded database the predicate succeeds silently.

rdf_save(+*File*)
Save all known triples to the given *File*. Same as `rdf_save`(*File, []*).

rdf_save(+*File, +Options*)
Save with options. Provided options are:

db(+*FileRef*)
Save all triples whose file-part of their *SourceRef* matches *FileRef* to the given *File*. Saving arbitrary selections is possible using predicates from section 3.4.1.

anon(+*Bool*)
if `anon`(*false*) is provided anonymous resources are only saved if the resource appears in the object field of another triple that is saved.

convert_typed_literal(*:Converter*)
If present, raw literal values are first passed to *Converter* to apply the reverse of the `convert_typed_literal` option of the RDF parser. The *Converter* is called with

the same arguments as in the RDF parser, but now with the last argument instantiated and the first two unbound. A proper convertor that can be used for both loading and saving must be a logical predicate.

encoding(+*Encoding*)
Define the XML encoding used for the file. Defined values are `utf8` (default), `iso_latin_1` and `ascii`. Using `iso_latin_1` or `ascii`, characters not covered by the encoding are emitted as XML character entities (`&#...;`).

document_language(+*XMLLang*)
The value *XMLLang* is used for the `xml:lang` attribute in the outermost `rdf:RDF` element. This language acts as a default, which implies that the `xml:lang` tag is only used for literals with a *different* language identifier. Please note that this option will cause all literals without language tag to be interpreted using *XMLLang*.

rdf_source(*?File*)
Test or enumerate the files loaded using `rdf_load/1`.

rdf_make
Re-load all RDF sourcefiles (see `rdf_source/1`) that have changed since they were loaded the last time. This implies all triples that originate from the file are removed and the file is re-loaded. If the file is cached a new cache-file is written. Please note that the new triples are added at the end of the database, possibly changing the order of (conflicting) triples.

3.4.1 Partial save

Sometimes it is necessary to make more arbitrary selections of material to be saved or exchange RDF descriptions over an open network link. The predicates in this section provide for this. Character encoding issues are derived from the encoding of the *Stream*, providing support for `utf8`, `iso_latin_1` and `ascii`.

rdf_save_header(+*Stream, +Options*)
Save an RDF header, with the XML header, `DOCTYPE`, `ENTITY` and opening the `rdf:RDF` element with appropriate namespace declarations. It uses the primitives from section 3.5 to generate the required namespaces and desired short-name. *Options* is one of:

db(+*FileRef*)
Only search for namespaces used in triples labeled with *FileRef*.

namespaces(+*List*)
Where *List* is a list of namespace abbreviations (see section 3.5). With this option, the expensive search for all namespaces that may be used by your data is omitted. The namespaces `rdf` and `rdfs` are added to the provided *List*. If a namespace is not declared, the resource is emitted in non-abreviated form.

rdf_save_footer(+*Stream*)
Close the work opened with `rdf_save_header/2`.

rdf_save_subject(+*Stream, +Subject, +FileRef*)
Save everything known about *Subject* that matches *FileRef*. Using an variable for *FileRef* saves all triples with *Subject*.

rdf_quote_uri(*Q*)
uote a UNICODE *URI*. First the Unicode is represented as UTF-8 and then the unsafe characters are mapped to be represented as US-ASCII.

3.4.2 Fast loading and saving

Loading and saving RDF format is relatively slow. For this reason we designed a binary format that is more compact, avoids the complications of the RDF parser and avoids repetitive lookup of (URL) identifiers. Especially the speed improvement of about 25 times is worth-while when loading large databases. These predicates are used for caching by `rdf_load/[1,2]` under certain conditions.

rdf_save_db(+*File*)
Save all known triples into *File*. The saved version includes the *SourceRef* information.

rdf_save_db(+*File, +FileRef*)
Save all triples with *SourceRef FileRef*, regardless of the line-number. For example, using `user` all information added using `rdf_assert/3` is stored in the database.

rdf_load_db(+*File*)
Load triples from *File*.

3.4.3 MD5 digests

The `rdf_db` library provides for *MD5 digests*. An MD5 digest is a 128 bit long hash key computed from the triples based on the RFC-1321 standard. MD5 keys are computed for each individual triple and added together to compute the final key, resulting in a key that describes the triple-set but is independant from the order in which the triples appear. It is claimed that it is practically impossible for two different datasets to generate the same MD5 key. The Triple20 editor uses the MD5 key for detecting whether the triples associated to a file have changed as well as to maintain a directory with snapshots of versioned ontology files.

rdf_md5(+*Source, -MD5*)
Return the MD5 digest for all triples in the database associated to *Source*. The *MD5* digest itself is represented as an atom holding a 32-character hexadecimal string. The library maintains the digest incrementally on `rdf_load/[1,2]`, `rdf_load_db/1`, `rdf_assert/[3,4]` and `rdf_retractall/[3,4]`. Checking whether the digest has changed since the last `rdf_load/[1,2]` call provides a practical means for checking whether the file needs to be saved.

rdf_atom_md5(+*Text, +Times, -MD5*)
Computes the MD5 hash from *Text*, which is an atom, string or list of character codes. *Times* is an integer ≥ 1. When > 0, the MD5 algorithm is repeated *Times* times on the generated hash. This can be used for password encryption algorithms to make generate-and-test loops slow.

This predicate bears little relation to RDF handling. It is provided because the RDF library already contains the MD5 algorithm and semantic web services may involve security and consistency checking. This predicate provides a platform independant alternative to the `crypt` library provided with the `clib` package.

3.5 Namespace Handling

Prolog code often contains references to constant resources in a known XML namespace. For example, `http://www.w3.org/2000/01/rdf-schema#Class` refers to the most general notion of a class. Readability and maintability concerns require for abstraction here. The dynamic and multifile predicate rdf_db:ns/2 maintains a mapping between short meaningful names and namespace locations very much like the XML `xmlns` construct. The initial mapping contains the namespaces required for the semantic web languages themselves:

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

All predicates for the semweb libraries use `goal_expansion/2` rules to make the SWI-Prolog compiler rewrite terms of the form *Id* : *Local* into the fully qualified URL. In addition, the following predicates are supplied:

rdf_equal(*Resource1, Resource2*)
: Defined as *Resource1* = *Resource2*. As this predicate is subject to goal-expansion it can be used to obtain or test global URL values to readable values. The following goal unifies *X* with `http://www.w3.org/2000/01/rdf-schema#Class` without more runtime overhead than normal Prolog unification.

```
rdf_equal(rdfs:'Class', X)
```

rdf_register_ns(*+Alias, +URL*)
: Same as `rdf_register_ns`(*Alias, URL, []*).

rdf_register_ns(*+Alias, +URL, +Options*)
: Register *Alias* as a shorthand for *URL*. Note that the registration must be done before loading any files using them as namespace aliases are handled at compiletime through `goal_expansion/2`. If *Alias* already exists the default is to raise a permission error. If the option `force`(*true*) is provided, the alias is silently modified. Rebinding an alias must be done *before* any code is compiled that relies on the alias. If the option `keep`(*true*) is provided the new registration is silently ignored.

rdf_global_id(*?Alias:Local, ?Global*)
: Runtime translation between *Alias* and *Local* and a *Global* URL. Expansion is normally done at compiletime. This predicate is often used to turn a global URL into a more readable term.

rdf_global_object(*?Object, ?NameExpandedObject*)
: As `rdf_global_id/2`, but also expands the type field if the object is of the form `literal`(`type`*(Type, Value)*). This predicate is used for goal expansion of the object fields in `rdf/3` and similar goals.

rdf_global_term(+*Term0, -Term*)

Expands all *Alias*:*Local* in *Term0* and return the result in *Term*. Use infrequently for runtime expansion of namespace identifiers.

rdf_split_url(*?Base, ?Local, ?URL*)

Split a URL into a prefix and local part if used in mode -,-,+ or simply behave as `atom_concat/3` in other modes. The *URL* is split on the last # or / character.

3.5.1 Namespace handling for custom predicates

If we implement a new predicate based on one of the predicates of the semweb libraries that expands namespaces, namespace expansion is not automatically available to it. Consider the following code computing the number of distinct objects for a certain property on a certain object.

```
cardinality(S, P, C) :-
        (   setof(O, rdf_has(S, P, O), Os)
        ->  length(Os, C)
        ;   C = 0
        ).
```

Now assume we want to write `labels/2` that returns the number of distict labels of a resource:

```
labels(S, C) :-
        cardinality(S, rdfs:label, C).
```

This code will *not work* as `rdfs:label` is not expanded at compile time. To make this work, we need to add an `rdf_meta/1` declaration.

```
:- rdf_meta
        cardinality(r,r,-).
```

rdf_meta(*Heads*)

This predicate defines the argument types of the named predicates, which will force compile time namespace expansion for these predicates. *Heads* is a coma-separated list of callable terms. Defined argument properties are:

:

Argument is a goal. The goal is processed using `expand_goal/2`, recursively applying goal transformation on the argument.

+

The argument is instantiated at entry. Nothing is changed.

-

The argument is not instantiated at entry. Nothing is changed.

?

The argument is unbound or instantiated at entry. Nothing is changed.

@

The argument is not changed.

r
: The argument must be a resource. If it is a term ⟨*namespace*⟩:⟨*local*⟩ it is translated.

o
: The argument is an object or resource.

t
: The argument is a term that must be translated. Expansion will translate all occurences of ⟨*namespace*⟩:⟨*local*⟩ appearing anywhere in the term.

As it is subject to `term_expansion/2`, the `rdf_meta/1` declaration can only be used as a *directive*. The directive must be processed before the definition of the predicates as well as before compiling code that uses the rdf meta-predicates. The atom `rdf_meta` is declared as an operator exported from library `rdf_db.pl`. Files using `rdf_meta/1` *must* explicitely load `rdf_db.pl`.

Below are some examples from `rdf_db.pl`

```
:- rdf_meta
        rdf(r,r,o),
        rdf_source_location(r,-),
        rdf_transaction(:).
```

3.6 Monitoring the database

Considering performance and modularity, we are working on a replacement of the `rdf_edit` (see section 6) layered design to deal with updates, journalling, transactions, etc. Where the rdf_edit approach creates a single layer on top of rdf_db and code using the RDF database must select whether to use rdf_db.pl or rdf_edit.pl, the new approach allows to register *monitors*. This allows multiple modules to provide additional services, while these services will be used regardless of how the database is modified.

rdf_monitor(*:Goal, +Mask*)
: *Goal* is called for modifications of the database. It is called with a single argument that describes the modification. Defined events are:

  assert(*+S, +P, +O, +DB*)
  : A triple has been asserted.

  retract(*+S, +P, +O, +DB*)
  : A triple has been deleted.

  update(*+S, +P, +O, +DB, +Action*)
  : A triple has been updated.

  new_literal(*+Literal*)
  : A new literal has been created. *Literal* is the argument of `literal`(*Arg*) of the triple's object. This event is introduced in version 2.5.0 of this library.

  old_literal(*+Literal*)
  : The literal *Literal* is no longer used by any triple.

transaction(+*BeginOrEnd, +Id*)

Mark begin or end of the *commit* of a toplevel transaction started by `rdf_transaction/2`. *BeginOrEnd* is `begin` or `end`. *Id* is the second argument of `rdf_transaction/2`. The following transaction Ids are pre-defined by the library:

parse(*Id*)

A file is loaded using `rdf_load/2`. *Id* is one of `file`(*Path*) or `stream`(*Stream*).

unload(*DB*)

All triples with source *DB* are being unloaded using `rdf_unload/1`.

reset

Issued by `rdf_reset_db/0`.

load(+*BeginOrEnd, +Spec*)

Mark begin or end of `rdf_load_db/1` or load through `rdf_load/2` from a cached file. *Spec* is currently defined as `file`(*Path*).

rehash(+*BeginOrEnd*)

Marks begin/end of a re-hash due to required re-indexing or garbage collection.

Mask is a list of events this monitor is interested in. Default (empty list) is to report all events. Otherwise each element is of the form +Event or -Event to include or exclude monitoring for certain events. The event-names are the functor names of the events described above. The special name `all` refers to all events and `assert`(*load*) to assert events originating from `rdf_load_db/1`. As loading triples using `rdf_load_db/1` is very fast, monitoring this at the triple level may seriously harm performance.

This predicate is intended to maintain derived data, such as a journal, information for *undo*, additional indexing in literals, etc. There is no way to remove registered monitors. If this is required one should register a monitor that maintains a dynamic list of subscribers like the XPCE broadcast library. A second subscription of the same hook predicate only re-assignes the mask.

The monitor hooks are called in the order of registration and in the same thread that issued the database manipulation. To process all changes in one thread they should be send to a thread message queue. For all updating events, the monitor is called while the calling thread has a write lock on the RDF store. This implies that these events are processed strickly synchronous, even if modifications originate from multiple threads. In particular, the `transaction` *begin*, ... *updates* ... *end* sequence is never interleaved with other events. Same for `load` and `parse`.

3.7 Miscellaneous predicates

This section describes the remaining predicates of the `rdf_db` module.

rdf_node(-*Id*)

Generate a unique reference. The returned atom is guaranteed not to occur in the current database in any field of any triple.

rdf_bnode(-*Id*)

Generate a unique blank node reference. The returned atom is guaranteed not to occur in the current database in any field of any triple and starts with '__bnode'.

rdf_is_bnode(+*Id*)
Succeeds if *Id* is a blank node identifier (also called *anonymous resource*). In the current implementation this implies it is an atom starting with a double underscore.

rdf_source_location(+*Subject, -SourceRef*)
Return the source-location as *File*:*Line* of the first triple that is about *Subject*.

rdf_generation(-*Generation*)
Returns the *Generation* of the database. Each modification to the database increments the generation. It can be used to check the validity of cached results deduced from the database. Modifications changing multiple triples increment *Generation* with the number of triples modified, providing a heuristic for 'how dirty' cached results may be.

rdf_estimate_complexity(*R*)
eturn the number of alternatives as indicated by the database internal hashed indexing. This is a rough measure for the number of alternatives we can expect for an `rdf_has/3` call using the given three arguments. When called with three variables, the total number of triples is returned. This estimate is used in query optimisation. See also `rdf_predicate_property/2` and `rdf_statistics/1` for additional information to help optimisers.

rdf_statistics(*?Statistics*)
Report statistics collected by the `rdf_db` module. Defined values for *Statistics* are:

lookup(*?Index, -Count*)
Number of lookups using a pattern of instantiated fields. *Index* is a term `rdf`(*S,P,O*), where *S*, *P* and *O* are either + or −. For example `rdf`(+,+,-) returns the lookups with subject and predicate specified and object unbound.

properties(-*Count*)
Number of unique values for the second field of the triple set.

sources(-*Count*)
Number of files loaded through `rdf_load/1`.

subjects(-*Count*)
Number of unique values for the first field of the triple set.

literals(-*Count*)
Total number of unique literal values in the database. See also section 3.1.1.

triples(-*Count*)
Total number of triples in the database.

triples_by_file(*?File, -Count*)
Enumerate the number of triples associated to each file.

searched_nodes(-*Count*)
Number of nodes explored in `rdf_reachable/3`.

gc(-*Count, -Time*)
Number of garbage collections and time spent in seconds represented as a float.

rehash(-*Count, -Time*)
Number of times the hash-tables were enlarged and time spent in seconds represented as a float.

core(*-Bytes*)
Core used by the triple store. This includes all memory allocated on behalf of the library, but *not* the memory allocated in Prolog atoms referenced (only) by the triple store.

rdf_match_label(*+Method, +Search, +Atom*)
True if *Search* matches *Atom* as defined by *Method*. All matching is performed case-insensitive. Defines methods are:

exact
Perform exact, but case-insensitive match.

substring
Search is a sub-string of *Text*.

word
Search appears as a whole-word in *Text*.

prefix
Text start with *Search*.

like
Text matches *Search*, case insensitively, where the '*' character in *Search* matches zero or more characters.

rdf_reset_db
Erase all triples from the database and reset all counts and statistics information.

rdf_version(*-Version*)
Unify *Version* with the library version number. This number is, like to the SWI-Prolog version flag, defined as $10,000 \times Major + 100 \times Minor + Patch$.

3.8 Library rdf_litindex: Indexing words in literals

The library `semweb/rdf_litindex.pl` exploits the primitives of section 3.9 and the NLP package to provide indexing on words inside literal constants. It also allows for fuzzy matching using stemming and 'sounds-like' based on the *double metaphone* algorithm of the NLP package.

rdf_find_literals(*+Spec, -ListOfLiterals*)
Find literals (without type or language specification) that satisfy *Spec*. The required indices are created as needed and kept up-to-date using hooks registered with `rdf_monitor/2`. *Spec* is defined as:

and(*Spec1, Spec2*)
Intersection of both specifications.

or(*Spec1, Spec2*)
Union of both specifications.

not(*Spec*)
Negation of *Spec*. After translation of the full specification to *Disjunctive Normal Form* (DNF), negations are only allowed inside a conjunction with at least one positive literal.

case(*Word*)
: Matches all literals containing the word *Word*, doing the match case insensitive and after removing diacritics.

stem(*Like*)
: Matches all literals containing at least one word that has the same stem as *Like* using the Porter stem algorithm. See NLP package for details.

sounds(*Like*)
: Matches all literals containing at least one word that 'sounds like' *Like* using the double metaphone algorithm. See NLP package for details.

prefix(*Prefix*)
: Matches all literals containing at least one word that starts with Prefix, discarding diacritics and case.

Token
: Matches all literals containing the given token. See `tokenize_atom/2` of the NLP package for details.

rdf_token_expansions(+*Spec, -Expansions*)
: Uses the same database as `rdf_find_literals/2` to find possible expansions of *Spec*, i.e. which words 'sound like', 'have prefix', etc. *Spec* is a compound expression as in `rdf_find_literals/2`. *Expansions* is unified to a list of terms `sounds`(*Like, Words*), `stem`(*Like, Words*) or `prefix`(*Prefix, Words*). On compound expressions, only combinations that provide literals are returned. Below is an example after loading the ULAN[5] database and showing all words that sounds like 'rembrandt' and appear together in a literal with the word 'Rijn'. Finding this result from the 228,710 literals contained in ULAN requires 0.54 milliseconds (AMD 1600+).

```
?- rdf_token_expansions(and('Rijn', sounds(rembrandt)), L).

L = [sounds(rembrandt, ['Rambrandt', 'Reimbrant', 'Rembradt',
                        'Rembrand', 'Rembrandt', 'Rembrandtsz',
                        'Rembrant', 'Rembrants', 'Rijmbrand'])]
```

Here is another example, illustrating handling of diacritics:

```
?- rdf_token_expansions(case(cafe), L).

L = [case(cafe, [cafe, café])]
```

rdf_tokenize_literal(+*Literal, -Tokens*)
: Tokenize a literal, returning a list of atoms and integers in the range $-1073741824 \ldots 1073741823$. As tokenization is in general domain and task-dependent this predicate first calls the hook `rdf_litindex:tokenization`(*Literal, -Tokens*). On failure it calls `tokenize_atom/2` from the NLP package and deletes the following: atoms of length 1, floats, integers that are out of range and the english words `and`, `an`, `or`, `of`, `on`, `in`, `this` and `the`. Deletion first calls the hook `rdf_litindex:exclude_from_index`(*token, X*). This hook is called as follows:

[5] Unified List of Artist Names from the Getty Foundation.

```
no_index_token(X) :-
        exclude_from_index(token, X), !.
no_index_token(X) :-
        ...
```

3.9 Literal maps: Creating additional indices on literals

'Literal maps' provide a relation between literal values, intended to create additional indexes on literals. The current implementation can only deal with integers and atoms (string literals). A literal map maintains an ordered set of *keys*. The ordering uses the same rules as described in section 3.1.1. Each key is associated with an ordered set of *values*. Literal map objects can be shared between threads, using a locking strategy that allows for multiple concurrent readers.

Typically, this module is used together with `rdf_monitor/2` on the channals `new_literal` and `old_literal` to maintain an index of words that appear in a literal. Further abstraction using Porter stemming or Metaphone can be used to create additional search indices. These can map either directly to the literal values, or indirectly to the plain word-map. The SWI-Prolog NLP package provides complimentary building blocks, such as a tokenizer, Porter stem and Double Metaphone.

rdf_new_literal_map(-*Map*)
Create a new literal map, returning an opaque handle.

rdf_destroy_literal_map(+*Map*)
Destroy a literal map. After this call, further use of the *Map* handle is illegal. Additional synchronisation is needed if maps that are shared between threads are destroyed to guarantee the handle is no longer used. In some scenarios `rdf_reset_literal_map/1` provides a safe alternative.

rdf_reset_literal_map(+*Map*)
Delete all content from the literal map.

rdf_insert_literal_map(+*Map, +Key, +Value*)
Add a relation between *Key* and *Value* to the map. If this relation already exists no action is performed.

rdf_delete_literal_map(+*Map, +Key*)
Delete *Key* and all associated values from the map. Succeeds always.

rdf_delete_literal_map(+*Map, +Key, +Value*)
Delete the association between *Key* and *Value* from the map. Succeeds always.

rdf_find_literal_map(+*Map, +KeyList, -ValueList*)
Unify *ValueList* with an ordered set of values associated to all keys from *KeyList*. I.e. perform an *intersection* of the value-sets associated with the keys. Unifies *ValueList* with the empty list if no matches are found.

rdf_keys_in_literal_map(+*Map, +Spec, -Answer*)
Realises various queries on the key-set:

all
Unify *Answer* with an ordered list of all keys.

key(+*Key*)
Succeeds if *Key* is a key in the map and unify *Answer* with the number of values associated with the key. This provides a fast test of existence without fetching the possibly large associated value set as with `rdf_find_literal_map/3`.

prefix(+*Prefix*)
Unify *Answer* with an ordered set of all keys that have the given prefix. See section 3.1 for details on prefix matching. *Prefix* must be an atom. This call is intended for auto-completion in user interfaces.

ge(+*Min*)
Unify *Answer* with all keys that are larger or equal to the integer *Min*.

le(+*Max*)
Unify *Answer* with all keys that are smaller or equal to the integer *Max*.

between(+*Min, +Max*)
Unify *Answer* with all keys between *Min* and *Max* (including).

rdf_statistics_literal_map(+*Map, +Key(-Arg...)*)
Query some statistics of the map. Provides keys are:

size(-*Keys, -Relations*)
Unify *Keys* with the total key-count of the index and *Relation* with the total *Key-Value* count.

3.10 Issues with rdf_db

This RDF low-level module has been created after two year experimenting with a plain Prolog based module and a brief evaluation of a second generation pure Prolog implementation. The aim was to be able to handle upto about 5 million triples on standard (notebook) hardware and deal efficiently with `subPropertyOf` which was identified as a crucial feature of RDFS to realise fusion of different data-sets.

The following issues are identified and not solved in suitable manner.

`subPropertyOf` **of** `subPropertyOf` is not supported.

Equivalence Similar to `subPropertyOf`, it is likely to be profitable to handle resource identity efficient. The current system has no support for it.

4 Module rdfs

The `rdfs` library adds interpretation of the triple store in terms of concepts from RDF-Schema (RDFS). There are two ways to provide support for more high level languages in RDF. One is to view such languages as a set of *entailment rules*. In this model the rdfs library would provide a predicate `rdfs/3` providing the same functionality as `rdf/3` on union of the raw graph and triples that can be derived by applying the RDFS entailment rules.

Alternatively, RDFS provides a view on the RDF store in terms of individuals, classes, properties, etc., and we can provide predicates that query the database with this view in mind. This is the approach taken in the `rdfs.pl` library, providing calls like `rdfs_individual_of`(*?Resource, ?Class*).[6]

[6]The SeRQL language is based on querying the deductive closure of the triple set. The SWI-Prolog SeRQL library

4.1 Hierarchy and class-individual relations

The predicates in this section explore the `rdfs:subPropertyOf`, `rdfs:subClassOf` and `rdf:type` relations. Note that the most fundamental of these, `rdfs:subPropertyOf`, is also used by `rdf_has/[3,4]`.

rdfs_subproperty_of(*?SubProperty, ?Property*)
: True if *SubProperty* is equal to *Property* or *Property* can be reached from *SubProperty* following the `rdfs:subPropertyOf` relation. It can be used to test as well as generate sub-properties or super-properties. Note that the commonly used semantics of this predicate is wired into `rdf_has/[3,4]`.[7] [8]

rdfs_subclass_of(*?SubClass, ?Class*)
: True if *SubClass* is equal to *Class* or *Class* can be reached from *SubClass* following the `rdfs:subClassOf` relation. It can be used to test as well as generate sub-classes or super-classes.[9].

rdfs_class_property(+*Class, ?Property*)
: True if the domain of *Property* includes *Class*. Used to generate all properties that apply to a class.

rdfs_individual_of(*?Resource, ?Class*)
: True if *Resource* is an indivisual of *Class*. This implies *Resource* has an `rdf:type` property that refers to *Class* or a sub-class thereof. Can be used to test, generate classes *Resource* belongs to or generate individuals described by *Class*.

4.2 Collections and Containers

The RDF construct `rdf:parseType=Collection` constructs a list using the `rdf:first` and `rdf:next` relations.

rdfs_member(*?Resource, +Set*)
: Test or generate the members of *Set*. *Set* is either an individual of `rdf:List` or `rdf:Container`.

rdfs_list_to_prolog_list(+*Set, -List*)
: Convert *Set*, which must be an individual of `rdf:List` into a Prolog list of objects.

rdfs_assert_list(+*List, -Resource*)
: Equivalent to `rdfs_assert_list/3` using *DB* = `user`.

rdfs_assert_list(+*List, -Resource, +DB*)
: If *List* is a list of resources, create an RDF list *Resource* that reflects these resources. *Resource* and the sublist resources are generated with `rdf_bnode/1`. The new triples are associated with the database *DB*.

provides *entailment modules* that take the approach outlined above.

[7] BUG: The current implementation cannot deal with cycles

[8] BUG: The current implementation cannot deal with predicates that are an `rdfs:subPropertyOf` of `rdfs:subPropertyOf`, such as `owl:samePropertyAs`.

[9] BUG: The current implementation cannot deal with cycles

4.3 Labels and textual search

Textual search is partly handled by the predicates from the `rdf_db` module and its underlying C-library. For example, literal objects are hashed case-insensitive to speed up the commonly used case-insensitive search.

rdfs_label(*?Resource, ?Language, ?Label*)
: Extract the label from *Resource* or generate all resources with the given *Label*. The label is either associated using a sub-property of `rdfs:label` or it is extracted from the URL using `rdf_split_url/3`. *Language* is unified to the value of the `xml:lang` attribute of the label or a variable if the label has no language specified.

rdfs_label(*?Resource, ?Label*)
: Defined as `rdfs_label`(*Resource, _, Label*).

rdfs_ns_label(*?Resource, ?Language, ?Label*)
: Similar to `rdfs_label/2`, but prefixes the result using the declared namespace alias (see section 3.5) to facilitate user-friendly labels in applications using multiple namespaces that may lead to confusion.

rdfs_ns_label(*?Resource, ?Label*)
: Defined as `rdfs_ns_label`(*Resource, _, Label*).

rdfs_find(*+String, +Description, ?Properties, +Method, -Subject*)
: Find (on backtracking) *Subject*s that satisfy a search specification for textual attributes. *String* is the string searched for. *Description* is an OWL description (see section 5) specifying candidate resources. *Properties* is a list of properties to search for literal objects, *Method* defines the textual matching algorithm. All textual mapping is performed case-insensitive. The matching-methods are described with `rdf_match_label/3`. If *Properties* is unbound, the search is performed in any property and *Properties* is unified with a list holding the property on which the match was found.

5 Module owl

The current SemWeb library distributed with SWI-Prolog does not yet contain an OWL module. A module `owl.pl` is part of the Triple20 triple browser and editor provides limited support for OWL reasoning.

6 Module rdf_edit

> *It is anticipated that this library will eventually be superseeded by facilities running on top of the native* `rdf_transaction/2` *and* `rdf_monitor/2` *facilities. See section 3.6.*

The module `rdf_edit.pl` is a layer than encasulates the modification predicates from section 3.3 for use from a (graphical) editor of the triple store. It adds the following features:

- *Transaction management*
  Modifications are grouped into *transactions* to safeguard the system from failing operations as well as provide meaningfull chunks for undo and journalling.
- *Undo*
  Undo and redo-transactions using a single mechanism to support user-friendly editing.
- *Journalling*
  Record all actions to support analysis, versioning, crash-recovery and an alternative to saving.

6.1 Transaction management

Transactions group low-level modification actions together.

rdfe_transaction(*:Goal*)
Run *Goal*, recording all modifications to the triple store made through section 6.3. Execution is performed as in `once/1`. If *Goal* succeeds the changes are committed. If *Goal* fails or throws an exception the changes are reverted.

Transactions may be nested. A failing nested transaction only reverts the actions performed inside the nested transaction. If the outer transaction succeeds it is committed normally. Contrary, if the outer transaction fails, comitted nested transactions are reverted as well. If any of the modifications inside the transaction modifies a protected file (see `rdfe_set_file_property/2`) the transaction is reverted and `rdfe_transaction/1` throws a permission error.

A successful outer transaction ('level-0') may be undone using `rdfe_undo/0`.

rdfe_transaction(*:Goal, +Name*)
As `rdfe_transaction/1`, naming the transaction *Name*. Transaction naming is intended for the GUI to give the user an idea of the next undo action. See also `rdfe_set_transaction_name/1` and `rdfe_transaction_name/2`.

rdfe_set_transaction_name(*+Name*)
Set the 'name' of the current transaction to *Name*.

rdfe_transaction_name(*?TID, ?Name*)
Query assigned transaction names.

rdfe_transaction_member(*+TID, -Action*)
Enumerate the actions that took place inside a transaction. This can be used by a GUI to optimise the MVC (Model-View-Controller) feedback loop. *Action* is one of:

assert(*Subject, Predicate, Object*)

retract(*Subject, Predicate, Object*)

update(*Subject, Predicate, Object, Action*)

file(*load(Path)*)

file(*unload(Path)*)

6.2 File management

rdfe_is_modified(*?File*)
Enumerate/test whether *File* is modified sinds it was loaded or sinds the last call to `rdfe_clear_modified/1`. Whether or not a file is modified is determined by the MD5 checksum of all triples belonging to the file.

rdfe_clear_modified(*+File*)
Set the *unmodified-MD5* to the current MD5 checksum. See also `rdfe_is_modified/1`.

rdfe_set_file_property(*+File, +Property*)
Control access right and default destination of new triples. *Property* is one of

access(*+Access*)
Where access is one of `ro` or `rw`. Access `ro` is default when a file is loaded for which the user has no write access. If a transaction (see `rdfe_transaction/1`) modifies a file with access `ro` the transaction is reversed.

default(*+Default*)
Set this file to be the default destination of triples. If *Default* is `fallback` it is only the default for triples that have no clear default destination. If it is `all` all new triples are added to this file.

rdfe_get_file_property(*?File, ?Property*)
Query properties set with `rdfe_set_file_property/2`.

6.3 Encapsulated predicates

The following predicates encapsulate predicates from the `rdf_db` module that modify the triple store. These predicates can only be called when inside a *transaction*. See `rdfe_transaction/1`.

rdfe_assert(*+Subject, +Predicate, +Object*)
Encapsulates `rdf_assert/3`.

rdfe_retractall(*?Subject, ?Predicate, ?Object*)
Encapsulates `rdf_retractall/3`.

rdfe_update(*+Subject, +Predicate, +Object, +Action*)
Encapsulates `rdf_update/4`.

rdfe_load(*+In*)
Encapsulates `rdf_load/1`.

rdfe_unload(*+In*)
Encapsulates `rdf_unload/1`.

6.4 High-level modification predicates

This section describes a (yet very incomplete) set of more high-level operations one would like to be able to perform. Eventually this set may include operations based on RDFS and OWL.

rdfe_delete(+*Resource*)
Delete all traces of *resource*. This implies all triples where *Resource* appears as *subject*, *predicate* or *object*. This predicate starts a transation.

6.5 Undo

Undo aims at user-level undo operations from a (graphical) editor.

rdfe_undo
Revert the last outermost ('level 0') transaction (see `rdfe_transaction/1`). Successive calls go further back in history. Fails if there is no more undo information.

rdfe_redo
Revert the last `rdfe_undo/0`. Successive calls revert more `rdfe_undo/0` operations. Fails if there is no more redo information.

rdfe_can_undo(-*TID*)
Test if there is another transaction that can be reverted. Used for activating menus in a graphical environment. *TID* is unified to the transaction id of the action that will be reverted.

rdfe_can_redo(-*TID*)
Test if there is another undo that can be reverted. Used for activating menus in a graphical environment. *TID* is unified to the transaction id of the action that will be reverted.

6.6 Journalling

Optionally, every action through this module is immediately send to a *journal-file*. The journal provides a full log of all actions with a time-stamp that may be used for inspection of behaviour, version management, crash-recovery or an alternative to regular save operations.

rdfe_open_journal(+*File, +Mode*)
Open a existing or new journal. If *Mode* equala `append` and *File* exists, the journal is first replayed. See `rdfe_replay_journal/1`. If *Mode* is `write` the journal is truncated if it exists.

rdfe_close_journal
Close the currently open journal.

rdfe_current_journal(-*Path*)
Test whether there is a journal and to which file the actions are journalled.

rdfe_replay_journal(+*File*)
Read a journal, replaying all actions in it. To do so, the system reads the journal a transaction at a time. If the transaction is closed with a *commit* it executes the actions inside the journal. If it is closed with a *rollback* or not closed at all due to a crash the actions inside the journal are discarded. Using this predicate only makes sense to inspect the state at the end of a journal without modifying the journal. Normally a journal is replayed using the `append` mode of `rdfe_open_journal/2`.

6.7 Broadcasting change events

To realise a modular graphical interface for editing the triple store, the system must use some sort of *event* mechanism. This is implemented by the XPCE library `broadcast` which is described in the XPCE User Guide. In this section we describe the terms brodcasted by the library.

rdf_transaction(+*Id*)

A 'level-0' transaction has been committed. The system passes the identifier of the transaction in *Id*. In the current implementation there is no way to find out what happened inside the transaction. This is likely to change in time.

If a transaction is reverted due to failure or exception *no* event is broadcasted. The initiating GUI element is supposed to handle this possibility itself and other components are not affected as the triple store is not changed.

rdf_undo(+*Type*, +*Id*)

This event is broadcasted after an `rdfe_undo/0` or `rdfe_redo/0`. *Type* is one of `undo` or `redo` and *Id* identifies the transaction as above.

7 Related packages and issues

The SWI-Prolog SemWeb package is designed to provide access to the Semantic Web languages from Prolog. It consists of the low level `rdf_db.pl` store with layers such as `rdfs.pl` to provide more high level querying of a triple set with relations such as `rdfs_individual_of/2`, `rdfs_subclass_of/2`, etc. SeRQL is a semantic web query language taking another route. Instead of providing alternative relations SeRQL defines a graph query on de *deductive closure* of the triple set. For example, under assumption of RDFS entailment rules this makes the query `rdf`(*S, rdf:type, Class*) equivalent to `rdfs_individual_of`(*S, Class*).

We developed a parser for SeRQL which compiles SeRQL path expressions into Prolog conjunctions of `rdf`(*Subject, Predicate, Object*) calls. *Entailment modules* realise a fully logical implementation of `rdf/3` including the entailment reasoning required to deal with a Semantic Web language or application specific reasoning. The infra structure is completed with a query optimiser and an HTTP server compliant to the Sesame implementation of the SeRQL language. The Sesame Java client can be used to access Prolog servers from Java, while the Prolog client can be used to access the Sesame SeRQL server. For further details, see the project home.

Acknowledgements

This research was supported by the following projects: MIA and MultimediaN project (www.multimedian.nl) funded through the BSIK programme of the Dutch Government and the FP-6 project HOPS of the European Commision.

The implementation of AVL trees is based on libavl by Brad Appleton. See the source file `avl.c` for details.