

SWAD–Europe Deliverable 7.1: RDF API requirements and comparison

Project name:

W3C Semantic Web Advanced Development for Europe (SWAD-Europe)

Project Number:

IST-2001-34732

Workpackage name:

7. Databases, Query, API, Interfaces

Workpackage description:

<http://www.w3.org/2001/sw/Europe/plan/workpackages/live/esw-wp-7.html>

Deliverable title:

RDF APIs: requirements and current practice

URI:

http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/

Author:

Jan Grant

Abstract:

This area of work researches the storage API requirements for RDF applications. Such APIs are required to access Semantic Web data stored within various kinds of RDF-aware system. They can be very low-level or more like query languages in their expressivity. Existing experience shows that different sorts of RDF datastore will have different levels of API, and some datastores may implement several layers. Work here connects closely with the W3C RDF Core working group decisions and constraints on the RDF model; some deployed RDF APIs may not yet precisely reflect the clarifications of RDF produced by the RDF Core Working Group. This report looks at RDF features as expressed by the existing round of Working Drafts produced by the RDF Core Working Group and should provide a framework for the comparison of APIs and their associated semantics. "Sticking points" and potential problem areas are highlighted. The intention is to provide a reasonable basis for the evaluation of existing and future RDF APIs.

STATUS:

Started 2003-01-14 (Jan); This version 2003-02-27 (Jan).

Contents

- 1 [Introduction](#)
 - 2 [RDF Features and API Considerations](#)
 - 3 [Existing Work](#)
 - A [References](#)
-

1 Introduction

While the adoption of a particular RDF API will, of course, be influenced heavily by platform and language considerations, there is a common core set of considerations that apply to all APIs. We begin therefore by considering general features that those APIs may expose and support. We look primarily at the features of RDF and RDF "triple-matching" implementations in imperative languages. After the general discussion of the issues and potential points of variability is concluded, existing production-quality APIs are examined in the light of these features. This survey is not exhaustive; instead, we seek to highlight features that may represent "corner-cases" or areas of complexity in API specification, and to provide sufficient examples of API examination that a reader could undertake their own survey.

The RDF Specifications [\[RDF-WD\]](#) define a number of concepts and constructs which current APIs may offer differing degrees of support for. As well as these features, the discussion below includes *extra-RDF* features which have proved to be useful in practice (that is, during the development of experimental RDF applications). Where such features are not found in the RDF Specifications an indication is given.

In addition, an API provides services to an environment for programming: we also consider some environmental features that may be worth consideration when the choice of API is made.

Where textual examples are given, we use the compact version of the N-Triple notation used in [\[RDF-PRIMER\]](#).

2 RDF Features and API Considerations

The following concepts are likely to be found (in one guise or another) in common imperative RDF APIs.

2.1 The RDF Graph - As one might expect, the RDF Graph (that is, the set of triples that makes up the graph) is

likely to appear in an API. Without such a facility being explicitly present, it is hard to write programs that deal with multiple RDF documents or graphs concurrently.

The assumption that a program deals with a single, implicit instance of an RDF graph may be found in some experimental APIs. Such an assumption may give rise to some notational convenience, but that convenience comes at the expense of explicitly being able to distinguish between separate instances of graphs.

In general, the graph corresponds to the [RDF Graph](#) defined in [\[RDF-CONCEPTS\]](#); that is, to the abstract syntax.

2.2 Graph operations - We look now at the basic operations on a graph. These form the basis of what has been characterised as a "triple-matching" API - that is, they permit the basic manipulation and interrogation of a graph on a triple-by-triple basis. Graph manipulation is considered first; we consider the interrogation of a graph [later](#).

Essentially, the minimal requirements for an RDF API are that an updatable graph should support the assertion and retraction of triples.

2.3 The results of operations - At this stage it is worthwhile considering briefly the way in which the API reports the results of its operations. Early experimental RDF APIs used a simple "triple-matching" style to select target triples; more will be said of this below. When an operation (such as a selection) returns a set of triples, there are a number of patterns or styles which an API might adopt to present the results: for instance, as a "record set" or snapshot of the results; or as a sequence, generator, or iterator that produces successive results. The choice of return type may be largely influenced by idioms common to the implementation language (and potentially, efficiency considerations in the underlying implementation). There may be benefits from using a style that integrates closely with existing language facilities (eg, the C++ STL or Java's Collection Framework). In the discussion that follows, where illustrative examples are needed, we adopt an iterator-based idiom.

2.4 Provenance (extension) - One extension that may be supported, which has shown some utility in practical RDF applications (eg, [\[FOAF\]](#)), is the notion of *provenance*. Mechanisms for support for this vary; essentially, the term is used to describe the tracking of the source of a particular subset of assertions within a graph. The use of this provides a quick-and-dirty way to model the [merge](#) of a number of source graphs into one resultant graph, while maintaining origin information on each source. The manner of identification of each source graph varies; typically, each source is associated with an [RDF URIref](#).

The mechanism is particularly useful if one source graph changes; it allows the fast retraction of the original statements sourced from that graph followed by a reassertion, for a fast "partial update".

As a nonstandard extension, the precise semantics of provenance may vary between implementations: for instance, after the following sequence of operations, the question arises whether the originally asserted triple still exists within the resultant graph *g*:

1. into *g* assert the triple (**eg:n1 eg:p eg:n2 .**) with provenance *g1*
2. into *g* assert the triple (**eg:n1 eg:p eg:n2 .**) with provenance *g2*
3. from *g* drop all assertions with provenance *g1*

An alternative approach to provenance may be suitable with an object-oriented API: to provide an implementation of *RDF Graph* that provides an amalgamating view over one or more source graphs. There, the semantics of updates could be more clearly expressed. However, there may be disadvantages to this approach too: the creation of many persistent graphs can be expensive.

2.5 The Triple - As defined by [\[RDF-CONCEPTS\]](#), an RDF Triple is composed of three parts: the subject, the predicate, and the object. (Whether the *Triple* is exposed as a first-class citizen via the API or is treated implicitly may be a stylistic issue, depending on features of the language of implementation.)

Using the language of the [\[RDF-CONCEPTS\]](#), the *Subject* may be a *URIref* or a *Blank Node*; the *Predicate* is a *URIref* and the *Object* may be a *URIref*, a *Blank Node*, or a *Literal* (which may be a *Typed Literal*).

We now look briefly at the features of each of these classes, as required by the RDF Specifications.

2.6 URIrefs - The definition of an RDF URIref is given in [\[RDF-CONCEPTS\]](#). Note that there is no requirement that a URIref be dereferencable, and that the RDF Specifications [makes explicit](#) the fact that there is no automatic connection between a URIref's denotation and the document (fragment) it may address. However, there seems to be some *de facto* convention in this regard. In any respect, such a fact only has a simplifying impact upon the requirements for an RDF API.

The namespace used to create a URIref in the RDF/XML serialisation is not considered by the RDF Specifications to play any part in the URIref itself. The behaviour of RDF APIs may not completely reflect this. See [below](#).

2.7 Literals - Again, a full definition of RDF Literals can be found in [\[RDF-CONCEPTS\]](#). Rather than repeat that, we note here that literals may carry an optional language tag. Support for interrogation of literals based upon the language tag may be present; however, current opinion seems to hold that the language tag is a somewhat archaic misfeature preserved solely for the sake of backwards compatibility.

2.8 Typed literals - The revised RDF Specifications have introduced the notion of *Typed Literals*. Taking the view that a basic RDF API offers access to the abstract syntax, a typed literal is simply the pairing of a unicode string (the lexical form of the literal) and a URIref naming the datatype of the literal. [NOTE: at time of writing, a language tag may optionally be part of a typed literal. The future of this is still unclear, and this decision may be revised in a future draft of the specification.]

Therefore, at a minimum, an RDF API simply needs to be able to record this pair. The RDF Specifications place the determination of the *value* of a datatyped literal in the realm of the model theoretic semantics (at the inference layer); thus, there is no onus on an implementation of an abstract syntax to verify that a particular literal corresponds

to a legal lexical form of the named datatype. In particular, an RDF Graph may indeed contain "illformed" typed literals.

However, it is quite possible that an RDF API and its implementation may "go the extra mile" with datatyped literals. At the time of writing, there is little practical experience of using datatyped literals.

2.9 XML literals - One of the literal types mandated by the RDF Specification is *XMLLiteral*. Support for this may be as minimal as the requirement outlined above for general typed literals.

2.10 Blank Nodes - The node type with the most potential for complexity is the *Blank Node*. These nodes have an identity with respect to the graph that contains them, but no universal identifying name. When merging two RDF graphs (at an abstract syntax level), every blank node in the result will have originated with a corresponding blank node in one or other of the source graphs - no node will correspond to a blank node in *both* source graphs.

This semantics presents some complexity. The internal identity of blank nodes must be preserved, even if two blank nodes appear initially to be the same. Consider the following sequence of operations on a graph, *g*, in which we use an illustrative pseudocode:

```
b1 = new Blank Node
b2 = new Blank Node

g.assert( triple ( URIref(eg:a) URIref(eg:p1) b1 ) )
g.assert( triple ( URIref(eg:a) URIref(eg:p1) b2 ) )

g.assert( triple ( b1 URIref(eg:p2) URIref(eg:c) ) )
g.assert( triple ( b2 URIref(eg:p2) URIref(eg:d) ) )
```

After the first pair of assertions, *g* contains two triples which are identical apart from the blank nodes in them. In fact, as far as the semantics is concerned, the interpretation of this graph would be unchanged were we to "merge" the two triple into a single one. However, as can be seen, the separate identities of *b1* and *b2* must be preserved, since the second two assertions add concrete disambiguating triples to the graph. The merging of the two blank nodes would not give rise to a logically equivalent graph.

A second illustration is necessary to indicate that care must be taken when writing code that utilises blank nodes. Consider the following pseudocode, which attempts to copy graph *g1* to graph *g2*:

```
i = iterator over the triples of g1

while ( i has more triples ) {

    t = next triple from i

    g2.assert( t )

}
```

It is possible to produce an implementation which would enable this pseudocode to work as written. Consider the operation on the following graph (expressed in N-Triples):

```
eg:a eg:p _:a .
_:a eg:q eg:b .
```

Clearly, for the pseudocode to duplicate this trivial structure in a second graph, a (temporary) relationship between the identity of the blank node in graph *g1* and the corresponding blank node in graph *g2* must be established. Not every RDF API may support this implicit semantics, however, in which case the trivial "copy a graph" test case must be expressed by hand as follows (here we assume that blank nodes from the same graph may be compared for identity):

```
i = iterator over the triples of g1
m = new map < blank nodes of g1, blank nodes of g2 >

while ( i has more triples ) {

    t = next triple from i

    s = t.subject
    if ( s is a blank node ) {

        if ( s exists as a key in m ) {

            s' = m(s)
```

```

    } else {

        s' = new blank node in g2
        m(s) = s'

    }

}

o = t.object
... similar code for o follows ...

g2.assert( triple( s' p o' ) )

}

```

We note that some trivial tactics, such as assigning a GUID for blank nodes, may suffice in practice. If the GUID name assigned to a "blank node" by an implementation is truly unique, then logically the resulting graph entails all the same consequences of the graph with the "really" blank node. This tactic, however, does admit of the possibility that the names assigned to blank nodes will not be unique, with corresponding subtle errors or renaming checks required in the event of a name clash.

The treatment of blank nodes is a difficult area, and one in which there is a high degree of variability amongst APIs. Indeed, the behaviour of some APIs may vary depending on the precise implementation of backing store selected. If particular behaviour is required, testing for the exact semantics may be required.

2.11 Graph-level operations - Of course, an API may choose to mitigate some of the impact of this complication by providing convenience operations that work over set of triples en bloc. One obvious operation is the *merging* of one graph into another.

There are also operations that make sense only at a whole-graph level. [\[RDF-CONCEPTS\]](#) defines a syntactic equality between graphs: essentially, an isomorphism relation between instances of the abstract syntax. This high-level operation is expensive; at least one RDF API provides it (with a good implementation) as a convenience. (See Jena, below.)

2.12 RDF support - Thus far, we have only considered simple operations against an abstract syntax consisting of triples. RDF and RDFS both provide additional vocabulary elements and idiomatic constructs. An RDF API may provide support for these. RDF itself has three such features: the collection elements, containers, and reification.

2.12.1 RDF Containers

The container mechanism is from the original RDF specification. It uses indexical properties (**rdf:_1**, **rdf:_2**, ...) to link a container (of type **rdf:Bag**, **rdf:Alt** or **rdf:Seq**) to its members.

RDF itself makes no guarantees for the wellformedness of containers. However, an API may supply convenience methods, for example:

- add a *node* to a container;
- test for the membership of *node* in a given container;
- enumerate the members of a container.

All of these are possible to build from basic triple-matching operations; however, the API may supply such methods itself. "Support" for this construct may give rise to a number of corner cases, for example: do the methods operate on non-well-formed collections? If a "graph merge" operation is performed, will it renumber clashing collections? While there is no definitive "right" behaviour, answers to these questions may vary between implementations; APIs may give differing guarantees in this regard.

2.12.2 RDF Collections

The collection mechanism is a new feature added to RDF, modelled on *cons lists* as specified by DAML+OIL. It has an added bonus under the open world assumption of many RDF systems, in that it is possible to explicitly close a list (that is, assert that it has no more members) through the use of **rdf:nil**. Supporting operations, if present, will be similar to those for containers. There are analogous issues with non-well-formed collections.

2.12.3 RDF Reification

Reification is a de re mechanism for talking about "statings" - that is, occurrences of RDF triples, within RDF itself. Typical operations in support of reification are:

- assert the reification of the triple, *t*, as stating *s*;
- find all the reifications of a triple, *t*

Again, these are purely convenience methods which may be expressed in terms of more fundamental operations.

2.13 RDFS support - RDFS adds additional vocabulary to RDF in support of a simple class system. From an abstract syntax point of view, there is little additional support required, but the following operations may be provided in the absence of full inferencing support.

2.13.1 RDFS Classes

RDFS classes are related to each other via **rdfs:subClassOf** relationships. Such relationships may be circular. Typical simple support for classes might extend to providing convenience mechanisms as follows:

- is class *A* a subclass (directly, or indirectly) of class *B*?
- enumerate all the subclasses (or superclasses) of a given class.

Due to interactions with the subproperty relationship (below), these methods may not be as straightforward as first appears. Supporting these operations may require fuller support of RDFS inferencing.

2.13.2 RDFS Properties

In addition to the analogous subproperty chains, RDFS properties may also have range and domain specifications. The ability to make such declarations leads naturally to ask whether an RDF API can support validation.

2.14 Instance validation - Although RDF permits schema and instance data to be mixed freely, it is quite common in practice to see a separation between RDF graphs used to specify a schema, and RDF graphs used to specify "instance data".

That being the case, a natural question that arises is whether instance data can be validated against a schema. Although such a question makes something of a closed-world assumption, it is asked often enough in practice that an API may support such a decision-making operation:

- given a "schema specification" in a graph *gS* and an instance graph, *gi*, validate the instance graph against the provided schema

Such an operation may include checking that collections and containers are well-formed, confirming that range and domain constraints apply, and validating the lexical forms of datatyped literals (where possible). While RDFS is somewhat weak as a schema language, this data-validation operation may nevertheless find frequent practical use, particularly by users new to the open world of the Semantic Web, who come from a more traditional data-processing background.

2.15 I/O - Of course, while graph manipulation is useful, for applications to take their place in a large Semantic Web they must be able to communicate. For this purpose, the ability to transform a graph to and from a serialised form is necessary. Currently there is one standard form for RDF, RDF/XML [\[RDF-SYNTAX\]](#), although informal support for [N-Triples](#) [\[RDF-TESTCASES\]](#) may also be provided. Additionally there is some current interest in embedding RDF within other XML formats (particularly XHTML), although standardisation effort in this regard is still ongoing.

The minimal requirement for a parser is to be able to produce an RDF graph from an RDF/XML serialisation. Exact details vary; for instance, many parsers are stream-oriented, producing a sequence of triples as they consume RDF. Most parsers build on top of existing XML SAX-based parsers (for example, Expat).

While namespaces do not appear in the RDF abstract syntax, it is conceivable that an RDF API offer programmatic access and control over the namespaces used in a document, for example, to select the namespaces used for graph serialisation.

Although the RDF Syntax specification does not deal with embedding RDF within larger XML documents, it is certainly possible that a parser may offer facilities to extract RDF "opportunistically" from the whole XML document, or to operate over a subsequence of a stream of SAX events.

2.16 External considerations with RDF API semantics - 2.16.1 Concurrent and Overlapping Operations

Where operations produce and consume sequences of triples, it is natural to ask whether such operations can be safely overlapped, and what the semantics of such an overlap may be. The detailed semantics may actually vary not only between RDF APIs, but within one API that offers multiple implementations of a graph store.

There are two extremes that an API's semantics might range between: that each operation acts on a "snapshot" of the underlying RDF graph; and that each operation provides a synchronised "view" of the underlying store. The API may either provide a default semantics or permit (via different methods, or different wrapper classes) the selection of the precise semantics.

In addition, the selection of an appropriate API might hinge on the safety of the underlying data. Therefore, it is appropriate to ask whether the API offers some support for compound operations being combined into transactions.

A related issue is the thread safety of an API, and the possibility of using multiple processes to access the same underlying graph store. (No de facto standard has yet arisen for the support of these facilities; an API's implementation may provide greater or lesser degrees of support for communicating these requirements to underlying layers through its SPI.)

2.16.2 Integration with Standard Language Features

As has been briefly mentioned, the evaluation of an API for suitability may wish to include a consideration of how well the API integrates with standard features of the language and platform. For example, if a triple-matching

operation returns its results using an iterator pattern, and the language provides a standard framework for iterators, it would be reasonable for the returned iterator to be compatible with that framework.

We do not dwell too strongly on this aspect of an API, however, since the consideration is largely stylistic. Providing the API fits broadly into well-established patterns for containers, sequences, and iterators common to modern imperative languages, targetted wrappers are usually trivial to implement.

2.16.3 Selection of Graph Implementation

Of course, an API may supply several implementations of a graph store, with varying semantics: memory-based and transient, database-backed, and so on. In the survey below we highlight briefly the mechanism used to select a graph implementation (where present).

2.17 Query Support -

2.18 Triple Matching - The simplest mechanism for querying the RDF abstract syntax arose early in experimental APIs, and has been dubbed "triple-matching". Essentially, a triple template is presented: each of the subject, predicate and object are either specified with constant values or a wildcard placeholder (meaning "match anything") is supplied. The operation returns a set or sequence of triples which match the specified constraint.

The common operations which one would expect to see of this ilk include:

- test for the presence of matching triples;
- return the matching triples;
- drop the matching triples;
- drop the matching triples, returning those deleted; and so on.

It is possible to extend the triple-matching to a more general notion where a filtering function is passed, which accepts or rejects triples based upon programmatically-supplied criteria.

As an implementation detail, it is helpful if the API can guarantee an ordering on triples (at least, those originating in the same graph). Again, this may be dependent on the precise graph store used; but such a facility makes the merging of triple sequences much more efficient (although this effectively requires a low-level ordering relationship be supplied between triples).

2.19 RDF Path - [RDF Path](#) is an informal working group attempting to produce a language for node selection roughly analogous to XPath. While no formal proposal exists, the notion is an interesting one since it offers the promise of a higher-level mechanism for selecting sets of nodes at once, through what is essentially a variant on a regular expression describing a set of paths through an RDF graph. In particular, such mechanisms offer simple ways to calculate the transitive closure of a property, etc.

Support for such constructs is somewhat limited at present. Work in this area may prove to be useful; it certainly offers a more high-level approach than the simplistic triple-matching.

2.20 RDF Query - The highest-level abstractions for querying the RDF abstract syntax have all tended to revolve around RDF Query languages, which are (more or less) reminiscent of SQL's DQL subset. This topic is covered in more detail in [7.2](#); suffice to say that a typical RDF Query API takes a graph and a query specification, and returns a set or sequence of bindings for the free variables in the query which satisfy the constraints expressed by it.

Of course, a query need not merely be expressed over the abstract syntax; it may be expanded to include the *closure* of a graph under particular *closure rules* (see [\[RDFSEMANTICS\]](#)). In this case, the query system represents an interface to an implementation's inferencing capabilities.

2.21 Inference - Not all current implementations provide support for inferencing, instead providing RDF graph operations solely at the level of the abstract syntax. These implementations can still be extremely useful for many practical applications involving the manipulation of metadata.

Experimental systems for RDF/RDFS inferencing thus far are varied in their approaches; the imperative triple-matching idiom is far from common in this context. However, inference support may be offered via a triple-matching API that exposes an inferencing "layer" built on top of a graph store. It is not clear yet if this is the best or most natural approach to the problem of providing a programmatic interface to RDF(S) inference.

However, it *is* possible that such an approach may be taken. For example, a triple-matching interface may be offered over the [closure](#) of an RDF graph. The RDF, RDFS and RDF datatyping closure rules are not amenable to naive forward-chaining (that is, just adding triples to a model until no more closure rules can be satisfied) although at least one system [\[CWM\]](#) has tried this approach. However, the full RDF closure rules introduce a countably infinite set of additional triples. Therefore, the use of a combination of forward- and backward-chaining in an inference layer is likely to be more appropriate.

Inference support does not have to stop with mere closure rules. A higher-level - that is, graph-level - operation is to determine if one RDF graph entails another, and if so, what values must be bound to do so. This actually gives rise to another way of expressing some RDF Query constraints.

2.22 Simple Inference Operations - APIs do not have to offer full support for RDF(S) inferencing to be useful. In fact, two simple facilities have proven useful that are not expressible using RDF or RDFS.

The first facility is a simple one to assert that two nodes in an RDF graph have the same denotation; that is, that they refer to the same thing. Note that the simple "merging" of the two nodes in the graph is not as straightforward as one might expect; instead, it might be better to properly record that two nodes have the same denotation. The effect on the operation of an API to interrogations on the existence of triples would then be as follows: if **eg:X** and **eg:Y** are declared to have the same denotation, then a query such as

- does the triple (**eg:f eg:p eg:X .**) exist in the graph?

may be answered in the positive if (**eg:f eg:p eg:Y .**) is present in the graph.

A second operation that has proven of great utility for simple inferencing applications (eg, [FOAF](#)) is the ability to declare RDF properties to functional and/or "inverse functional", that is:

- for a property **eg:p** and some nodes *N1* and *N2* (which denote the same thing), if (**X eg:p N1 .**) and (**Y eg:p N2 .**) then X and Y have the same denotation; alternatively,
- for a property **eg:p** and some nodes *N1* and *N2* (which denote the same thing), if (**N1 eg:p X .**) and (**N2 eg:p Y .**) then X and Y have the same denotation.

2.22 Beyond RDF and RDFS - More expressive languages for ontology description are currently being defined: the OWL family of languages [WEBONT](#). In the future we can expect to see the adoption of more OWL functionality into RDF systems, and exposed via RDF/OWL-capable APIs.

3 Existing Work

This is not intended to be an exhaustive survey; instead, we look at the APIs of a few existing RDF implementations in the light of the features discussed above.

3.1 The Redland RDF Application Framework - We look first at the Redland [REDLAND](#) toolkit.

3.1.1 Language and Platform

The core Redland implementation is written in a fairly portable C style. It has been ported to a number of Unix and Unix-like platforms as well as the Windows platform. The API uses explicit calls to initialise and finalise its subsystems, rather than relying on linker technology and `.init` (etc.) sections. The implementation has a reputation for efficiency; here, however, we concentrate on the characteristics of the API itself.

It has been widely used within applications which utilise it for the storage and manipulation of metadata.

Redland is implemented in C; however, wrappers for many languages have been produced using the SWIG [SWIG](#) system: for a full list, consult the Redland documentation. The API is object-oriented in flavour (implemented using a few conventions on top of C). Typically for C, it uses explicit object allocation, ownership and destruction (this may be automatic in languages that support automatic object deallocation where SWIG supports it). The API is well-documented with care taken to highlight where functions take object ownership. Where plug-ins may be provided to implement particular features of the API, a consistent factory-registration system is used.

3.1.2 Basic types

The API supports nodes, which may be URIs, literals (with language tagging and support for XML literals) and blank nodes. Ordinal properties are considered a separate type. Typed literal constructors and inspectors are provided.

Blank nodes carry string identifiers; these may be manipulated explicitly via API calls.

3.1.3 Graph operations and Results

Redland offers a full set of wildcarded triple-matching query operations, together with statement assertion and retraction. Returned sets of statements are implemented via an iterator-style interface. Triples (called *Statements*) are first-class citizens in this API.

3.1.4 Extra-RDF operations

A model may contain statements associated with a *context*; this provides a provenance-tracking style of interface.

3.1.5 Whole-graph operations

Redland offers copy constructors for graphs; however, no computationally expensive syntactic equality is offered.

3.1.6 RDF construct support

Redland offers convenience constructors for ordinal properties.

3.1.7 RDFS construct support

While there are implementation plans reported to add fast checking of **rdf:type**, Redland operates solely on the abstract syntax; no special support for RDFS is provided.

3.1.8 Parser

Redland itself provides no parser; however, there are hooks for supplying a "plug-in" parser; a number of parsers are supported. The parsing and serialisation interfaces are built around a "stream" of statements. This is similar in concept to an iterator or generator; it additionally permits the attachment of a filter function for fine-grained statement selection.

3.1.9 Serialiser

Similarly, serialisation mechanisms can be provided via a plug-in interface. Current plug-ins provide for the serialisation and parsing of RDF/XML and N-Triples. Fine-grained control over serialisation behaviour is available generically through the association of URIs with serialisation features.

3.1.10 Overlapping operations

No explicit transaction-control operations are present in Redland; the API does not define a guaranteed semantics for overlapping operations.

3.1.11 Storage implementation and selection

A number of storage systems are available in the existing implementation. They offer in-memory and persistent storage via Berkeley DB [\[BDB\]](#).

3.1.12 Query

Triple-matching is supported using templated statements with total wildcards. Matched results are returned as iterators. An iterator may be converted into a stream and additional fine-grained filters applied to it for further selection.

The current implementation contains no RDF Query *implementation*, however the API supports queries expressed either as strings or as opaque query objects (eg, for pre-parsed queries).

3.1.13 Inference

Redland is an RDF API that operates solely on the abstract syntax. In its current state, an inferencing layer *could* be provided by an "RDF Model Storage" implementation; however, the API is strongly slanted towards operation on the abstract syntax.

3.2 The Jena Toolkit - Jena is a popular open-source framework for RDF processing. At the time of writing, work on the a full revision to produce a "Jena 2" is still underway; we examine instead the Jena 1 API.

3.2.1 Language and Platform

The Jena API is implemented in Java, and should therefore be reasonably platform-neutral. One of the supplied model storage implementations uses Berkeley DB.

3.2.2 Basic types

Jena provides a simple hierarchy of types based on *RDFNode: Resource, Literal*, and so on. In addition, there are some convenience types (*Alt, Bag, and Seq* plus the DAML support) which directly support the manipulation of those RDF constructs.

Blank nodes get "anonymous IDs" assigned automatically; of these, Jena says:

The id is unique within the scope of a particular implementation. All models within an implementation will use the same id for the same anonymous resource.

3.2.3 Graph operations and Results

Jena's graph operations take a resource-centric approach (that is, a resource carries its associated graph). This permits an interesting idiom which works well in Java.

Jena has its own iterator classes which require an explicit disposal. This is principally due to the need to ensure proper resource management within storage implementations: Java's *finalization* mechanism does not make sufficient guarantees to be usable for this purpose.

3.2.4 Extra-RDF operations

Jena provides convenience classes and methods to support DAML ontology constructs. These predate the OWL Working Group's standardisation efforts. Future versions of Jena will probably update this support.

3.2.5 Whole-graph operations

Jena offers graph isomorphism checking: in this version, this is somewhat ambitiously implemented as the `.equals()` method on a *Model*.

In addition, the *Model* class has methods for the difference, intersection and union of whole graphs at once.

3.2.6 RDF construct support

The support for RDF's containers is quite good, with automatic renumbering of ordinal properties to maintain wellformedness of the container construct. Jena 1.6 doesn't support collections directly, although the parser, ARP, understands the `rdf:parseType="daml:collection"` syntax.

3.2.7 RDFS construct support

Jena 1 does not support much in the way of RDFS; Jena 2 promises more support for RDFS manipulation and inference. However, the current support for manipulating DAML ontologies is very convenient.

3.2.8 Parser

Jena's parser, ARP, is fully-featured. It uses a callback mechanism to pass individual statements to an RDF consumer.

3.2.9 Serialiser

Jena's serialisation support is also good. It can target RDF/XML (with the option of taking advantage of RDF/XML's abbreviated forms where possible), N-Triples and N3.

3.2.10 Overlapping operations

Jena's *Model* interface provides explicit methods for starting, committing and aborting transactions, if the underlying storage facility supports such notions.

3.2.11 Storage implementation and selection

The stock implementation comes with a number of storage mechanisms. These are selected by instantiating the appropriate implementation of *Model*.

3.2.12 Query

Jena offers an interesting and useful query interface. Its facilities are currently based around SquishQL [<http://swordfish.rdfweb.org/rdfquery>]. A query can be specified either as a data structure or as a string; the results are given as a set of variable bindings that satisfy the query (similar to a JDBS ResultSet).

3.2.13 Inference

Jena 2 promises an implementation of RDFS inference. Currently this is not available.

Appendix A. References

[BDB]

[Berkeley DB](#) A popular embedded multi-platform database system.

[CWM]

[CWM](#) A general-purpose data-processor for the Semantic Web.

[FOAF]

[Friend Of A Friend](#) A collection of RDF applications based around the FOAF namespace.

[RDF-CONCEPTS]

[RDF Concepts and Abstract Syntax](#)

[RDF-PRIMER]

[RDF Primer](#) Including the abbreviated N-Triples syntax.

[RDF-SEMANTICS]

[RDF Semantics](#) The model-theoretic semantics for RDF; includes informative closure rules.

[RDF-SYNTAX]

[RDF/XML Syntax Specification \(Revised\)](#) The serialised form of RDF in XML.

[RDF-TESTCASES]

[RDF Test Cases](#) Including the N-Triples syntax.

[RDF-WD]

[RDF Primer](#); [RDF Concepts](#); [RDF Semantics](#); [RDF/XML Syntax Specification \(Revised\)](#); [RDF Vocabulary Description Language 1.0: RDF Schema](#); [RDF Test Cases](#). The RDF Working Drafts.

[REDLAND]

[The Redland RDF Application Framework](#) A multi-language, multi-platform implementation.

[SWIG]

[SWIG](#) Simplified Wrapper and Interface Generator.

[WEBONT]

© Web Ontology Working Group Producing the OWL family of languages.