SWI-Prolog Semantic Web Library 3.0

Jan Wielemaker
University of Amsterdam/VU University Amsterdam
The Netherlands
E-mail: J.Wielemaker@vu.nl

April 18, 2017

Abstract

This document describes the SWI-Prolog *semweb* package. The core of this package is an efficient main-memory based RDF store that is tightly connected to Prolog. Additional libraries provide reading and writing RDF/XML and Turtle data, caching loaded RDF documents and persistent storage. This package is the core of a ready-to-run platform for developing Semantic Web applications named [ClioPatria]http://cliopatria.swi-prolog.org, which is distributed seperately. The SWI-Prolog RDF store is amoung the most memory efficient main-memory stores for RDF[1]

Version 3 of the RDF library enhances concurrent use of the library by allowing for lock-free reading and writing using short-held locks. It provides Prolog compatible *logical update view* on the triple store and isolation using *transactions* and jargonsnapshots. This version of the library provides near real-time modification and querying of RDF graphs, making it particularly interesting for handling streaming RDF and graph manipulation tasks.

[1] http://cliopatria.swi-prolog.org/help/source/doc/home/vnc/prolog/src/ClioPatria/web/help/memusage.txt

Contents

1 Introduction

The core of the SWI-Prolog package `semweb` is an efficient main-memory RDF store written in C that is tightly integrated with Prolog. It provides a fully logical predicate `rdf/3` to query the RDF store efficiently by using multiple (currently 9) indexes. In addition, SWI-Prolog provides libraries for reading and writing XML/RDF and Turtle and a library that provides persistency using a combination of efficient binary snapshots and journals.

Below, we describe a few usage scenarios that guides the current design of this Prolog-based RDF store.

Application prototyping platform Bundled with [ClioPatria]http://cliopatria.swi-prolog.org, the store is an efficient platform for prototyping a wide range of semantic web applications. Prolog, connected to the main-memory based store is a productive platform for writing application logic that can be made available through the SPARQL endpoint of ClioPatria, using an application specific API (typically based on JSON or XML) or as an HTML based end-user application. Prolog is more versatile than SPARQL, allows composing of the logic from small building blocks and does not suffer from the *Object-relational impedance mismatch*.

Data integration The SWI-Prolog store is optimized for entailment on the `rdfs:subPropertyOf` relation. The `rdfs:subPropertyOf` relation is crucial for integrating data from multiple sources while preserving the original richness of the sources because integration can be achieved by defining the native properties as sub-properties of properties from a unifying schema such as Dublin Core.

Dynamic data This RDF store is one of the few stores that is primarily based on *backward reasoning*. The big advantage of backward reasoning is that it can much easier deal with changes to the database because it does not have to take care of propagating the consequences. Backward reasoning reduces storage requirements. The price is more reasoning during querying. In many scenarios the extra reasoning using a main memory will outperform the fetching the precomputed results from external storage.

Prototyping reasoning systems Reasoning systems, not necessarily limited to entailment reasoning, can be prototyped efficiently on the Prolog based store. This includes 'what-if' reasoning, which is supported by *snapshot* and *transaction* isolation. These features, together with the concurrent loading capabilities, make the platform well equiped to collect relevant data from large external stores for intensive reasoning. Finally, the [TIPC]http://www.swi-prolog.org/pldoc/package/tipc.html package can be used to create networks of cooperating RDF based agents.

Streaming RDF Transactions, snapshots, concurrent modifications and the database monitoring facilities (see `rdf_monitor/2`) make the platform well suited for prototyping systems that deal with streaming RDF data.

2 Scalability

Depending on the OS and further application restrictions, the SWI-Prolog RDF stores scales to about 15 million triples on 32-bit hardware. On 64-bit hardware, the scalability is limited by the amount of

physical memory, allowing for approximately 4 million triples per gigabyte. The other limiting factor for practical use is the time required to load data and/or restore the database from the persistent file backup. Performance depends highly on hardware, concurrent performance and whether or not the data is spread over multiple (named) graphs that can be loaded in parallel. Restoring over 20 million triples per minute is feasible on medium hardware (Intel i7/2600 running Ubuntu 12.10).

3 Two RDF APIs

The current 'semweb' package provides two sets of interface predicates. The original set is described in section 3.2. The new API is described in section 3.1. The original API was designed when RDF was not yet standardised and did not yet support data types and language indicators. The new API is designed from the RDF 1.1 specification, introducing consistent naming and access to literals using the *value space*. The new API is currently defined on top of the old API, so both APIs can be mixed in a single application.

3.1 library(semweb/rdf11): The RDF database

The `library(semweb/rdf11)` provides a new interface to the SWI-Prolog RDF database based on the RDF 1.1 specification.

3.1.1 Query the RDF database

rdf(*?S, ?P, ?O*) *[nondet]*
rdf(*?S, ?P, ?O, ?G*) *[nondet]*

True if an RDF triple <*S*,*P*,*O*> exists, optionally in the graph *G*. The object *O* is either a resource (atom) or one of the terms listed below. The described types apply for the case where *O* is unbound. If *O* is instantiated it is converted according to the rules described with `rdf_assert/3`.

Triples consist of the following three terms:

- Blank nodes are encoded by atoms that start with '_:'.
- IRIs appear in two notations:
  - Full IRIs are encoded by atoms that do not start with '_:'. Specifically, an IRI term is not required to follow the IRI standard grammar.
  - Abbreviated IRI notation that allows IRI prefix aliases that are registered by rdf_register_prefix/[2,3] to be used. Their notation is `Alias:Local`, where Alias and Local are atoms. Each abbreviated IRI is expanded by the system to a full IRI.
- Literals appear in two notations:
  - String@Lang A language-tagged string, where String is a Prolog string and Lang is an atom.
  - Value^^Type A type qualified literal. For unknown types, Value is a Prolog string. If type is known, the Prolog representations from the table below are used.

Datatype IRI	Prolog term
xsd:float	float
xsd:double	float
xsd:decimal	float (1)
xsd:integer	integer
XSD integer sub-types	integer
xsd:boolean	`true` or `false`
xsd:date	`date(Y,M,D)`
xsd:dateTime	`date_time(Y,M,D,HH,MM,SS)` (2,3)
xsd:gDay	integer
xsd:gMonth	integer
xsd:gMonthDay	`month_day(M,D)`
xsd:gYear	integer
xsd:gYearMonth	`year_month(Y,M)`
xsd:time	`time(HH,MM,SS)` (2)

Notes:

(1) The current implementation of `xsd:decimal` values as floats is formally incorrect. Future versions of SWI-Prolog may introduce decimal as a subtype of rational.

(2) *SS* fields denote the number of seconds. This can either be an integer or a float.

(3) The `date_time` structure can have a 7th field that denotes the timezone offset **in seconds** as an integer.

In addition, a *ground* object value is translated into a properly typed RDF literal using `rdf_canonical_literal/2`.

There is a fine distinction in how duplicate statements are handled in rdf/[3,4]: backtracking over `rdf/3` will never return duplicate triples that appear in multiple graphs. `rdf/4` will return such duplicate triples, because their graph term differs.

Arguments

S	is the subject term. It is either a blank node or IRI.
P	is the predicate term. It is always an IRI.
O	is the object term. It is either a literal, a blank node or IRI (except for `true` and `false` that denote the values of datatype XSD boolean).
G	is the graph term. It is always an IRI.

See also

- Triple pattern querying
- `xsd_number_string/2` and `xsd_time_string/3` are used to convert between lexical representations and Prolog terms.

rdf_has(*?S, ?P, ?O*) *[nondet]*

rdf_has(*?S, ?P, ?O, -RealP*) *[nondet]*

Similar to `rdf/3` and `rdf/4`, but *P* matches all predicates that are defined as an rdfs:subPropertyOf of *P*. This predicate also recognises the predicate properties `inverse_of` and `symmetric`. See `rdf_set_predicate/2`.

rdf_reachable(*?S, +P, ?O*) *[nondet]*
rdf_reachable(*?S, +P, ?O, +MaxD, -D*) *[nondet]*

True when *O* can be reached from *S* using the transitive closure of *P*. The predicate uses (the internals of) `rdf_has/3` and thus matches both rdfs:subPropertyOf and the `inverse_of` and `symmetric` predicate properties. The version `rdf_reachable/5` maximizes the steps considered and returns the number of steps taken.

If both *S* and *O* are given, these predicates are `semidet`. The number of steps *D* is minimal because the implementation uses *breath first* search.

Constraints on literal values

{}(*+Where*) *[semidet]*
rdf_where(*+Where*) *[semidet]*

Formulate constraints on RDF terms, notably literals. These are intended to be used as illustrated below. RDF constraints are pure: they may be placed before, after or inside a graph pattern and, provided the code contains no *commit* operations (!, `->`), the semantics of the goal remains the same. Preferably, constraints are placed *before* the graph pattern as they often help the RDF database to exploit its literal indexes. In the example below, the database can choose between using the subject and/or predicate hash or the ordered literal table.

```
{ Date >= "2000-01-01"^^xsd:dateTime },
rdf(S, P, Date)
```

The following constraints are currently defined:

`> , >= , == , =< , <`
: The comparison operators are defined between numbers (of any recognised type), typed literals of the same type and langStrings of the same language.

prefix(*String, Pattern*)

substring(*String, Pattern*)

word(*String, Pattern*)

like(*String, Pattern*)

icase(*String, Pattern*)
: Text matching operators that act on both typed literals and langStrings.

lang_matches(*Term, Pattern*)
: Demands a full RDF term (Text@Lang) or a plain *Lang* term to match the language pattern *Pattern*.

The predicates `rdf_where/1` and {}/1 are identical. The `rdf_where/1` variant is provided to avoid ambiguity in applications where {}/1 is used for other purposes. Note that it is also possible to write `rdf11:{...}`.

3.1.2 Enumerating and testing objects

Enumerating objects by role

rdf_subject(*?S*) *[nondet]*

True when *S* is a currently known *subject*, i.e. it appears in the subject position of some visible triple. The predicate is *semidet* if *S* is ground.

rdf_predicate(*?P*) *[nondet]*

True when *P* is a currently known predicate, i.e. it appears in the predicate position of some visible triple. The predicate is *semidet* if *P* is ground.

rdf_object(*?O*) *[nondet]*

True when *O* is a currently known object, i.e. it appeasr in the object position of some visible triple. If Term is ground, it is pre-processed as the object argument of `rdf_assert/3` and the predicate is *semidet*.

rdf_node(*?T*) *[nondet]*

True when *T* appears in the subject or object position of a known triple, i.e., is a node in the RDF graph.

rdf_graph(*?Graph*) *[nondet]*

True when *Graph* is an existing graph.

Enumerating objects by type

rdf_literal(*?Term*) *[nondet]*

True if *Term* is a known literal. If *Term* is ground, it is pre-processed as the object argument of `rdf_assert/3` and the predicate is *semidet*.

rdf_bnode(*?BNode*) *[nondet]*

True if *BNode* is a currently known blank node. The predicate is *semidet* if *BNode* is ground.

rdf_iri(*?IRI*) *[nondet]*

True if *IRI* is a current *IRI*. The predicate is *semidet* if *IRI* is ground.

rdf_name(*?Name*) *[nondet]*

True if *Name* is a current IRI or literal. The predicate is *semidet* if *Name* is ground.

rdf_term(*?Term*) *[nondet]*

True if *Term* appears in the RDF database. *Term* is either an iri, literal or blank node and may appear in any position of any triple. If *Term* is ground, it is pre-processed as the object argument of `rdf_assert/3` and the predicate is *semidet*.

Testing objects types

rdf_is_iri(@*IRI*) *[semidet]*

True if *IRI* is an RDF *IRI* term.

For performance reasons, this does not check for compliance to the syntax defined in RFC 3987. This checks whether the term is (1) an atom and (2) not a blank node identifier.

Success of this goal does not imply that the *IRI* is present in the database (see `rdf_iri/1` for that).

rdf_is_bnode(@*Term*) *[semidet]*

True if *Term* is an RDF blank node identifier.

A blank node is represented by an atom that starts with `_:`.

Success of this goal does not imply that the blank node is present in the database (see `rdf_bnode/1` for that).

For backwards compatibility, atoms that are represented with an atom that starts with `___` are also considered to be a blank node.

rdf_is_literal(@*Term*) *[semidet]*

True if *Term* is an RDF literal term.

An RDF literal term is of the form 'String@LanguageTag `or` Value^^Datatype'.

Success of this goal does not imply that the literal is well-formed or that it is present in the database (see `rdf_literal/1` for that).

rdf_is_name(@*Term*) *[semidet]*

True if *Term* is an RDF Name, i.e., an IRI or literal.

Success of this goal does not imply that the name is well-formed or that it is present in the database (see `rdf_name/1`) for that).

rdf_is_object(@*Term*) *[semidet]*

True if *Term* can appear in the object position of a triple.

Success of this goal does not imply that the object term in well-formed or that it is present in the database (see `rdf_object/1`) for that).

Since any RDF term can appear in the object position, this is equaivalent to `rdf_is_term/1`.

rdf_is_predicate(@*Term*) *[semidet]*

True if *Term* can appear in the predicate position of a triple.

Success of this goal does not imply that the predicate term is present in the database (see `rdf_predicate/1`) for that).

Since only IRIs can appear in the predicate position, this is equivalent to `rdf_is_iri/1`.

rdf_is_subject(@*Term*) *[semidet]*

True if *Term* can appear in the subject position of a triple.

Only blank nodes and IRIs can appear in the subject position.

Success of this goal does not imply that the subject term is present in the database (see `rdf_subject/1`) for that).

Since blank nodes are represented by atoms that start with '_:' and an IRIs are atoms as well, this is equivalent to `atom(Term)`.

rdf_is_term(@*Term*) *[semidet]*

True if *Term* can be used as an RDF term, i.e., if *Term* is either an IRI, a blank node or an RDF literal.

Success of this goal does not imply that the RDF term is present in the database (see `rdf_term/1`) for that).

3.1.3 RDF literals

rdf_canonical_literal(++*In, -Literal*) *[det]*

Transform a relaxed literal specification as allowed for `rdf_assert/3` into its canonical form. The following Prolog terms are translated:

Prolog Term	**Datatype IRI**
float	xsd:double
integer	xsd:integer
string	xsd:string
`true` or `false`	xsd:boolean
`date(Y,M,D)`	xsd:date
`date_time(Y,M,D,HH,MM,SS)`	xsd:dateTime
`date_time(Y,M,D,HH,MM,SS,TZ)`	xsd:dateTime
`month_day(M,D)`	xsd:gMonthDay
`year_month(Y,M)`	xsd:gYearMonth
`time(HH,MM,SS)`	xsd:time

For example:

```
?- rdf_canonical_literal(42, X).
X = 42^^'http://www.w3.org/2001/XMLSchema#integer'.
```

rdf_lexical_form(++*Literal, -Lexical:compound*) *[det]*

True when *Lexical* is the lexical form for the literal *Literal*. *Lexical* is of one of the forms below. The ntriples serialization is obtained by transforming String into a proper ntriples string using double quotes and escaping where needed and turning Type into a proper IRI reference.

- String^^Type
- String@Lang

rdf_compare(*-Diff, +Left, +Right*) *[det]*

True if the RDF terms *Left* and *Right* are ordered according to the comparison operator *Diff*. The ordering is defines as:

- Literal < BNode < IRI
- For literals
  - Numeric < non-numeric
  - Numeric literals are ordered by value. If both are equal, floats are ordered before integers.
  - Other data types are ordered lexicographically.
- BNodes and IRIs are ordered lexicographically.

Note that this ordering is a complete ordering of RDF terms that is consistent with the partial ordering defined by SPARQL.

Arguments

Diff	is one of <, = or >

3.1.4 Accessing RDF graphs

rdf_default_graph(-*Graph*) *[det]*
rdf_default_graph(-*Old, +New*) *[det]*
Query/set the notion of the default graph. The notion of the default graph is local to a thread. Threads created inherit the default graph from their creator. See `set_prolog_flag/2`.

3.1.5 Modifying the RDF store

rdf_assert(+*S, +P, +O*) *[det]*
rdf_assert(+*S, +P, +O, +G*) *[det]*
Assert a new triple. If *O* is a literal, certain Prolog terms are translated to typed RDF literals. These conversions are described with `rdf_canonical_literal/2`.

If a type is provided using Value^^Type syntax, additional conversions are performed. All types accept either an atom or Prolog string holding a valid RDF lexical value for the type and xsd:float and xsd:double accept a Prolog integer.

rdf_retractall(*?S, ?P, ?O*) *[nondet]*
rdf_retractall(*?S, ?P, ?O, ?G*) *[nondet]*
Remove all matching triples from the database. Matching is performed using the same rules as `rdf/3`. The call does not instantiate any of its arguments.

rdf_create_bnode(–*BNode*)
Create a new *BNode*. A blank node is an atom starting with `_:`. Blank nodes generated by this predicate are of the form `_:genid` followed by a unique integer.

3.1.6 Accessing RDF collections

The following predicates are utilities to access RDF 1.1 *collections*. A collection is a linked list created from `rdf:first` and `rdf:next` triples, ending in `rdf:nil`.

rdf_last(+*RDFList, -Last*) *[det]*
True when *Last* is the last element of *RDFList*. Note that if the last cell has multiple rdf:first triples, this predicate becomes nondet.

rdf_list(*?RDFTerm*) *[semidet]*
True if *RDFTerm* is a proper RDF list. This implies that every node in the list has an `rdf:first` and `rdf:rest` property and the list ends in `rdf:nil`.

If *RDFTerm* is unbound, *RDFTerm* is bound to each *maximal* RDF list. An RDF list is *maximal* if there is no triple `rdf(_, rdf:rest, RDFList)`.

rdf_list(+*RDFList, -PrologList*) *[det]*
True when *PrologList* represents the rdf:first objects for all cells in *RDFList*. Note that this can be non-deterministic if cells have multiple rdf:first or rdf:rest triples.

rdf_length(+*RDFList, -Length:nonneg*) *[nondet]*
True when *Length* is the number of cells in *RDFList*. Note that a list cell may have multiple rdf:rest triples, which makes this predicate non-deterministic. This predicate does not check whether the list cells have associated values (rdf:first). The list must end in rdf:nil.

rdf_member(*?Member, +RDFList*) *[nondet]*

True when *Member* is a member of *RDFList*

rdf_nth0(*?Index, +RDFList, ?X*) *[nondet]*

True when *X* is the *Index*-th element (0-based) of *RDFList*. This predicate is deterministic if *Index* is given and the list has no multiple rdf:first or rdf:rest values.

rdf_assert_list(*+PrologList, ?RDFList*) *[det]*
rdf_assert_list(*+PrologList, ?RDFList, +Graph*) *[det]*

Create an RDF list from the given Prolog List. *PrologList* must be a proper Prolog list and all members of the list must be acceptable as object for `rdf_assert/3`. If *RDFList* is unbound and *PrologList* is not empty, `rdf_create_bnode/1` is used to create *RDFList*.

rdf_retract_list(*+RDFList*) *[det]*

Retract the rdf:first, rdf:rest and rdf:type=rdf:'List' triples from all nodes reachable through rdf:rest. Note that other triples that exist on the nodes are left untouched.

3.1.7 library(semweb/rdf11_containers): RDF 1.1 Containers

author
- Wouter Beek
- Jan Wielemaker

version 2016/01

See also http://www.w3.org/TR/2014/REC-rdf-schema-20140225/#ch_containervocab

Compatibility RDF 1.1

Implementation of the conventional human interpretation of RDF 1.1 containers.

RDF containers are open enumeration structures as opposed to RDF collections or RDF lists which are closed enumeration structures. The same resource may appear in a container more than once. A container may be contained in itself.

---

rdf_alt(*+Alt, ?Default, ?Others*) *[nondet]*

True when *Alt* is an instance of `rdf:Alt` with first member *Default* and remaining members *Others*.

Notice that this construct adds no machine-processable semantics but is conventionally used to indicate to a human reader that the numerical ordering of the container membership properties of Container is intended to only be relevant in distinguishing between the first and all non-first members.

Default denotes the default option to take when choosing one of the alternatives container in Container. *Others* denotes the non-default options that can be chosen from.

rdf_assert_alt(*?Alt, +Default, +Others:list*) *[det]*
rdf_assert_alt(*?Alt, +Default, +Others:list, +Graph*) *[det]*

Create an rdf:*Alt* with the given *Default* and *Others*. *Default* and the members of *Others* must be valid object terms for `rdf_assert/3`.

rdf_bag(+*Bag, -List:list*) *[nondet]*

True when *Bag* is an rdf:*Bag* and set is the set values related through container membership properties to *Bag*.

Notice that this construct adds no machine-processable semantics but is conventionally used to indicate to a human reader that the numerical ordering of the container membership properties of Container is intended to not be significant.

rdf_assert_bag(*?Bag, +Set:list*) *[det]*
rdf_assert_bag(*?Bag, +Set:list, +Graph*) *[det]*

Create an rdf:*Bag* from the given set of values. The members of *Set* must be valid object terms for `rdf_assert/3`.

rdf_seq(+*Seq, -List:list*) *[nondet]*

True when *Seq* is an instance of rdf:*Seq* and *List* is a list of associated values, ordered according to the container membership property used.

Notice that this construct adds no machine-processable semantics but is conventionally used to indicate to a human reader that the numerical ordering of the container membership properties of Container is intended to be significant.

rdf_assert_seq(*?Seq, +List*) *[det]*
rdf_assert_seq(*?Seq, +List, +Graph*) *[det]*

rdfs_container(+*Container, -List*) *[nondet]*

True when *List* is the list of objects attached to *Container* using a container membership property (rdf:_0, rdf:_1, ...). If multiple objects are connected to the *Container* using the same membership property, this predicate selects one value non-deterministically.

rdfs_container_membership_property(*?Property*) *[nondet]*

True when *Property* is a container membership property (rdf:_1, rdf:_2, ...).

rdfs_container_membership_property(*?Property, ?Number:nonneg*) *[nondet]*

True when *Property* is the Nth container membership property.

Success of this goal does not imply that *Property* is present in the database.

rdfs_member(*?Elem, ?Container*) *[nondet]*

True if `rdf(Container, P, Elem)` is true and P is a container membership property.

rdfs_nth0(*?N, ?Container, ?Elem*) *[nondet]*

True if `rdf(Container, P, Elem)` is true and P is the *N*-th (0-based) container membership property.

3.2 library(semweb/rdf_db): The RDF database

The central module of the RDF infrastructure is `library(semweb/rdf_db)`. It provides storage and indexed querying of RDF triples. RDF data is stored as quintuples. The first three elements denote the RDF triple. The extra *Graph* and *Line* elements provide information about the origin of the triple.

The actual storage is provided by the *foreign language (C)* module. Using a dedicated C-based implementation we can reduced memory usage and improve indexing capabilities, for example by

providing a dedicated index to support entailment over `rdfs:subPropertyOf`. Currently the following indexes are provided (S=subject, P=predicate, O=object, G=graph):

- S, P, O, SP, PO, SPO, G, SG, PG
- Predicates connect by **rdfs:subPropertyOf** are combined in a *predicate cloud*. The system causes multiple predicates in the cloud to share the same hash. The cloud maintains a 2-dimensional array that expresses the closure of all rdfs:subPropertyOf relations. This index supports `rdf_has/3` to query a property and all its children efficiently.
- Additional indexes for predicates, resources and graphs allow enumerating these objects without duplicates. For example, using `rdf_resource/1` we enumerate all resources in the database only once, while enumeration using e.g., `(rdf(R,_,_);rdf(_,_,R))` normally produces many duplicate answers.
- Literal *Objects* are combined in a *skip list* after case normalization. This provides for efficient case-insensitive search, prefix and range search. The plugin library `library(semweb/litindex)` provides indexed search on tokens inside literals.

3.2.1 Query the RDF database

rdf(*?Subject, ?Predicate, ?Object*) *[nondet]*

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 performance penalty. 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 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 literal querying purposes, *Object* can be of the form `literal(+Query, -Value)`, where Query is one of the terms below. If the Query takes a literal argument and the value has a numeric type numerical comparison is performed.

plain(+*Text*)
: Perform exact match and demand the language or type qualifiers to match. This query is fully indexed.

icase(+*Text*)
Perform a full but case-insensitive match. This query is fully indexed.

exact(+*Text*)
Same as `icase(Text)`. Backward compatibility.

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 indexed using the skip list of literals.

ge(+*Literal*)
Match any literal that is equal or larger then *Literal* in the ordered set of literals.

gt(+*Literal*)
Match any literal that is larger then *Literal* in the ordered set of literals.

eq(+*Literal*)
Match any literal that is equal to *Literal* in the ordered set of literals.

le(+*Literal*)
Match any literal that is equal or smaller then *Literal* in the ordered set of literals.

lt(+*Literal*)
Match any literal that is smaller then *Literal* in the ordered set of literals.

between(+*Literal1*, +*Literal2*)
Match any literal that is between *Literal1* and *Literal2* in the ordered set of literals. This may include both *Literal1* and *Literal2*.

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`. The predicate `rdf/3` raises a type-error if called with improper arguments. If `rdf/3` is called with a term `literal(_)` as *Subject* or *Predicate* object it fails silently. This allows for graph matching goals like `rdf(S,P,O),rdf(O,P2,O2)` to proceed without errors.

rdf(*?Subject, ?Predicate, ?Object, ?Source*) *[nondet]*
As `rdf/3` but in addition query the graph to which the triple belongs. Unlike `rdf/3`, this predicate does not remove duplicates from the result set.

Arguments

Source	is a term Graph:Line. If *Source* is instatiated, passing an atom is the same as passing Atom:_.

rdf_has(*?Subject, +Predicate, ?Object*) *[nondet]*
Succeeds if the triple `rdf(Subject, Predicate, Object)` is true exploiting the rdfs:subPropertyOf predicate as well as inverse predicates declared using `rdf_set_predicate/2` with the `inverse_of` property.

rdf_has(*?Subject, +Predicate, ?Object, -RealPredicate*) *[nondet]*

Same as `rdf_has/3`, but *RealPredicate* is unified to the actual predicate that makes this relation true. *RealPredicate* must be *Predicate* or an rdfs:subPropertyOf *Predicate*. If an inverse match is found, *RealPredicate* is the term `inverse_of(Pred)`.

rdf_reachable(*?Subject, +Predicate, ?Object*) *[nondet]*

Is true if *Object* can be reached from *Subject* following the transitive predicate *Predicate* or a sub-property thereof, while repecting the `symetric(true)` or `inverse_of(P2)` properties.

If used with either *Subject* or *Object* unbound, it first returns the origin, followed by the reachable nodes in breath-first search-order. The implementation internally looks one solution ahead and succeeds deterministically on the last solution. This predicate never generates the same node twice and is robust against cycles in the transitive relation.

With all arguments instantiated, it succeeds deterministically 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:

```
?- 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_reachable(*?Subject, +Predicate, ?Object, +MaxD, -D*) *[nondet]*

Same as `rdf_reachable/3`, but in addition, *MaxD* limits the number of edges expanded and *D* is unified with the 'distance' between *Subject* and *Object*. Distance 0 means *Subject* and *Object* are the same resource. *MaxD* can be the constant `infinite` to impose no distance-limit.

3.2.2 Enumerating objects

The predicates below enumerate the basic objects of the RDF store. Most of these predicates also enumerate objects that are not associated to any currently visible triple. Objects are retained as long as they are visible in active queries or *snapshots*. After that, some are reclaimed by the RDF garbage collector, while others are never reclaimed.

rdf_subject(*?Resource*) *[nondet]*

True if *Resource* appears as a subject. This query respects the visibility rules implied by the logical update view.

See also `rdf_resource/1`.

rdf_resource(*?Resource*) *[nondet]*

True when *Resource* is a resource used as a subject or object in a triple.

This predicate is primarily intended as a way to process all resources without processing resources twice. The user must be aware that some of the returned resources may not appear in any *visible* triple.

rdf_current_predicate(*?Predicate*) *[nondet]*

True when *Predicate* is a currently known predicate. Predicates are created if a triples is created that uses this predicate or a property of the predicate is set using `rdf_set_predicate/2`. The predicate may (no longer) have triples associated with it.

Note that resources that have `rdf:type rdf:Property` are not automatically included in the result-set of this predicate, while *all* resources that appear as the second argument of a triple *are* included.

See also `rdf_predicate_property/2`.

rdf_current_literal(*-Literal*) *[nondet]*

True when *Literal* is a currently known literal. Enumerates each unique literal exactly once. Note that it is possible that the literal only appears in already deleted triples. Deleted triples may be locked due to active queries, transactions or snapshots or may not yet be reclaimed by the garbage collector.

rdf_graph(*?Graph*) *[nondet]*

True when *Graph* is an existing graph.

rdf_current_ns(*:Prefix, ?URI*) *[nondet]* **deprecated**

. Use `rdf_current_prefix/2`.

3.2.3 Modifying the RDF database

The predicates below modify the RDF store directly. In addition, data may be loaded using `rdf_load/2` or by restoring a persistent database using `rdf_attach_db/2`. Modifications follow the Prolog *logical update view* semantics, which implies that modifications remain invisible to already running queries. Further isolation can be achieved using `rdf_transaction/3`.

rdf_assert(*+Subject, +Predicate, +Object*) *[det]*

Assert a new triple into the database. This is equivalent to `rdf_assert/4` using Graph `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. Complete duplicates (including the same graph and 'line' and with a compatible 'lifespan') are not added to the database.

rdf_assert(*+Subject, +Predicate, +Object, +Graph*) *[det]*

As `rdf_assert/3`, adding the predicate to the indicated named graph.

Arguments

Graph	is either the name of a graph (an atom) or a term *Graph*:Line, where Line is an integer that denotes a line number.

rdf_retractall(*?Subject, ?Predicate, ?Object*) *[det]*

Remove all matching triples from the database. As `rdf_retractall/4` using an unbound graph.

rdf_retractall(*?Subject, ?Predicate, ?Object, ?Graph*) *[det]*

As `rdf_retractall/3`, also matching *Graph*. This is particulary useful to remove all triples coming from a loaded file. See also `rdf_unload/1`.

rdf_update(+*Subject, +Predicate, +Object, +Action*) *[det]*

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)`.

graph(*Graph*)
Moves the triple from its current named graph to *Graph*.

rdf_update(+*Subject, +Predicate, +Object, +Graph, +Action*) *[det]*

As `rdf_update/4` but allows for specifying the graph.

3.2.4 Update view, transactions and snapshots

The update semantics of the RDF database follows the conventional Prolog *logical update view*. In addition, the RDF database supports *transactions* and *snapshots*.

rdf_transaction(*:Goal*) *[semidet]*

Same as `rdf_transaction(Goal, user, [])`. See `rdf_transaction/3`.

rdf_transaction(*:Goal, +Id*) *[semidet]*

Same as `rdf_transaction(Goal, Id, [])`. See `rdf_transaction/3`.

rdf_transaction(*:Goal, +Id, +Options*) *[semidet]*

Run *Goal* in an RDF transaction. Compared to the ACID model, RDF transactions have the following properties:

1. Modifications inside the transactions become all atomically visible to the outside world if *Goal* succeeds or remain invisible if *Goal* fails or throws an exception. I.e., the *atomicy* property is fully supported.
2. *Consistency* is not guaranteed. Later versions may implement consistency constraints that will be checked serialized just before the actual commit of a transaction.
3. Concurrently executing transactions do not infuence each other. I.e., the *isolation* property is fully supported.
4. *Durability* can be activated by loading `library(semweb/rdf_persistency)`.

Processed options are:

snapshot(+*Snapshot*)
Execute *Goal* using the state of the RDF store as stored in *Snapshot*. See `rdf_snapshot/1`. *Snapshot* can also be the atom `true`, which implies that an anonymous snapshot is created at the current state of the store. Modifications due to executing *Goal* are only visible to *Goal*.

rdf_snapshot(-*Snapshot*) *[det]*

Take a snapshot of the current state of the RDF store. Later, goals may be executed in the context of the database at this moment using `rdf_transaction/3` with the `snapshot` option. A snapshot created outside a transaction exists until it is deleted. Snapshots taken inside a transaction can only be used inside this transaction.

rdf_delete_snapshot(+*Snapshot*) *[det]*

Delete a snapshot as obtained from `rdf_snapshot/1`. After this call, resources used for maintaining the snapshot become subject to garbage collection.

rdf_active_transaction(*?Id*) *[nondet]*

True if *Id* is the identifier of a transaction in the context of which this call is executed. If *Id* is not instantiated, backtracking yields transaction identifiers starting with the innermost nested transaction. Transaction identifier terms are not copied, need not be ground and can be instantiated during the transaction.

rdf_current_snapshot(*?Term*) *[nondet]*

True when *Term* is a currently known snapshot.

bug Enumeration of snapshots is slow.

3.2.5 Type checking predicates

rdf_is_resource(@*Term*) *[semidet]*

True if *Term* is an RDF resource. Note that this is merely a type-test; it does not mean this resource is involved in any triple. Blank nodes are also considered resources.

See also `rdf_is_bnode/1`

rdf_is_bnode(+*Id*)

Tests if a resource is a blank node (i.e. is an anonymous resource). A blank node is represented as an atom that starts with `_:`. For backward compatibility reason, ___ is also considered to be a blank node.

See also `rdf_bnode/1`.

rdf_is_literal(@*Term*) *[semidet]*

True if *Term* is an RDF literal object. Currently only checks for groundness and the literal functor.

3.2.6 Loading and saving to file

The RDF library can read and write triples in RDF/XML and a proprietary binary format. There is a plugin interface defined to support additional formats. The `library(semweb/turtle)` uses this plugin API to support loading Turtle files using `rdf_load/2`.

rdf_load(+*FileOrList*) *[det]*

Same as `rdf_load(FileOrList, [])`. See `rdf_load/2`.

rdf_load(+*FileOrList, :Options*) *[det]*

Load RDF data. *Options* provides additional processing options. Defined options are:

blank_nodes(+*ShareMode*)
: How to handle equivalent blank nodes. If `share` (default), equivalent blank nodes are shared in the same resource.

base_uri(+*URI*)
: *URI* that is used for rdf:about="" and other RDF constructs that are relative to the base uri. Default is the source URL.

concurrent(+*Jobs*)
: If *FileOrList* is a list of files, process the input files using *Jobs* threads concurrently. Default is the mininum of the number of cores and the number of inputs. Higher values can be useful when loading inputs from (slow) network connections. Using 1 (one) does not use separate worker threads.

format(+*Format*)
: Specify the source format explicitly. Normally this is deduced from the filename extension or the mime-type. The core library understands the formats xml (RDF/XML) and triples (internal quick load and cache format). Plugins, such as `library(semweb/turtle)` extend the set of recognised extensions.

graph(?*Graph*)
: Named graph in which to load the data. It is **not** allowed to load two sources into the same named graph. If *Graph* is unbound, it is unified to the graph into which the data is loaded. The default graph is a =file://= URL when loading a file or, if the specification is a URL, its normalized version without the optional *#fragment*.

if(?*Condition*)
: When to load the file. One of `true`, `changed` (default) or `not_loaded`.

modified(-*Modified*)
: Unify *Modified* with one of `not_modified`, `cached(File)`, `last_modified(Stamp)` or `unknown`.

cache(+*Bool*)
: If `false`, do not use or create a cache file.

register_namespaces(+*Bool*)
: If `true` (default `false`), register `xmlns` namespace declarations or Turtle `@prefix` prefixes using `rdf_register_prefix/3` if there is no conflict.

silent(+*Bool*)
: If `true`, the message reporting completion is printed using level `silent`. Otherwise the level is `informational`. See also `print_message/2`.

Other options are forwarded to `process_rdf/3`. By default, `rdf_load/2` only loads RDF/XML from files. It can be extended to load data from other formats and locations using plugins. The full set of plugins relevant to support different formats and locations is below:

```
:- use_module(library(semweb/turtle)).        % Turtle and TRiG
:- use_module(library(semweb/rdf_ntriples)).
:- use_module(library(semweb/rdf_zlib_plugin)).
:- use_module(library(semweb/rdf_http_plugin)).
:- use_module(library(http/http_ssl_plugin)).
```

See also `rdf_db:rdf_open_hook/3`, `library(semweb/rdf_persistency)` and `library(semweb/rdf_cache)`

rdf_unload(+*Source*) *[det]*

Identify the graph loaded from *Source* and use `rdf_unload_graph/1` to erase this graph.

deprecated For compatibility, this predicate also accepts a graph name instead of a source specification. Please update your code to use `rdf_unload_graph/1`.

rdf_save(+*Out*) *[det]*

Same as `rdf_save(Out, [])`. See `rdf_save/2` for details.

rdf_save(+*Out, :Options*) *[det]*

Write RDF data as RDF/XML. *Options* is a list of one or more of the following options:

graph(+*Graph*)
Save only triples associated to the given named *Graph*.

anon(*Bool*)
If false (default true) do not save blank nodes that do not appear (indirectly) as object of a named resource.

base_uri(*URI*)
BaseURI used. If present, all URIs that can be represented relative to this base are written using their shorthand. See also `write_xml_base` option

convert_typed_literal(*:Convertor*)
Call *Convertor*(-Type, -Content, +RDFObject), providing the opposite for the convert_typed_literal option of the RDF parser.

document_language(+*Lang*)
Initial xml:lang saved with rdf:RDF element

encoding(*Encoding*)
Encoding for the output. Either utf8 or iso_latin_1

inline(+*Bool*)
If `true` (default `false`), inline resources when encountered for the first time. Normally, only bnodes are handled this way.

namespaces(+*List*)
Explicitely specify saved namespace declarations. See `rdf_save_header/2` option namespaces for details.

sorted(+*Boolean*)
If `true` (default `false`), emit subjects sorted on the full URI. Useful to make file comparison easier.

write_xml_base(*Bool*)
If `false`, do *not* include the `xml:base` declaration that is written normally when using the `base_uri` option.

xml_attributes(+*Bool*)
If `false` (default `true`), never use xml attributes to save plain literal attributes, i.e., always used an XML element as in `<name>Joe</name>`.

Arguments

Out Location to save the data. This can also be a file-url (`file://path`) or a stream wrapped in a term `stream(Out)`.

See also `rdf_save_db/1`

rdf_make
Reload all loaded files that have been modified since the last time they were loaded.

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(+*Fd, +Options*)
Save XML document header, doctype and open the RDF environment. This predicate also sets up the namespace notation.

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:

graph(+*URI*)
Only search for namespaces used in triples that belong to the given named graph.

namespaces(+*List*)
Where *List* is a list of namespace abbreviations. 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(*Out:stream*) *[det]*
Finish XML generation and write the document footer.

See also `rdf_save_header/2`, `rdf_save_subject/3`.

rdf_save_subject(+*Out, +Subject:resource, +Options*) *[det]*
Save the triples associated to *Subject* to *Out*. *Options*:

graph(+*Graph*)
Only save properties from *Graph*.

base_uri(+*URI*)

convert_typed_literal(*:Goal*)

document_language(+*XMLLang*)

See also `rdf_save/2` for a description of these options.

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/2` under certain conditions as well as for maintaining persistent snapshots of the database using `library(semweb/rdf_persistency)`.

rdf_save_db(+*File*) *[det]*
rdf_save_db(+*File, +Graph*) *[det]*
Save triples into *File* in a quick-to-load binary format. If *Graph* is supplied only triples flagged to originate from that database are added. Files created this way can be loaded using `rdf_load_db/1`.

rdf_save_db(+*File*) *[det]*
rdf_save_db(+*File, +Graph*) *[det]*
Save triples into *File* in a quick-to-load binary format. If *Graph* is supplied only triples flagged to originate from that database are added. Files created this way can be loaded using `rdf_load_db/1`.

rdf_load_db(+*File*) *[det]*
Load triples from a file created using `rdf_save_db/2`.

3.2.7 Graph manipulation

Many RDF stores turned triples into quadruples. This store is no exception, initially using the 4th argument to store the filename from which the triple was loaded. Currently, the 4th argument is the RDF *named graph*. A named graph maintains some properties, notably to track origin, changes and modified state.

rdf_create_graph(+*Graph*) *[det]*
Create an RDF graph without triples. Succeeds silently if the graph already exists.

rdf_unload_graph(+*Graph*) *[det]*
Remove *Graph* from the RDF store. Succeeds silently if the named graph does not exist.

rdf_graph_property(*?Graph, ?Property*) *[nondet]*
True when *Property* is a property of *Graph*. Defined properties are:

hash(*Hash*)
Hash is the (MD5-)hash for the content of *Graph*.

modified(*Boolean*)
True if the graph is modified since it was loaded or `rdf_set_graph/2` was called with `modified(false)`.

source(*Source*)
The graph is loaded from the *Source* (a URL)

source_last_modified(*?Time*)
Time is the last-modified timestamp of Source at the moment that the graph was loaded from Source.

triples(*Count***)**
True when *Count* is the number of triples in *Graph*.

Additional graph properties can be added by defining rules for the multifile predicate `property_of_graph/2`. Currently, the following extensions are defined:

- `library(semweb/rdf_persistency)`

  persistent(*Boolean***)**
  Boolean is `true` if the graph is persistent.

rdf_set_graph(*+Graph, +Property***)** *[det]*
Set properties of *Graph*. Defined properties are:

modified(*false***)**
Set the modified state of *Graph* to false.

3.2.8 Literal matching and indexing

Literal values are ordered and indexed using a *skip list* The aim of this index is threefold.

- Unlike hash-tables, binary trees allow for efficient *prefix* and *range* matching. Prefix matching is 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'. See `library(semweb/litindex)`.

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.

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 queries that use the tree index are returned in alphabetical order.

3.2.9 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. Alo note that there is no relation to defined RDF properties. Properties that have no triples are not reported by this predicate, while predicates that are involved in triples do not need to be defined as an instance of rdf:Property.

rdf_set_predicate(+*Predicate, +Property*) *[det]*

Define a property of the predicate. This predicate currently supports the following properties:

symmetric(+*Boolean*)

Set/unset the predicate as being symmetric. Using `symmetric(true)` is the same as `inverse_of(Predicate)`, i.e., creating a predicate that is the inverse of itself.

transitive(+*Boolean*)

Sets the transitive property.

inverse_of(+*Predicate2*)

Define *Predicate* as the inverse of *Predicate2*. An inverse relation is deleted using `inverse_of([])`.

The `transitive` property is currently not used. The `symmetric` and `inverse_of` properties are considered by `rdf_has/3`,4 and `rdf_reachable/3`.

To be done Maintain these properties based on OWL triples.

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}. Setting symmetric is equivalent to `inverse_of(Self)`.

inverse_of(*Inverse*)

True if this predicate is the inverse of *Inverse*. This property is used by `rdf_has/3`, `rdf_has/4`, `rdf_reachable/3` and `rdf_reachable/5`.

transitive(*Bool*)

True if this predicate is transitive. This predicate is currently not used. It might be used to make `rdf_has/3` imply `rdf_reachable/3` for transitive predicates.

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 intended 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`, but also considering triples of 'subPropertyOf' this relation. See also `rdf_has/3`.

rdfs_object_branch_factor(-*Float*)

Same as `rdf_object_branch_factor`, but also considering triples of 'subPropertyOf' this relation. See also `rdf_has/3`.

See also `rdf_set_predicate/2`.

3.2.10 Prefix Handling

Prolog code often contains references to constant resources with a known *prefix* (also known as XML *namespaces*). For example, `http://www.w3.org/2000/01/rdf-schema#Class` refers to the most general notion of an RDFS class. Readability and maintability concerns require for abstraction here. The RDF database maintains a table of known *prefixes*. This table can be queried using `rdf_current_ns/2` and can be extended using `rdf_register_ns/3`. The prefix database is used to expand `prefix:local` terms that appear as arguments to calls which are known to accept a *resource*. This expansion is achieved by Prolog preprocessor using `expand_goal/2`.

rdf_current_prefix(*:Alias, ?URI*) *[nondet]*

Query predefined prefixes and prefixes defined with `rdf_register_prefix/2` and local prefixes defined with `rdf_prefix/2`. If *Alias* is unbound and one *URI* is the prefix of another, the longest is returned first. This allows turning a resource into a prefix/local couple using the simple enumeration below. See `rdf_global_id/2`.

```
rdf_current_prefix(Prefix, Expansion),
atom_concat(Expansion, Local, URI),
```

rdf_register_prefix(+*Prefix, +URI*) *[det]*

rdf_register_prefix(+*Prefix, +URI, +Options*) *[det]*

Register *Prefix* as an abbreviation for *URI*. *Options*:

force(*Boolean*)

If `true`, Replace existing namespace alias. Please note that replacing a namespace is dangerous as namespaces affect preprocessing. Make sure all code that depends on a namespace is compiled after changing the registration.

keep(*Boolean*)

If `true` and Alias is already defined, keep the original binding for *Prefix* and succeed silently.

Without options, an attempt to redefine an alias raises a permission error.

Predefined prefixes are:

Alias	IRI prefix
dc	http://purl.org/dc/elements/1.1/
dcterms	http://purl.org/dc/terms/
eor	http://dublincore.org/2000/03/13/eor#
foaf	http://xmlns.com/foaf/0.1/
owl	http://www.w3.org/2002/07/owl#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
serql	http://www.openrdf.org/schema/serql#
skos	http://www.w3.org/2004/02/skos/core#
void	http://rdfs.org/ns/void#
xsd	http://www.w3.org/2001/XMLSchema#

rdf_register_prefix(+*Prefix*, +*URI*) *[det]*
rdf_register_prefix(+*Prefix*, +*URI*, +*Options*) *[det]*

Register *Prefix* as an abbreviation for *URI*. *Options*:

force(*Boolean*)

If `true`, Replace existing namespace alias. Please note that replacing a namespace is dangerous as namespaces affect preprocessing. Make sure all code that depends on a namespace is compiled after changing the registration.

keep(*Boolean*)

If `true` and Alias is already defined, keep the original binding for *Prefix* and succeed silently.

Without options, an attempt to redefine an alias raises a permission error.

Predefined prefixes are:

Alias	IRI prefix
dc	http://purl.org/dc/elements/1.1/
dcterms	http://purl.org/dc/terms/
eor	http://dublincore.org/2000/03/13/eor#
foaf	http://xmlns.com/foaf/0.1/
owl	http://www.w3.org/2002/07/owl#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
serql	http://www.openrdf.org/schema/serql#
skos	http://www.w3.org/2004/02/skos/core#
void	http://rdfs.org/ns/void#
xsd	http://www.w3.org/2001/XMLSchema#

Explicit expansion is achieved using the predicates below. The predicate `rdf_equal/2` performs this expansion at compile time, while the other predicates do it at runtime.

rdf_equal(*?Resource1, ?Resource2*)

Simple equality test to exploit goal-expansion

rdf_global_id(*?IRISpec, :IRI*) *[semidet]*

Convert between Prefix:Local and full *IRI* (an atom). If *IRISpec* is an atom, it is simply unified with *IRI*. This predicate fails silently if *IRI* is an RDF literal.

Note that this predicate is a meta-predicate on its output argument. This is necessary to get the module context while the first argument may be of the form (:)/2. The above mode description is correct, but should be interpreted as (?,?).

Errors `existence_error(rdf_prefix, Prefix)`

See also

- `rdf_equal/2` provides a compile time alternative
- The `rdf_meta/1` directive asks for compile time expansion of arguments.

bug Error handling is incomplete. In its current implementation the same code is used for compile-time expansion and to facilitate runtime conversion and checking. These use cases have different requirements.

rdf_global_object(*+Object, :GlobalObject*) *[semidet]*

rdf_global_object(*-Object, :GlobalObject*) *[semidet]*

Same as `rdf_global_id/2`, but intended for dealing with the object part of a triple, in particular the type for typed literals. Note that the predicate is a meta-predicate on the output argument. This is necessary to get the module context while the first argument may be of the form (:)/2.

Errors `existence_error(rdf_prefix, Prefix)`

rdf_global_term(*+TermIn, :GlobalTerm*) *[det]*

Does `rdf_global_id/2` on all terms NS:Local by recursively analysing the term. Note that the predicate is a meta-predicate on the output argument. This is necessary to get the module context while the first argument may be of the form (:)/2.

Terms of the form Prefix:Local that appear in *TermIn* for which Prefix is not defined are not replaced. Unlike `rdf_global_id/2` and `rdf_global_object/2`, no error is raised.

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 because `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/1]

The example below defines the rule `concept/1`.

```
:- use_module(library(semweb/rdf_db)).  % for rdf_meta
:- use_module(library(semweb/rdfs)).    % for rdfs_individual_of

:- rdf_meta
        concept(r).

%%      concept(?C) is nondet.
%
%       True if C is a concept.

concept(C) :-
        rdfs_individual_of(C, skos:'Concept').
```

In addition to expanding *calls*, `rdf_meta/1` also causes expansion of *clause heads* for predicates that match a declaration. This is typically used write Prolog statements about resources. The following example produces three clauses with expanded (single-atom) arguments:

```
:- use_module(library(semweb/rdf_db)).

:- rdf_meta
        label_predicate(r).

label_predicate(rdfs:label).
label_predicate(skos:prefLabel).
label_predicate(skos:altLabel).
```

3.2.11 Miscellaneous predicates

This section describes the remaining predicates of the `library(semweb/rdf_db)` module.

rdf_bnode(*-Id*)
Generate a unique anonymous identifier for a subject.

rdf_source_location(*+Subject, -Location*) *[nondet]*
True when triples for *Subject* are loaded from *Location*.

Arguments

Location	is a term File:Line.

rdf_generation(*-Generation*) *[det]*

True when *Generation* is the current 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. Committing a non-empty transaction increments the generation by one.

When inside a transaction, *Generation* is unified to a term *TransactionStartGen + InsideTransactionGen*. E.g., 4+3 means that the transaction was started at generation 4 of the global database and we have created 3 new generations inside the transaction. Note that this choice of representation allows for comparing generations using Prolog arithmetic. Comparing a generation in one transaction with a generation in another transaction is meaningless.

rdf_estimate_complexity(*?Subject, ?Predicate, ?Object, -Complexity*)

Return 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 optimizers.

rdf_statistics(*?KeyValue*) *[nondet]*

Obtain statistics on the RDF database. Defined statistics are:

graphs(*-Count*)
: Number of named graphs

triples(*-Count*)
: Total number of triples in the database. This is the number of asserted triples minus the number of retracted ones. The number of *visible* triples in a particular context may be different due to visibility rules defined by the logical update view and transaction isolation.

resources(*-Count*)
: Number of resources that appear as subject or object in a triple. See `rdf_resource/1`.

properties(*-Count*)
: Number of current predicates. See `rdf_current_predicate/1`.

literals(*-Count*)
: Number of current literals. See `rdf_current_literal/1`.

gc(*GCCount, ReclaimedTriples, ReindexedTriples, Time*)
: Information about the garbage collector.

searched_nodes(*-Count*)
: Number of nodes expanded by `rdf_reachable/3` and `rdf_reachable/5`.

lookup(*rdf(S,P,O,G), Count*)
: Number of queries for this particular instantiation pattern. Each of *S*,*P*,*O*,*G* is either + or -.

hash_quality(*rdf(S,P,O,G), Buckets, Quality, PendingResize*)
: Statistics on the index for this pattern. Indices are created lazily on the first relevant query.

triples_by_graph(*Graph, Count*)
: This statistics is produced for each named graph. See `triples` for the interpretation of this value.

rdf_match_label(+*How, +Pattern, +Label*) *[semidet]*

True if *Label* matches *Pattern* according to *How*. *How* is one of `icase`, `substring`, `word`, `prefix` or `like`. For backward compatibility, `exact` is a synonym for `icase`.

lang_matches(+*Lang, +Pattern*) *[semidet]*

True if *Lang* matches *Pattern*. This implements XML language matching conform RFC 4647. Both *Lang* and *Pattern* are dash-separated strings of identifiers or (for *Pattern*) the wildcart *. Identifiers are matched case-insensitive and a * matches any number of identifiers. A short pattern is the same as *.

lang_equal(+*Lang1, +Lang2*) *[semidet]*

True if two RFC language specifiers denote the same language

See also `lang_matches/2`.

rdf_reset_db

Remove all triples from the RDF database and reset all its statistics.

bug This predicate checks for active queries, but this check is not properly synchronized and therefore the use of this predicate is unsafe in multi-threaded contexts. It is mainly used to run functionality tests that need to start with an empty database.

rdf_version(-*Version*) *[det]*

True when *Version* is the numerical version-id of this library. The version is computed as

```
Major*10000 + Minor*100 + Patch.
```

3.2.12 Memory management considerations

Storing RDF triples in main memory provides much better performance than using external databases. Unfortunately, although memory is fairly cheap these days, main memory is severely limited when compared to disks. Memory usage breaks down to the following categories. Rough estimates of the memory usage is given **for 64-bit systems**. 32-bit system use slightly more than half these amounts.

- Actually storing the triples. A triple is stored in a C struct of 144 bytes. This struct both holds the quintuple, some bookkeeping information and the 10 next-pointers for the (max) to hash tables.
- The bucket array for the hashes. Each bucket maintains a *head*, and *tail* pointer, as well as a count for the number of entries. The bucket array is allocated if a particular index is created, which implies the first query that requires the index. Each bucket requires 24 bytes.

  Bucket arrays are resized if necessary. Old triples remain at their original location. This implies that a query may need to scan multiple buckets. The garbage collector may relocate old indexed triples. It does so by copying the old triple. The old triple is later reclaimed by GC. Reindexed triples will be reused, but many reindexed triples may result in a significant memory fragmentation.
- Resources are maintained in a seperate table to support `rdf_resource/1`. A resources requires approximately 32 bytes.

- Identical literals are shared (see `rdf_current_literal/1`) and stored in a *skip list*. A literal requires approximately 40 bytes, excluding the atom used for the lexical representation.
- Resources are stored in the Prolog atom-table. Atoms with the average length of a resource require approximately 88 bytes.

The hash parameters can be controlled with `rdf_set/1`. Applications that are tight on memory and for which the query characteristics are more or less known can optimize performance and memory by fixing the hash-tables. By fixing the hash-tables we can tailor them to the frequent query patterns, we avoid the need for to check multiple hash buckets (see above) and we avoid memory fragmentation due to optimizing triples for resized hashes.

```
set_hash_parameters :-
      rdf_set(hash(s,   size, 1048576)),
      rdf_set(hash(p,   size, 1024)),
      rdf_set(hash(sp,  size, 2097152)),
      rdf_set(hash(o,   size, 1048576)),
      rdf_set(hash(po,  size, 2097152)),
      rdf_set(hash(spo, size, 2097152)),
      rdf_set(hash(g,   size, 1024)),
      rdf_set(hash(sg,  size, 1048576)),
      rdf_set(hash(pg,  size, 2048)).
```

rdf_set(+*Term***)** *[det]*

Set properties of the RDF store. Currently defines:

hash(+*Hash, +Parameter, +Value***)**

Set properties for a triple index. *Hash* is one of `s`, `p`, `sp`, `o`, `po`, `spo`, `g`, `sg` or `pg`. *Parameter* is one of:

size

Value defines the number of entries in the hash-table. *Value* is rounded *down* to a power of 2. After setting the size explicitly, auto-sizing for this table is disabled. Setting the size smaller than the current size results in a `permission_error` exception.

average_chain_len

Set maximum average collision number for the hash.

optimize_threshold

Related to resizing hash-tables. If 0, all triples are moved to the new size by the garbage collector. If more then zero, those of the last *Value* resize steps remain at their current location. Leaving cells at their current location reduces memory fragmentation and slows down access.

The garbage collector The RDF store has a garbage collector that runs in a separate thread named =__rdf_GC=. The garbage collector removes the following objects:

- Triples that have died before the the generation of last still active query.

- Entailment matrices for `rdfs:subPropertyOf` relations that are related to old queries.

In addition, the garbage collector reindexes triples associated to the hash-tables before the table was resized. The most recent resize operation leads to the largest number of triples that require reindexing, while the oldest resize operation causes the largest slowdown. The parameter `optimize_threshold` controlled by `rdf_set/1` can be used to determine the number of most recent resize operations for which triples will not be reindexed. The default is 2.

Normally, the garbage collector does it job in the background at a low priority. The predicate `rdf_gc/0` can be used to reclaim all garbage and optimize all indexes.

Warming up the database The RDF store performs many operations lazily or in background threads. For maximum performance, perform the following steps:

- Load all the data without doing queries or retracting data in between. This avoids creating the indexes and therefore the need to resize them.
- Perform each of the indexed queries. The following call performs this. Note that it is irrelevant whether or not the query succeeds.

```
warm_indexes :-
    ignore(rdf(s, _, _)),
    ignore(rdf(_, p, _)),
    ignore(rdf(_, _, o)),
    ignore(rdf(s, p, _)),
    ignore(rdf(_, p, o)),
    ignore(rdf(s, p, o)),
    ignore(rdf(_, _, _, g)),
    ignore(rdf(s, _, _, g)),
    ignore(rdf(_, p, _, g)).
```

- Duplicate adminstration is initialized in the background after the first call that returns a significant amount of duplicates. Creating the adminstration can be forced by calling `rdf_update_duplicates/0`.

Predicates:

rdf_gc *[det]*

Run the RDF-DB garbage collector until no garbage is left and all tables are fully optimized. Under normal operation a seperate thread with identifier =_rdf_GC= performs garbage collection as long as it is considered 'useful'.

Using `rdf_gc/0` should only be needed to ensure a fully clean database for analysis purposes such as leak detection.

rdf_update_duplicates *[det]*

Update the duplicate administration of the RDF store. This marks every triple that is potentionally a duplicate of another as duplicate. Being potentially a duplicate means that subject, predicate and object are equivalent and the life-times of the two triples overlap.

The duplicates marks are used to reduce the administrative load of avoiding duplicate answers. Normally, the duplicates are marked using a background thread that is started on the first query that produces a substantial amount of duplicates.

3.3 Monitoring the database

The predicate `rdf_monitor/2` allows registrations of call-backs with the RDF store. These call-backs are typically used to keep other databases in sync with the RDF store. For example, `library(semweb/rdf_persistency)` monitors the RDF store for maintaining a persistent copy in a set of files and `library(semweb/rdf_litindex)` uses added and deleted literal values to maintain a fulltext index of literals.

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 transaction started by `rdf_transaction/2`. *BeginOrEnd* is `begin`(*Nesting*) or `end`(*Nesting*). *Nesting* expresses the nesting level of transactions, starting at '0' for a toplevel transaction. *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`.

3.4 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 Plugin modules for rdf_db

The `rdf_db` module provides several hooks for extending its functionality. Database updates can be monitored and acted upon through the features described in section 3.3. The predicate `rdf_load/2` can be hooked to deal with different formats such as *rdfturtle*, different input sources (e.g. http) and different strategies for caching results.

4.1 Hooks into the RDF library

The hooks below are used to add new RDF file formats and sources from which to load data to the library. They are used by the modules described below and distributed with the package. Please examine the source-code if you want to add new formats or locations.

`library(semweb/turtle)` Load files in the Turtle format. See section **??**.

library(semweb/rdf_zlib_plugin) Load `gzip` compressed files transparently. See section 4.2.

library(semweb/rdf_http_plugin) Load RDF documents from HTTP servers. See section 4.3.

library(http/http_ssl_plugin) May be combined with `library(semweb/rdf_http_plugin)` to load RDF from HTTPS servers.

library(semweb/rdf_persistency) Provide persistent backup of the triple store.

library(semweb/rdf_cache) Provide caching RDF sources using fast load/safe files to speedup restarting an application.

rdf_db:rdf_open_hook(+*Input, -Stream, -Format*)
Open an input. *Input* is one of `file`(+*Name*), `stream`(+*Stream*) or `url`(*Protocol, URL*). If this hook succeeds, the RDF will be read from Stream using `rdf_load_stream/3`. Otherwise the default open functionality for file and stream are used.

rdf_db:rdf_load_stream(+*Format, +Stream, +Options*)
Actually load the RDF from *Stream* into the RDF database. *Format* describes the format and is produced either by `rdf_input_info/3` or `rdf_file_type/2`.

rdf_db:rdf_input_info(+*Input, -Modified, -Format*)
Gather information on *Input*. *Modified* is the last modification time of the source as a POSIX time-stamp (see `time_file/2`). *Format* is the RDF format of the file. See `rdf_file_type/2` for details. It is allowed to leave the output variables unbound. Ultimately the default modified time is '0' and the format is assumed to be `xml`.

rdf_db:rdf_file_type(*?Extension, ?Format*)
True if *Format* is the default RDF file format for files with the given extension. *Extension* is lowercase and without a '.'. E.g. `owl`. *Format* is either a built-in format (`xml` or `triples`) or a format understood by the `rdf_load_stream/3` hook.

rdf_db:url_protocol(*?Protocol*)
True if *Protocol* is a URL protocol recognised by `rdf_load/2`.

4.2 library(semweb/rdf_zlib_plugin): Reading compressed RDF

This module uses the `zlib` library to load compressed files on the fly. The extension of the file must be `.gz`. The file format is deduced by the extension after stripping the `.gz` extension. E.g. `rdf_load('file.rdf.gz')`.

4.3 library(semweb/rdf_http_plugin): Reading RDF from a HTTP server

This module allows for `rdf_load('http://...')`. It exploits the library `http/http_open.pl`. The format of the URL is determined from the mime-type returned by the server if this is one of `text/rdf+xml`, `application/x-turtle` or `application/turtle`. As RDF mime-types are not yet widely supported, the plugin uses the extension of the URL if the claimed mime-type is not one of the above. In addition, it recognises `text/html` and `application/xhtml+xml`, scanning the XML content for embedded RDF.

4.4 library(semweb/rdf_cache): Cache RDF triples

The library `library(semweb/rdf_cache)` defines the caching strategy for triples sources. When using large RDF sources, caching triples greatly speedup loading RDF documents. The cache library implements two caching strategies that are controlled by `rdf_set_cache_options/1`.

Local caching This approach applies to files only. Triples are cached in a sub-directory of the directory holding the source. This directory is called `.cache` (`_cache` on Windows). If the cache option `create_local_directory` is `true`, a cache directory is created if posible.

Global caching This approach applies to all sources, except for unnamed streams. Triples are cached in directory defined by the cache option `global_directory`.

When loading an RDF file, the system scans the configured cache files unless `cache(false)` is specified as option to `rdf_load/2` or caching is disabled. If caching is enabled but no cache exists, the system will try to create a cache file. First it will try to do this locally. On failure it will try to configured global cache.

rdf_set_cache_options(+*Options*)

Change the cache policy. Provided options are:

- `enabled(Boolean)` If `true`, caching is enabled.
- `local_directory(Name)`. Plain name of local directory. Default `.cache` (`_cache` on Windows).
- `create_local_directory(Bool)` If `true`, try to create local cache directories
- `global_directory(Dir)` Writeable directory for storing cached parsed files.
- `create_global_directory(Bool)` If `true`, try to create the global cache directory.

rdf_cache_file(+*URL, +ReadWrite, -File*) *[semidet]*

File is the cache file for *URL*. If *ReadWrite* is `read`, it returns the name of an existing file. If `write` it returns where a new cache file can be overwritten or created.

4.5 library(semweb/rdf_litindex): Indexing words in literals

The library `semweb/rdf_litindex.pl` exploits the primitives of section 4.5.1 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`. Numerical indexing is currently limited to integers in the range $\pm 2^30$ ($\pm 2^62 on 64 - bitplatforms$). *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.

between(*Low, High*)
: Matches all literals containing an integer token in the range *Low*..*High*, including the boundaries.

ge(*Low*)
: Matches all literals containing an integer token with value *Low* or higher.

le(*High*)
: Matches all literals containing an integer token with value *High* or lower.

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[2] 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:

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

```
?- 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:

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

4.5.1 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 4.5. 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_insert_literal_map(+*Map, +Key, +Value, -KeyCount*)
As `rdf_insert_literal_map/3`. In addition, if *Key* is a new key in *Map*, unify *KeyCount* with the number of keys in *Map*. This serves two purposes. Derived maps, such as the stem and metaphone maps need to know about new keys and it avoids additional foreign calls for doing the progress in `rdf_litindex.pl`.

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*) *[det]*
Unify *ValueList* with an ordered set of values associated to all keys from *KeyList*. Each key in *KeyList* is either an atom, an integer or a term `not`(*Key*). If not-terms are provided, there must be at least one positive keywords. The negations are tested after establishing the positive matches.

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. *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.

4.6 library(semweb/rdf_persistency): Providing persistent storage

The `semweb/rdf_persistency` provides reliable persistent storage for the RDF data. The store uses a directory with files for each source (see `rdf_source/1`) present in the database. Each source

is represented by two files, one in binary format (see `rdf_save_db/2`) representing the base state and one represented as Prolog terms representing the changes made since the base state. The latter is called the *journal*.

rdf_attach_db(+*Directory, +Options*)
Attach *Directory* as the persistent database. If *Directory* does not exist it is created. Otherwise all sources defined in the directory are loaded into the RDF database. Loading a source means loading the base state (if any) and replaying the journal (if any). The current implementation does not synchronise triples that are in the store before attaching a database. They are not removed from the database, nor added to the presistent store. Different merging options may be supported through the *Options* argument later. Currently defined options are:

concurrency(+*PosInt*)
Number of threads used to reload databased and journals from the files in *Directory*. Default is the number of physical CPUs determined by the Prolog flag `cpu_count` or 1 (one) on systems where this number is unknown. See also `concurrent/3`.

max_open_journals(+*PosInt*)
The library maintains a pool of open journal files. This option specifies the size of this pool. The default is 10. Raising the option can make sense if many writes occur on many different named graphs. The value can be lowered for scenarios where write operations are very infrequent.

silent(*Boolean*)
If `true`, supress loading messages from `rdf_attach_db/2`.

log_nested_transactions(*Boolean*)
If `true`, nested *log* transactions are added to the journal information. By default (`false`), no log-term is added for nested transactions.

The database is locked against concurrent access using a file `lock` in *Directory*. An attempt to attach to a locked database raises a `permission_error` exception. The error context contains a term `rdf_locked`(*Args*), where args is a list containing `time`(*Stamp*) and `pid`(*PID*). The error can be caught by the application. Otherwise it prints:

```
ERROR: No permission to lock rdf_db `/home/jan/src/pl/packages/semweb/DB'
ERROR: locked at Wed Jun 27 15:37:35 2007 by process id 1748
```

rdf_detach_db
Detaches the persistent store. No triples are removed from the RDF triple store.

rdf_current_db(-*Directory*)
Unify *Directory* with the current database directory. Fails if no persistent database is attached.

rdf_persistency(+*DB, +Bool*)
Change presistency of named database (4th argument of `rdf/4`). By default all databases are presistent. Using `false`, the journal and snapshot for the database are deleted and further changes to triples associated with *DB* are not recorded. If *Bool* is `true` a snapshot is created for the current state and further modifications are monitored. Switching persistency does not affect the triples in the in-memory RDF database.

rdf_flush_journals(+*Options*)

Flush dirty journals. With the option `min_size`(*KB*) only journals larger than *KB* Kbytes are merged with the base state. Flushing a journal takes the following steps, ensuring a stable state can be recovered at any moment.

1. Save the current database in a new file using the extension `.new`.
2. On success, delete the journal
3. On success, atomically move the `.new` file over the base state.

Note that journals are *not* merged automatically for two reasons. First of all, some applications may decide never to merge as the journal contains a complete *changelog* of the database. Second, merging large databases can be slow and the application may wish to schedule such actions at quiet times or scheduled maintenance periods.

4.6.1 Enriching the journals

The above predicates suffice for most applications. The predicates in this section provide access to the journal files and the base state files and are intented to provide additional services, such as reasoning about the journals, loaded files, etc.[3]

Using `rdf_transaction`(*Goal, log(Message)*), we can add additional records to enrich the journal of affected databases with *Term* and some additional bookkeeping information. Such a transaction adds a term `begin`(*Id, Nest, Time, Message*) before the change operations on each affected database and `end`(*Id, Nest, Affected*) after the change operations. Here is an example call and content of the journal file `mydb.jrn`. A full explanation of the terms that appear in the journal is in the description of `rdf_journal_file/2`.

```
?- rdf_transaction(rdf_assert(s,p,o,mydb), log(by(jan))).
```

```
start([time(1183540570)]).
begin(1, 0, 1183540570.36, by(jan)).
assert(s, p, o).
end(1, 0, []).
end([time(1183540578)]).
```

Using `rdf_transaction`(*Goal, log(Message, DB)*), where *DB* is an atom denoting a (possibly empty) named graph, the system guarantees that a non-empty transaction will leave a possibly empty transaction record in DB. This feature assumes named graphs are named after the user making the changes. If a user action does not affect the user's graph, such as deleting a triple from another graph, we still find record of all actions performed by some user in the journal of that user.

rdf_journal_file(*?DB, ?JournalFile*)

True if *File* is the absolute file name of an existing named graph *DB*. A journal file contains a sequence of Prolog terms of the following format.[4]

[3] A library `rdf_history` is under development exploiting these features supporting wiki style editing of RDF.

[4] Future versions of this library may use an XML based language neutral format.

start(*Attributes*)
Journal has been opened. Currently *Attributes* contains a term `time`(*Stamp*).

end(*Attributes*)
Journal was closed. Currently *Attributes* contains a term `time`(*Stamp*).

assert(*Subject, Predicate, Object*)
A triple {Subject, Predicate, Object} was added to the database.

assert(*Subject, Predicate, Object, Line*)
A triple {Subject, Predicate, Object} was added to the database with given *Line* context.

retract(*Subject, Predicate, Object*)
A triple {Subject, Predicate, Object} was deleted from the database. Note that an `rdf_retractall/3` call can retract multiple triples. Each of them have a record in the journal. This allows for 'undo'.

retract(*Subject, Predicate, Object, Line*)
Same as above, for a triple with associated line info.

update(*Subject, Predicate, Object, Action*)
See `rdf_update/4`.

begin(*Id, Nest, Time, Message*)
Added before the changes in each database affected by a transaction with transaction identifier `log`(*Message*). *Id* is an integer counting the logged transactions to this database. Numbers are increasing and designed for binary search within the journal file. *Nest* is the nesting level, where '0' is a toplevel transaction. *Time* is a time-stamp, currently using float notation with two fractional digits. *Message* is the term provided by the user as argument of the `log`(*Message*) transaction.

end(*Id, Nest, Others*)
Added after the changes in each database affected by a transaction with transaction identifier `log`(*Message*). *Id* and *Nest* match the begin-term. *Others* gives a list of other databases affected by this transaction and the *Id* of these records. The terms in this list have the format *DB*:*Id*.

rdf_db_to_file(*?DB, ?FileBase*)
Convert between *DB* (see `rdf_source/1`) and file base-file used for storing information on this database. The full file is located in the directory described by `rdf_current_db/1` and has the extension `.trp` for the base state and `.jrn` for the journal.

5 library(semweb/turtle): Turtle: Terse RDF Triple Language

See also http://www.w3.org/TR/turtle/ (used W3C Recommendation 25 February 2014)

This module implements the Turtle language for representing the RDF triple model as defined by Dave Beckett from the Institute for Learning and Research Technology University of Bristol and later standardized by the W3C RDF working group.

This module acts as a plugin to `rdf_load/2`, for processing files with one of the extensions `.ttl` or `.n3`.

rdf_read_turtle(+*Input, -Triples, +Options*)

Read a stream or file into a set of triples or quadruples (if faced with TRiG input) of the format

```
rdf(Subject, Predicate, Object [, Graph])
```

The representation is consistent with the SWI-Prolog RDF/XML and ntriples parsers. Provided options are:

base_uri(+*BaseURI*)
Initial base URI. Defaults to file://<file> for loading files.

anon_prefix(+*Prefix*)
Blank nodes are generated as <*Prefix*>1, <*Prefix*>2, etc. If *Prefix* is not an atom blank nodes are generated as `node(1)`, `node(2)`, ...

format(+*Format*)
One of `auto` (default), `turtle` or `trig`. The auto mode switches to TRiG format of there is a `{` before the first triple. Finally, of the format is explicitly stated as `turtle` and the file appears to be a TRiG file, a warning is printed and the data is loaded while ignoring the graphs.

resources(*URIorIRI*)
Officially, Turtle resources are IRIs. Quite a few applications however send URIs. By default we do URI->IRI mapping because this rarely causes errors. To force strictly conforming mode, pass `iri`.

prefixes(-*Pairs*)
Return encountered prefix declarations as a list of Alias-URI

namespaces(-*Pairs*)
Same as `prefixes(Pairs)`. Compatibility to `rdf_load/2`.

base_used(-*Base*)
Base URI used for processing the data. Unified to `[]` if there is no base-uri.

on_error(+*ErrorMode*)
In `warning` (default), print the error and continue parsing the remainder of the file. If `error`, abort with an exception on the first error encountered.

error_count(-*Count*)
If `on_error(warning)` is active, this option cane be used to retrieve the number of generated errors.

Arguments

Input	is one of `stream(Stream)`, `atom(Atom)`, a `http`, `https` or `file` url or a filename specification as accepted by `absolute_file_name/3`.

rdf_load_turtle(+*Input, -Triples, +Options*) **deprecated**
Use `rdf_read_turtle/3`

rdf_process_turtle(+*Input, :OnObject, +Options*) *[det]*

Streaming Turtle parser. The predicate `rdf_process_turtle/3` processes Turtle data

from *Input*, calling *OnObject* with a list of triples for every Turtle *statement* found in *Input*. *OnObject* is called as below, where *ListOfTriples* is a list of `rdf(S,P,O)` terms for a normal Turtle file or `rdf(S,P,O,G)` terms if the `GRAPH` keyword is used to associate a set of triples in the document with a particular graph. The *Graph* argument provides the default graph for storing the triples and *Line* is the line number where the statement started.

```
call(OnObject, ListOfTriples, Graph:Line)
```

This predicate supports the same *Options* as `rdf_load_turtle/3`.

Errors encountered are sent to `print_message/2`, after which the parser tries to recover and parse the remainder of the data.

See also This predicate is normally used by `load_rdf/2` for processing RDF data.

rdf_db:rdf_load_stream(+*Format, +Stream, :Options*) *[multifile]*
(Turtle clauses)

rdf_save_ntriples(+*Spec, :Options*) *[det]*
Save RDF using ntriples format. The ntriples format is a subset of Turtle, writing each triple fully qualified on its own line.

rdf_save_canonical_trig(+*Spec, :Options*) *[det]*
Save triples in a canonical format. See `rdf_save_canonical_turtle/2` foir details.

rdf_save_trig(+*Spec, :Options*) *[det]*
Save multiple RDF graphs into a TriG file. *Options* are the same as for `rdf_save_turtle/2`. `rdf_save_trig/2` ignores the `graph(+Graph)` option and instead processes one additional option:

graphs(+*ListOfGraphs*)
List of graphs to save. When omitted, all graphs in the RDF store are stored in the TriG file.

rdf_save_canonical_turtle(+*Spec, :Options*) *[det]*
Save triples in a canonical format. This is the same as `rdf_save_turtle/2`, but using different defaults. In particular:

- `encoding(utf8)`,
- `indent(0)`,
- `tab_distance(0)`,
- `subject_white_lines(1)`,
- `align_prefixes(false)`,
- `user_prefixes(false)`
- `comment(false)`,
- `group(false)`,
- `single_line_bnodes(true)`

To be done Work in progress. Notably blank-node handling is incomplete.

rdf_save_turtle(+*Out, :Options*) *[det]*

Save an RDF graph as Turtle. *Options* processed are:

a(+*Boolean*)
If `true` (default), use `a` for the predicate `rdf:type`. Otherwise use the full resource.

align_prefixes(+*Boolean*)
Nicely align the @prefix declarations

base(+*Base*)
Save relative to the given *Base*

canonize_numbers(+*Boolean*)
If `true` (default `false`), emit numeric datatypes using Prolog's write to achieve canonical output.

comment(+*Boolean*)
It `true` (default), write some informative comments between the output segments

encoding(+*Encoding*)
Encoding used for the output stream. Default is UTF-8.

expand(*:Goal*)
Query an alternative graph-representation. See below.

indent(+*Column*)
Indentation for ; -lists. '0' does not indent, but writes on the same line. Default is 8.

graph(+*Graph*)
Save only the named graph

group(+*Boolean*)
If `true` (default), using P-O and O-grouping.

inline_bnodes(+*Boolean*)
if `true` (default), inline bnodes that are used once.

abbreviate_literals(+*Boolean*)
if `true` (default), omit the type if allowed by turtle.

only_known_prefixes(+*Boolean*)
Only use prefix notation for known prefixes. Without, some documents produce *huge* amounts of prefixes.

prefixes(+*List*)
If provided, uses exactly these prefixes. *List* is a list of prefix specifications, where each specification is either a term *Prefix-URI* or a prefix that is known to `rdf_current_prefix/2`.

silent(+*Boolean*)
If `true` (default `false`), do not print the final informational message.

single_line_bnodes(+*Bool*)
If `true` (default `false`), write [...] and (...) on a single line.

subject_white_lines(+*Count*)
Extra white lines to insert between statements about a different subject. Default is 1.

tab_distance(+*Tab*)
Distance between tab-stops. '0' forces the library to use only spaces for layout. Default is 8.

user_prefixes(+*Boolean*)
If `true` (default), use prefixes from `rdf_current_prefix/2`.

The option `expand` allows for serializing alternative graph representations. It is called through `call/5`, where the first argument is the expand-option, followed by S,P,O,G. G is the graph-option (which is by default a variable). This notably allows for writing RDF graphs represented as `rdf(S,P,O)` using the following code fragment:

```
triple_in(RDF, S,P,O,_G) :-
    member(rdf(S,P,O), RDF).

    ...,
    rdf_save_turtle(Out, [ expand(triple_in(RDF)) ]),
```

Arguments

Out is one of `stream(Stream)`, a stream handle, a file-URL or an atom that denotes a filename.

6 library(semweb/rdf_ntriples): Process files in the RDF N-Triples format

See also http://www.w3.org/TR/n-triples/
To be done Sync with RDF 1.1. specification.

The `library(semweb/rdf_ntriples)` provides a fast reader for the RDF N-Triples and N-Quads format. N-Triples is a simple format, originally used to support the W3C RDF test suites. The current format has been extended and is a subset of the Turtle format (see `library(semweb/turtle)`).

The API of this library is almost identical to `library(semweb/turtle)`. This module provides a plugin into `rdf_load/2`, making this predicate support the format `ntriples` and `nquads`.

read_ntriple(+*Stream, -Triple*) *[det]*
Read the next triple from *Stream* as *Triple*. *Stream* must have a byte-oriented encoding and must contain pure ASCII text.

Arguments

Triple is a term `triple(Subject,Predicate,Object)`. Arguments follow the normal conventions of the RDF libraries. NodeID elements are mapped to `node(Id)`. If end-of-file is reached, *Triple* is unified with `end_of_file`.

Errors `syntax_error(Message)` on syntax errors

read_nquad(+*Stream, -Quad*) *[det]*

Read the next quad from *Stream* as *Quad*. *Stream* must have a byte-oriented encoding and must contain pure ASCII text.

Arguments

Quad	is a term `quad(Subject,Predicate,Object,Graph)`. Arguments follow the normal conventions of the RDF libraries. NodeID elements are mapped to `node(Id)`. If end-of-file is reached, *Quad* is unified with `end_of_file`.

Errors `syntax_error(Message)` on syntax errors

read_ntuple(+*Stream, -Tuple*) *[det]*

Read the next triple or quad from *Stream* as *Tuple*. *Tuple* is one of the terms below. See `read_ntriple/2` and `read_nquad/2` for details.

- `triple(Subject,Predicate,Object)`
- `quad(Subject,Predicate,Object,Graph).`

rdf_read_ntriples(+*Input, -Triples, +Options*) *[det]*

rdf_read_nquads(+*Input, -Quads, +Options*) *[det]*

True when *Triples*/*Quads* is a list of triples/quads from *Input*. *Options*:

anon_prefix(+*AtomOrNode*)
Prefix nodeIDs with this atom. If *AtomOrNode* is the term `node(_)`, bnodes are returned as `node(Id)`.

base_uri(+*Atom*)
Defines the default anon_prefix as _:<baseuri>_

on_error(*Action*)
One of `warning` (default) or `error`

error_count(-*Count*)
If `on_error` is `warning`, unify *Count* with th number of errors.

graph(+*Graph*)
For `rdf_read_nquads/3`, this defines the graph associated to *triples* loaded from the input. For `rdf_read_ntriples/3` this opion is ignored.

Arguments

Triples	is a list of `rdf(Subject, Predicate, Object)`
Quads	is a list of `rdf(Subject, Predicate, Object, Graph)`

rdf_process_ntriples(+*Input, :CallBack, +Options*)

Call-back interface, compatible with the other triple readers. In addition to the options from `rdf_read_ntriples/3`, this processes the option `graph(Graph)`.

Arguments

CallBack	is called as `call(CallBack, Triples, Graph)`, where Triples is a list holding a single `rdf(S,P,O)` triple. Graph is passed from the `graph` option and unbound if this option is omitted.

rdf_db:rdf_load_stream(+*Format, +Stream, :Options*) *[semidet,multifile]*
Plugin rule that supports loading the `ntriples` and `nquads` formats.

rdf_db:rdf_file_type(+*Extension, -Format*) *[multifile]*
Bind the ntriples reader to files with the extensions `nt`, `ntriples` and `nquads`.

7 library(semweb/rdfa): Extract RDF from an HTML or XML DOM

See also
- http://www.w3.org/TR/2013/REC-rdfa-core-20130822/
- http://www.w3.org/TR/html-rdfa/

This module implements extraction of RDFa triples from parsed XML or HTML documents. It has two interfaces: `read_rdfa/3` to read triples from some input (stream, file, URL) and `xml_rdfa/3` to extract triples from an HTML or XML document that is already parsed with `load_html/3` or `load_xml/3`.

read_rdfa(+*Input, -Triples, +Options*) *[det]*
True when *Triples* is a list of `rdf(S,P,O)` triples extracted from *Input*. *Input* is either a stream, a file name, a URL referencing a file name or a URL that is valid for `http_open/3`. *Options* are passed to `open/4`, `http_open/3` and `xml_rdfa/3`. If no base is provided in *Options*, a base is deduced from *Input*.

xml_rdfa(+*DOM, -RDF, +Options*)
True when *RDF* is a list of `rdf(S,P,O)` terms extracted from *DOM* according to the RDFa specification. *Options* processed:

base(+*BaseURI*)
URI to use for ''. Normally set to the document URI.

anon_prefix(+*AnnonPrefix*)
Prefix for blank nodes.

lang(+*Lang*)
Default for `lang`

vocab(+*Vocab*)
Default for `vocab`

markup(+*Markup*)
Markup language processed (xhtml, xml, ...)

rdf_db:rdf_load_stream(+*Format, +Stream, :Options*) *[multifile]*
Register `library(semweb/rdfa)` as loader for HTML RDFa files.

To be done Which options need to be forwarded to `read_rdfa/3`?

8 library(semweb/rdfs): RDFS related queries

The `semweb/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 `semweb/rdfs.pl` library, providing calls like `rdfs_individual_of`(*?Resource, ?Class*).[5]

8.1 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]`.[6,7]

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.[8].

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*.

8.2 Collections and Containers

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

[5] The SeRQL language is based on querying the deductive closure of the triple set. The SWI-Prolog SeRQL library provides *entailment modules* that take the approach outlined above.

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

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

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

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

9 Managing RDF input files

Complex projects require RDF resources from many locations and typically wish to load these in different combinations. For example loading a small subset of the data for debugging purposes or load a different set of files for experimentation. The library `semweb/rdf_library.pl` manages sets of RDF files spread over different locations, including file and network locations. The original version of this library supported metadata about collections of RDF sources in an RDF file called *Manifest*. The current version supports both the [VoID]http://www.w3.org/TR/void/ format and the original format. VoID files (typically named `void.ttl`) can use elements from the RDF Manifest vocabulary to support features that are not supported by VoID.

9.1 The Manifest file

A manifest file is an RDF file, often in [Turtle]http://www.w3.org/TeamSubmission/turtle/ format, that provides meta-data about RDF resources. Often, a manifest will describe RDF files in the current directory, but it can also describe RDF resources at arbitrary URL locations. The RDF schema for RDF library meta-data can be found in `rdf_library.ttl`. The namespace for the RDF library format is defined as `http://www.swi-prolog.org/rdf/library/` and abbreviated as `lib`.

The schema defines three root classes: lib:Namespace, lib:Ontology and lib:Virtual, which we describe below.

lib:Ontology
This is a subclass of owl:Ontology. It has two subclasses, lib:Schema and lib:Instances. These three classes are currently processed equally. The following properties are recognised on lib:Ontology:

dc:title
Title of the ontology. Displayed by `rdf_list_library/0`.

owl:versionInfo
Version of the ontology. Displayed by `rdf_list_library/0`.

owl:imports
: Ontologies imported. If `rdf_load_library/2` is used to load this ontology, the ontologies referenced here are loaded as well. There are two subProperties: lib:schema and lib:instances with the obvious meaning.

lib:source
: Defines the named graph into which the resource is loaded. If this ends in a `/`, the basename of each loaded file is appended to the given source. Defaults to the URL the RDF is loaded from.

lib:baseURI
: Defines the base for processing the RDF data. If not provided this defaults to the named graph, which in turn defaults to the URL the RDF is loaded from.

lib:Virtual
: Virtual ontologies do not refer to an RDF resource themselves. They only import other resources. For example the W3C WordNet manifest defines `wn-basic` and `wn-full` as virtual resources. The lib:Virtual resource is used as a second rdf:type:

```
<wn-basic>
        a lib:Ontology ;
        a lib:Virtual ;
        ...
```

lib:CloudNode
: Used by ClioPatria to combine this ontology and all data it imports into a node in the automatically generated datacloud.

lib:Namespace
: Defines a URL to be a namespace. The definition provides the preferred mnemonic and can be referenced in the lib:providesNamespace and lib:usesNamespace properties. The `rdf_load_library/2` predicates registers encountered namespace mnemonics with rdf-db using `rdf_register_ns/2`. Typically namespace declarations use @prefix declarations. E.g.

```
@prefix     lib: <http://www.swi-prolog.org/rdf/library/> .
@prefix    rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

[ a lib:Namespace ;
  lib:mnemonic "rdfs" ;
  lib:namespace rdfs:
] .
```

9.1.1 Support for the VoID and VANN vocabularies

The [VoID]http://www.w3.org/TR/void/ aims at resolving the same problem as the Manifest files described here. In addition, the [VANN]http://vocab.org/vann/ vocabulary provides the information

about preferred namepaces prefixes. The RDF library manager can deal with VoID files. The following relations apply:

- VoID `Dataset` and `Linkset` are similar to `lib:Ontology`, but a VoID resource is always *Virtual*. I.e., the VoID URI itself never refers to an RDF document.
- The `owl:imports` and its lib specializations are replaced by `void:subset` (referring to another VoID dataset) and `void:dataDump` (referring to a concrete document).
- A description of the dataset is given using `dcterms:description` rather than `rdfs:comment`
- The RDF library recognises `lib:source`, `lib:baseURI` and `lib:Cloudnode`, which have no equivalent in VoID.
- The RDF library recognises `vann:preferredNamespacePrefix` and `vann:preferredNamespaceUri` as alternatives to its proprietary way for defining prefixes. The domain of these predicates is unclear. The library recognises them regardless of the domain. Note that the range of `vann:preferredNamespaceUri` is a *literal*. A disadvantage of that is that the Turtle prefix declaration cannot be reused.

Currently, the RDF metadata is *not* stored in the RDF database. It is processed by low-level primitives that do *not* perform RDFS reasoning. In particular, this means that rdfs:supPropertyOf and rdfs:subClassOf cannot be used to specialise the RDF meta vocabulary.

9.1.2 Finding manifest files

The initial metadata file(s) are loaded into the system using `rdf_attach_library/1`.

rdf_attach_library(+*FileOrDirectory*)

Load meta-data on RDF repositories from *FileOrDirectory*. If the argument is a directory, this directory is processed recursively and each for each directory, a file named `void.ttl`, `Manifest.ttl` or `Manifest.rdf` is loaded (in this order of preference).

Declared namespaces are added to the rdf-db namespace list. Encountered ontologies are added to a private database of `rdf_list_library.pl`. Each ontology is given an *identifier*, derived from the basename of the URL without the extension. This, using the declaration below, the identifier of the declared ontology is `wn-basic`.

```
<wn-basic>
        a void:Dataset ;
        dcterms:title "Basic WordNet" ;
        ...
```

rdf_list_library

List the available resources in the library. Currently only lists resources that have a dcterms:title property. See section 9.2 for an example.

It is possible for the initial set of manifests to refer to RDF files that are not covered by a manifest. If such a reference is encountered while loading or listing a library, the library manager will look for a manifest file in the directory holding the referenced RDF file and load this manifest. If a manifest is found that covers the referenced file, the directives found in the manifest will be followed. Otherwise the RDF resource is simply loaded using the current defaults.

Further exploration of the library is achieved using `rdf_list_library/1` or `rdf_list_library/2`:

rdf_list_library(+*Id*)
Same as `rdf_list_library`(*Id, []*).

rdf_list_library(+*Id, +Options*)
Lists the resources that will be loaded if *Id* is handed to `rdf_load_library/2`. See `rdf_attach_library/1` for how ontology identifiers are generated. In addition it checks the existence of each resource to help debugging library dependencies. Before doing its work, `rdf_list_library/2` reloads manifests that have changed since they were loaded the last time. For HTTP resources it uses the HEAD method to verify existence and last modification time of resources.

rdf_load_library(+*Id, +Options*)
Load the given library. First `rdf_load_library/2` will establish what resources need to be loaded and whether all resources exist. Than it will load the resources.

9.2 Usage scenarios

Typically, a project will use a single file using the same format as a manifest file that defines alternative configurations that can be loaded. This file is loaded at program startup using `rdf_attach_library/1`. Users can now list the available libraries using `rdf_list_library/0` and `rdf_list_library/1`:

```
1 ?- rdf_list_library.
ec-core-vocabularies E-Culture core vocabularies
ec-all-vocabularies All E-Culture vocabularies
ec-hacks            Specific hacks
ec-mappings         E-Culture ontology mappings
ec-core-collections E-Culture core collections
ec-all-collections  E-Culture all collections
ec-medium           E-Culture medium sized data (artchive+aria)
ec-all              E-Culture all data
```

Now we can list a specific category using `rdf_list_library/1`. Note this loads two additional manifests referenced by resources encountered in `ec-mappings`. If a resource does not exist is is flagged using `[NOT FOUND]`.

```
2 ?- rdf_list_library('ec-mappings').
% Loaded RDF manifest /home/jan/src/eculture/vocabularies/mappings/Manifest.ttl
% Loaded RDF manifest /home/jan/src/eculture/collections/aul/Manifest.ttl
<file:///home/jan/src/eculture/src/server/ec-mappings>
```

```
. <file:///home/jan/src/eculture/vocabularies/mappings/mappings>
. . <file:///home/jan/src/eculture/vocabularies/mappings/interface>
. . . file:///home/jan/src/eculture/vocabularies/mappings/interface_class_mapping
. . . file:///home/jan/src/eculture/vocabularies/mappings/interface_property_mapp
. . <file:///home/jan/src/eculture/vocabularies/mappings/properties>
. . . file:///home/jan/src/eculture/vocabularies/mappings/ethnographic_property_m
. . . file:///home/jan/src/eculture/vocabularies/mappings/eculture_properties.ttl
. . . file:///home/jan/src/eculture/vocabularies/mappings/eculture_property_semar
. . <file:///home/jan/src/eculture/vocabularies/mappings/situations>
. . . file:///home/jan/src/eculture/vocabularies/mappings/eculture_situations.ttl
. <file:///home/jan/src/eculture/collections/aul/aul>
. . file:///home/jan/src/eculture/collections/aul/aul.rdfs
. . file:///home/jan/src/eculture/collections/aul/aul.rdf
. . file:///home/jan/src/eculture/collections/aul/aul9styles.rdf
. . file:///home/jan/src/eculture/collections/aul/extractedperiods.rdf
. . file:///home/jan/src/eculture/collections/aul/manual-periods.rdf
```

9.2.1 Referencing resources

Resources and manifests are located either on the local filesystem or on a network resource. The initial manifest can also be loaded from a file or a URL. This defines the initial *base URL* of the document. The base URL can be overruled using the Turtle @base directive. Other documents can be referenced relative to this base URL by exploiting Turtle's URI expansion rules. Turtle resources can be specified in three ways, as absolute URLs (e.g. <http://www.example.com/rdf/ontology.rdf¿), as relative URL to the base (e.g. <../rdf/ontology.rdf¿) or following a *prefix* (e.g. prefix:ontology).

The prefix notation is powerful as we can define multiple of them and define resources relative to them. Unfortunately, prefixes can only be defined as absolute URLs or URLs relative to the base URL. Notably, they cannot be defined relative to other prefixes. In addition, a prefix can only be followed by a Qname, which excludes . and /.

Easily relocatable manifests must define all resources relative to the base URL. Relocation is automatic if the manifest remains in the same hierarchy as the resources it references. If the manifest is copied elsewhere (i.e. for creating a local version) it can use @base to refer to the resource hierarchy. We can point to directories holding manifest files using @prefix declarations. There, we can reference *Virtual* resources using prefix:name. Here is an example, were we first give some line from the initial manifest followed by the definition of the virtual RDFS resource.

```
@base <http://gollem.science.uva.nl/e-culture/rdf/> .

@prefix base:           <base_ontologies/> .

<ec-core-vocabularies>
        a lib:Ontology ;
        a lib:Virtual ;
        dc:title "E-Culture core vocabularies" ;
        owl:imports
```

```
                base:rdfs ,
                base:owl ,
                base:dc ,
                base:vra ,
                ...
```

```
<rdfs>
        a lib:Schema ;
        a lib:Virtual ;
        rdfs:comment "RDF Schema" ;
        lib:source rdfs: ;
        lib:schema <rdfs.rdfs> .
```

9.3 Putting it all together

In this section we provide skeleton code for filling the RDF database from a password protected HTTP repository. The first line loads the application. Next we include modules that enable us to manage the RDF library, RDF database caching and HTTP connections. Then we setup the HTTP authentication, enable caching of processed RDF files and load the initial manifest. Finally `load_data/0` loads all our RDF data.

```
:- use_module(server).

:- use_module(library(http/http_open)).
:- use_module(library(semweb/rdf_library)).
:- use_module(library(semweb/rdf_cache)).

:- http_set_authorization('http://www.example.org/rdf',
                          basic(john, secret)).

:- rdf_set_cache_options([ global_directory('RDF-Cache'),
                           create_global_directory(true)
                         ]).


:- rdf_attach_library('http://www.example.org/rdf/Manifest.ttl').

%%      load_data
%
%       Load our RDF data

load_data :-
        rdf_load_library('all').
```

9.4 Example: A metadata file for W3C WordNet

The VoID metadata below allows for loading WordNet in the two predefined versions using one of

```
?- rdf_load_library('wn-basic', []).
?- rdf_load_library('wn-full', []).
```

```
@prefix    void: <http://rdfs.org/ns/void#> .
@prefix    vann: <http://purl.org/vocab/vann/> .
@prefix     lib: <http://www.swi-prolog.org/rdf/library/> .
@prefix     owl: <http://www.w3.org/2002/07/owl#> .
@prefix     rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix    rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix     xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix      dc: <http://purl.org/dc/terms/> .
@prefix   wn20s: <http://www.w3.org/2006/03/wn/wn20/schema/> .
@prefix   wn20i: <http://www.w3.org/2006/03/wn/wn20/instances/> .

[ vann:preferredNamespacePrefix "wn20i" ;
  vann:preferredNamespaceUri "http://www.w3.org/2006/03/wn/wn20/instances/"
] .

[ vann:preferredNamespacePrefix "wn20s" ;
  vann:preferredNamespaceUri "http://www.w3.org/2006/03/wn/wn20/schema/"
] .

<wn20-common>
        a void:Dataset ;
        dc:description "Common files between full and basic version" ;
        lib:source wn20i: ;
        void:dataDump
                <wordnet-attribute.rdf.gz> ,
                <wordnet-causes.rdf.gz> ,
                <wordnet-classifiedby.rdf.gz> ,
                <wordnet-entailment.rdf.gz> ,
                <wordnet-glossary.rdf.gz> ,
                <wordnet-hyponym.rdf.gz> ,
                <wordnet-membermeronym.rdf.gz> ,
                <wordnet-partmeronym.rdf.gz> ,
                <wordnet-sameverbgroupas.rdf.gz> ,
                <wordnet-similarity.rdf.gz> ,
                <wordnet-synset.rdf.gz> ,
                <wordnet-substancemeronym.rdf.gz> ,
                <wordnet-senselabels.rdf.gz> .

<wn20-skos>
```

```
        a void:Dataset ;
        void:subset <wnskosmap> ;
        void:dataDump <wnSkosInScheme.ttl.gz> .

<wnskosmap>
        a lib:Schema ;
        lib:source wn20s: ;
        void:dataDump
                <wnskosmap.rdfs> .

<wnbasic-schema>
        a void:Dataset ;
        lib:source wn20s: ;
        void:dataDump
                <wnbasic.rdfs> .

<wn20-basic>
        a void:Dataset ;
        a lib:CloudNode ;
        dc:title "Basic WordNet" ;
        dc:description "Light version of W3C WordNet" ;
        owl:versionInfo "2.0" ;
        lib:source wn20i: ;
        void:subset
                <wnbasic-schema> ,
                <wn20-skos> ,
                <wn20-common> .

<wnfull-schema>
        a void:Dataset ;
        lib:source wn20s: ;
        void:dataDump
                <wnfull.rdfs> .

<wn20-full>
        a void:Dataset ;
        a lib:CloudNode ;
        dc:title "Full WordNet" ;
        dc:description "Full version of W3C WordNet" ;
        owl:versionInfo "2.0" ;
        lib:source wn20i: ;
        void:subset
                <wnfull-schema> ,
                <wn20-skos> ,
                <wn20-common> ;
        void:dataDump
                <wordnet-antonym.rdf.gz> ,
```

```
                <wordnet-derivationallyrelated.rdf.gz> ,
                <wordnet-participleof.rdf.gz> ,
                <wordnet-pertainsto.rdf.gz> ,
                <wordnet-seealso.rdf.gz> ,
                <wordnet-wordsensesandwords.rdf.gz> ,
                <wordnet-frame.rdf.gz> .
```

10 library(semweb/sparql_client): SPARQL client library

This module provides a SPARQL client. For example:

```
?- sparql_query('select * where { ?x rdfs:label "Amsterdam" }', Row,
                [ host('dbpedia.org'), path('/sparql/')]).

Row = row('http://www.ontologyportal.org/WordNet#WN30-108949737') ;
false.
```

Or, querying a local server using an `ASK` query:

```
?- sparql_query('ask { owl:Class rdfs:label "Class" }', Row,
                [ host('localhost'), port(3020), path('/sparql/')]).
Row = true.
```

sparql_query(+*Query, -Result, +Options*) *[nondet]*

Execute a SPARQL query on an HTTP SPARQL endpoint. *Query* is an atom that denotes the query. *Result* is unified to a term `rdf(S,P,O)` for `CONSTRUCT` and `DESCRIBE` queries, `row(...)` for `SELECT` queries and `true` or `false` for `ASK` queries. *Options* are

host(+*Host*)

port(+*Port*)

path(+*Path*)
The above three options set the location of the server.

search(+*ListOfParams*)
Provide additional query parameters, such as the graph.

variable_names(-*ListOfNames*)
Unifies *ListOfNames* with a list of atoms that describe the names of the variables in a `SELECT` query.

Remaining options are passed to `http_open/3`. The defaults for Host, Port and Path can be set using `sparql_set_server/1`. The initial default for port is 80 and path is `/sparql/`.

For example, the ClioPatria server understands the parameter `entailment`. The code below queries for all triples using _rdfs_entailment.

```
?- sparql_query('select * where { ?s ?p ?o }',
                Row,
                [ search([entailment=rdfs])
                ]).
```

sparql_set_server(+*OptionOrList*)
Set sparql server default options. Provided defaults are: host, port and repository. For example:

```
    sparql_set_server([ host(localhost),
                        port(8080)
                        path(world)
                      ])
```

The default for port is 80 and path is `/sparql/`.

sparql_read_xml_result(+*Input, -Result*)
Specs from http://www.w3.org/TR/rdf-sparql-XMLres/. The returned *Result* term is of the format:

select(*VarNames, Rows*)
Where *VarNames* is a term `v(Name, ...)` and *Rows* is a list of `row(....)` containing the column values in the same order as the variable names.

ask(*Bool*)
Where *Bool* is either `true` or `false`

sparql_read_json_result(+*Input, -Result*) *[det]*
The returned *Result* term is of the format:

select(*VarNames, Rows*)
Where *VarNames* is a term `v(Name, ...)` and *Rows* is a list of `row(....)` containing the column values in the same order as the variable names.

ask(*Bool*)
Where *Bool* is either `true` or `false`

See also http://www.w3.org/TR/rdf-sparql-json-res/

11 library(semweb/rdf_compare): Compare RDF graphs

This library provides predicates that compare RDF graphs. The current version only provides one predicate: `rdf_equal_graphs/3` verifies that two graphs are identical after proper labeling of the blank nodes.

Future versions of this library may contain more advanced operations, such as diffing two graphs.

rdf_equal_graphs(+*GraphA, +GraphB, -Substition*) *[semidet]*

True if *GraphA* and *GraphB* are the same under *Substition*. *Substition* is a list of BNodeA = BNodeB, where BNodeA is a blank node that appears in *GraphA* and BNodeB is a blank node that appears in *GraphB*.

Arguments	
GraphA	is a list of `rdf(S,P,O)` terms
GraphB	is a list of `rdf(S,P,O)` terms
Substition	is a list if NodeA = NodeB terms.

To be done The current implementation is rather naive. After dealing with the subgraphs that contain no bnodes, it performs a fully non-deterministic substitution.

12 library(semweb/rdf_portray): Portray RDF resources

To be done

- Define alternate predicate to use for providing a comment
- Use rdf:type if there is no meaningful label?
- Smarter guess whether or not the local identifier might be meaningful to the user without a comment. I.e. does it look 'word-like'?

This module defines rules for `user:portray/1` to help tracing and debugging RDF resources by printing them in a more concise representation and optionally adding comment from the label field to help the user interpreting the URL. The main predicates are:

- `rdf_portray_as/1` defines the overall style
- `rdf_portray_lang/1` selects languages for extracting label comments

rdf_portray_as(+*Style*) *[det]*

Set the style used to portray resources. *Style* is one of:

`prefix:id` Write as NS:ID, compatible with what can be handed to the rdf predicates. This is the default.

`writeq` Use quoted write of the full resource.

`prefix:label` Write namespace followed by the label. This format cannot be handed to `rdf/3` and friends, but can be useful if resource-names are meaningless identifiers.

`prefix:id=label` This combines prefix:id with prefix:label, providing both human readable output and output that can be pasted into the commandline.

rdf_portray_lang(+*Lang*) *[det]*

If *Lang* is a list, set the list or preferred languages. If it is a single atom, push this language as the most preferred language.

13 Related packages

The core infrastructure for storing and querying RDF is provided by this package, which is distributed as a core package with SWI-Prolog. [ClioPatria]http://cliopatria.swi-prolog.org provides a comprehensive server infrastructure on top of the *semweb* and *http* packages. ClioPatria provides a SPARQL 1.1 endpoint, linked open data (LOD) support, user management, a web interface and an extension infrastructure for programming (semantic) web applications.

[Thea]http://www.semanticweb.gr/TheaOWLLib/ provides access to OWL ontologies at the level of the abstract syntax. Can interact with external DL reasoner using DIG.

14 Version 3 release notes

RDF-DB version 3 is a major redesign of the SWI-Prolog RDF infrastructure. Nevertheles, version 3 is almost perfectly upward compatible with version 2. Below are some issues to take into consideration when upgrading.

Version 2 did not allow for modifications while read operations were in progress, for example due to an open choice point. As a consequence, operations that both queried and modified the database had to be wrapped in a transaction or the modifications had to be buffered as Prolog data structures. In both cases, the RDF store was not modified during the query phase. In version 3, modifications are allowed while read operations are in progress and follow the Prolog **logical update view** semantics. This is different from using a transaction in version 2, where the view for all read operations was frozen at the start of the transaction. In version 3, every read operation sees the store frozen at the moment that the operation was started.

We illustrate the difference by writing a forwards entailment rule that adds a *sibling* relation. In **version 2**, we could perform this operation using one of the following:

```
add_siblings_1 :-
        findall(S-O,
                ( rdf(S, f:parent, P),
                  rdf(O, f:parent, P),
                  S \== O
                ),
                Pairs),
        forall(member(S-O, Pairs), rdf_assert(S,f:sibling,O)).

add_siblings_2 :-
        rdf_transaction(
            forall(( rdf(S, f:parent, P),
                     rdf(O, f:parent, P),
                     S \== O
                   ),
                   rdf_assert(S, f:sibling, O))).
```

In **version 3**, we can write this in the natural Prolog style below. In itself, this may not seem a big advantage because wrapping such operations in a transaction is often a good style anyway. The story changes with more complicated constrol structures that combine iterations with steps that depend on triples asserted in previous steps. Such scenarios can be programmed naturally in the current version.

```
add_siblings_3 :-
        forall(( rdf(S, f:parent, P),
                 rdf(O, f:parent, P),
                 S \== O
               ),
               rdf_assert(S, f:sibling, O)).
```

In version 3, code that combines queries with modification has the same semantics whether executed inside or outside a transaction. This property makes reusing such predicates predictable.

rdf_statistics/2 Various statistics have been renamed or changed:

- `sources` is renamed into `graphs`
- `triples_by_file` is renamed into `triples_by_graph`
- `gc` has additional arguments
- `core` is removed.

rdf_generation/1 Generations inside a transaction are represented as *BaseGeneration+TransactionGeneration*, where *BaseGeneration* is the global generation where the transaction started and *TransactionGeneration* expresses the generation within the transaction. Counting generation has changed as well. In particular, comitting a transaction steps the generation only by one.

rdf_current_ns/1, `rdf_register_ns/2`**,** `rdf_register_ns/3` These predicates are renamed into `rdf_current_prefix/1`, `rdf_register_prefix/2`, `rdf_register_prefix/3`. The old predicates are still available as deprecated predicates.

rdf_unload/1 now only accepts a source location and deletes the associated graph using `rdf_unload_graph/1`.

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, the FP-6 project HOPS of the European Commission, the COMBINE project supported by the ONR Global NICOP grant N62909-11-1-7060 and the Dutch national program COMMIT.

Index