A pragmatic approach to storing and distributing RDF in context using Snippets

Craig Sayers, Kevin Wilkinson
Enterprise Systems and Data Management Laboratory
HP Laboratories Palo Alto
HPL-2003-231
November 14th , 2003*

E-mail: craig_sayers@hp.com, kevin_wilkinson@hp.com

RDF, hash, context, Snippet, immutability, distribution, security

While RDF provides a powerful means to store knowledge, it can be cumbersome to represent and query collections of statements in context. To this end, we introduce a new higher-level object, the *Snippet*, to hold a fragment of RDF that is about a single subject and made within a particular context. Each snippet may be represented in a standard form as a bag of reified statements or in a non-standard compact form using many fewer triples.

Just as a quad serves as the internal representation for a single reified statement; so a compacted snippet serves as the internal representation for a bag of reified statements made about a single subject. Basic operations, such as storing and querying, may be performed on compact snippets without needing to expand them. Provided the application is aware of snippets this is possible using existing triple stores and query languages. In addition, the compacted snippets may always be expanded to permit reasoning using standard tools.

Particular consideration is given to the use of snippets in distributed applications. Algorithms are presented for computing content-based identifiers and for handling security using split-capabilities. An implementation is described.

* Internal Accession Date Only
Approved for External Publication

Contents

1 Introduction

The Resource Description Framework (RDF) [11, 14] is a system for making statements about resources. Each statement is a triple of the form *subject predicate object* and may be interpreted as a *subject* that has a *predicate* with value *object*.

In an interesting twist, and in a notable departure from simpler languages (like XML), RDF provides a well-defined means to make statements about statements[1]. This is termed reification (see Section 4).

Since reified triples can be cumbersome to store, it is common to have an internal representation which allows the triples representing a reified statement to be compactly stored as a quad [8, 6]. While this compact representation works well for individual reified triples, we expect that applications will want to collect statements together, storing not just the fact that there is a statement, but also the context in which sets of statements are made.

Accordingly, we desired the means to store a set of statements made about a particular subject and within a particular context. To this end we introduce a new higher-level object, which we call a *Snippet*.

We define two representations for a snippet. The expanded form is a bag of reified statements about a single subject. This is strictly conformant with the RDF specification; requiring no special treatment for querying, transporting or reasoning.

Since the expanded form is inconvenient and inefficient, we also define a compact representation for the snippet that maps the expanded form into a much smaller number of triples. Since the compacted snippet is represented entirely using triples, an application which understands snippets may query and manipulate them in their compact form using standard tools; only rarely needing to use the expanded form.

In addition to a space-saving, the snippet representation provides a convenient level of granularity for caching, distribution and security. Methods are described for using hashing to create immutable snippets and for providing security using a split capabilities system.

[1] Technically it is not *statements* about *statements* but rather *statements* about *statings* - readers concerned with such nuances will be well served by the latest RDF specification documents.

2 Presentation syntax

Throughout this paper, we'll present a number of example statements using a syntax of:

subject predicate object .

to indicate a Statement with that *subject*, *predicate*, and *object*. To ease readability we'll abbreviate most URIs to just a single word.

3 Illustrative example

For purposes of illustration we'll introduce an example with just four statements describing temperature and humidity readings. To make things interesting there are two sets of statements, each set made within a different context (perhaps at a different time, by a different sensor, or in a different location).

placeA temperature 20 .
placeA humidity 45 .

placeA temperature 30 .
placeA humidity 46 .

Storing all of those statements together is problematic - some means is needed to distinguish those apparently-conflicting statements and some means is also needed to encode the additional knowledge that the temperature and humidity reading were taken at the same instant.

4 Reification

The standard technique for handling statements made in different contexts is to use reification. For example, the statement:

placeA temperature 20 .

can be reified to give:

_1 type Statement .
_1 subject placeA .
_1 predicate temperature .
_1 object 20 .

Each reified statement is uniquely labeled and any number may co-exist independently within a single model. By making additional statements about those reified statements we can associate them with particular contexts and with each other. We can do this by using the rdf:Bag container. Again using the weather example:

_1 type Statement .
_1 subject placeA .
_1 predicate temperature .
_1 object 20 .

_2 type Statement .
_2 subject placeA .
_2 predicate humidity .
_2 object 45 .

_3 type Bag .
_3 _:1 _1 .
_3 _:2 _2 .

_4 type Statement .
_4 subject placeA .
_4 predicate temperature .
_4 object 30 .

_5 type Statement .
_5 subject placeA .
_5 predicate humidity .
_5 object 46 .

_6 type Bag .
_6 _:1 _4 .
_6 _:2 _5 .

This captures not only the original statements, but also the connection between

them. It is fully conformant with the specification and may be stored, transported and queried using standard tools.

Using so many triples looks extremely cumbersome, but in practice the reified statements are often stored internally in a quad form, so it really looks more like this:

_1 placeA temperature 20 .
_2 placeA humidity 45 .

_3 type Bag .
_3 _:1 _1 .
_3 _:2 _2 .

_4 placeA temperature 30 .
_5 placeA humidity 46 .

_6 type Bag .
_6 _:1 _4 .
_6 _:2 _5 .

Since this is expected to be a commonly-occurring pattern, we define a new higher-level object to handle this case. We call it a *Snippet*.

5 Snippets

We use a *snippet* to collect statements made about a single subject and within some context. (In previous work [15] we termed the precursor to Snippets an Object, but have changed the terminology here to avoid confusion).

5.1 Forward transform

If the original statements about a single subject, s, and in a context, C, have the form:

s p1 o1 .
s p2 o2 .
⋮
s pN oN .

Then we can represent those as an expanded Snippet using a bag of reified statements:

_1 type Statement .
_1 subject s .
_1 predicate p1 .
_1 object o1 .

_2 type Statement .
_2 subject s .
_2 predicate p2 .
_2 object o2 .
⋮
_N type Statement .
_N subject s .
_N predicate pN .
_N object oN .

S type SnippetX .
S type Bag .
S contextX C .
S _:1 _1 .
S _:2 _2 .
⋮
S _:N _N .

Here we have introduced a new type *SnippetX* to be a Bag of reified statements where all the statements are about a single subject and all are made within a particular context. We have also introduced a new predicate *contextX* to store the context within which the collected statements were made.

The new bag resource, S, is an automatically-generated node whose value is unique for every snippet. The use of this resource is somewhat analagous to the way the addition of the reification node allows apparently-conflicting reified statements to coexist within a single model.

Since this expanded snippet is clearly inefficient to store and query (even when using quads) we introduce a compact representation:

S type Snippet .
S about s .
S context C .
S p1 o1 .
S p2 o2 .
⋮
S pN oN .

Note that the type may usually be inferred by the presence of the *about* or *context* properties and so need not always be stored.

Notice that this compact representation is stored entirely using triples. An application which is aware of snippets may store and query the compacted snippet using existing triple stores/query languages.

5.2 Example

Taking the weather example from above, we can simply store that as two snippets:

S1 about placeA .
S1 temperature 20 .
S1 humidity 45 .

S2 about placeA .
S2 temperature 30 .
S2 humidity 46 .

Note that our example did not specify any particular context, so we can drop the contextual triples.

5.3 Inverse transform

Given a compacted snippet, we can convert that back into the expanded form for compatibility with existing reasoning tools and standard ontologies.

To do this we reintroduce bNodes for each statement and recreate the bag and the *contextX* predicate to store the context:

_1' type Statement .
_1' subject s .
⋮
_N' predicate pN .
_N' object oN .

S type SnippetX .
S type Bag .
S contextX C .
S _:1 _1' .
S _:2 _2' .
⋮
S _:N _N' .

It should be noted that these new statement bNodes will not usually be the same as those prior to compaction, so this only works when there are no other references to the same statements. In practice, we can usually guarantee this, since only an application which was aware of snippets could create a Bag of type SnippetX, and an application which is aware of snippets knows not to make references to the individual statement identifiers.

By handling the bag of reified statements using a standard triple structure we remove some of the need to efficiently handle the reification of individual statements. In particular it reduces the advantages of using a quad representation.

To summarize, we've introduced a higher-level object, the Snippet, to capture the notion of a set of statements about a single subject and made within a particular context. A snippet may be represented in standard RDF as a bag of reified statements about a single subject. In addition, we've introduced a compact representation for the same object along with forward and inverse transforms between the expanded and compact representations. In many cases, the snippet may be stored and queried in a compacted form within a standard triple store.

5.4 Handling bNodes

In the discussion so far all of the snippets have had an *about* property. In the special case where the original statements were about a bNode or resource that is not used elsewhere, then the *about* label may be discarded. For example, if the original statements to be collected in a snippet were:

_:1 creator C .
_:1 title T .

Then we can represent those as a compact snippet:

S1 type Snippet .
S1 creator C .
S1 title T .

In this case, we must explicitly state the type since the lack of any *about* or *context* property means it can't otherwise be inferred.

5.5 Statements about snippets

Just as one can make statements about statements, so it also makes sense to have statements about snippets. This requires some care.

Given an existing snippet, S1, it is tempting to make a statement about it by simply adding the statement:

S1 p o .

Unfortunately that doesn't work. When we compacted the snippet, those statements would be confused with statements defining the compact snippet itself. Instead we must make statements about a snippet by using another snippet.

5.6 Snippets about snippets

Given an existing snippet, S1, and a set of statements which we wish to collect together:

S1 p1 o1 .
⋮
S1 pN oN .

we can represent those in a compact snippet as:

S2 type Snippet .
S2 about S1 .
S2 p1 o1 .
⋮
S2 pN oN .

We have found it particularly convenient to use snippets about snippets ("Meta-snippets") to store system metadata such as security information.

5.7 Quads and context

While RDF does not define a formal notion of context, it is not uncommon for users to desire some way to represent that and a common technique is to use quads [9, 12]. We can map many of those contextual quads into snippets.

Specifically if the quad representation of statements about a single subject, s, in a context, C, is:

C s p1 o1 .
C s p2 o2 .
⋮
C s pN oN .

then we could map that into a snippet (using the compact representation):

S about s .
S context C .
S p1 o1 .
⋮
S pN oN .

6 Hashing to aid caching and distribution

In defining the *snippet* we mentioned that the snippet resource must be unique for each snippet. One simple way to achieve that would be to use a bNode. However bNodes are a poor choice if we wish to have snippets in one document/store refer to snippets in another document/store. So instead we suggest generating a unique URI (for example using a GUID-based scheme).

In our implementation we further need to distinguish each different instance of each snippet. In addition, when requesting a particular instance, we wanted to be able to verify that the returned snippet exactly matched what we had requested. Accordingly, we introduce the notion of a frozen instance of a snippet, computing a snippet URI which is a function of all the information in the snippet. This naturally implies that two snippets with identical content are identical and conveniently allows us to compare such snippets using only their URIs. The use of such content-based identifiers is particularly convenient for cache management in distributed stores.

Given an original snippet, compactly represented with the statements:

S p1 o1 .
S p2 o2 .
⋮
S pN oN .

a new URI, S', to represent an equivalent immutable instance of that snippet, is constructed by first computing a set hash of the snippet contents:

$$u = \sum_{i=1}^{N} h(p_i, o_i)$$

where $h(p, o)$ is the SHA-1 hash [4] of the serialization of the resource p and object o.

Once u is computed, S' is constructed:

$$S' = append(S, \texttt{"?sumsha1="}, u)$$

and for each statement:

S p o .

we add a new statement:

S' p o .

creating an immutable copy of that instance of that snippet.

Note that the use of the hash here is intended as a unique identifier suitable for comparing snippets and detecting errors.

In an advance over previous work [15], this is a particularly efficient way to compute the hash - since addition is associative and commutative, no sorting is required and the computation can be performed incrementally.

These immutable instances are of most value when all of the statements within them refer to either literals or other immutable snippets. This is not possible when a number of snippets form a cycle.

The presence of bNodes as properties reduces the benefits of hashing and complicates the verification of the hash after transporting the snippet. The preferred approach is to introduce additional snippets about each bNode (see Section 5.4) and replace the references to the bNodes with immutable instances of those newly-created snippets. An alternative is to add additional triples to encode the local label assigned to each bNode and use those to ensure the bNode naming is consistent across machines when computing the hashes.

Each immutable snippet instance has an identifier that serves as a short-hand for its entire contents. They are well-suited to distributed environments. You can store snippets from different sources within a single model without any danger of unforeseen interactions. You can transport snippets by serializing them using any of the RDF serialization standards. When retrieving an immutable instance from a remote store you can verify the returned statements match the instance you asked for. If two different agents make statements about the same snippet you can compare the URIs to see if they both refer to the same instance of that snippet.

It has previously been suggested that statements with a common subject might provide an appropriate granularity for caching and distribution [1]. Snippets provide a similar granularity, but provide the additional benefit of reification and context specificity.

The general idea of using a content-based identifier is not new [7]. An alternative to computing hashes over a snippet is to compute the hash of an entire RDF document. Mechanisms for doing that have been defined [3, 2] and a URN scheme, the "Secure Definition Hash" has been proposed [5].

7 Security

The simplest approach is to provide security at the Model level. This is equivalent to a traditional database notion of security, having access controls on each Table.

One approach for providing secure RDF queries is to provide security by filtering query results based on the identified role of an authenticated user [2]. That is appealing for some applications, but we wanted to avoid the need to authenticate users.

Security at the level of contexts has previously been used in the RDF Gateway

though use of the quad representation [9] with a conventional access control approach.

Access control lists could also have been used to provide security for snippets. One obvious approach would be to use a meta-snippet to record lists of users and their accorded access rights for each snippet. To avoid the maintenance and overhead of an access control list, we chose an alternative approach.

Security for snippets is provided by a simple split-capabilities system [10]. It permits operation in a purely distributed fashion, requiring no access control lists or central server. Each snippet may be individually secured. This is a compromise between having very fine-grained security at the Statement level and a very coarse-grained security at the Model level.

7.1 Operation

When a snippet is placed in a secure store, we require that at least one key be provided for each facet of interaction with that snippet. Then, when the content provider wishes to give someone else access, they provide both an identifier and an appropriate key.

Specifically, when a new snippet is added to the store, the content author provides both initial content for the item and a list of (Facet,Key) pairs, where each pair encodes a key which will unlock a particular secured facet for the item. Again, the key is provided by the content author, so it is their responsibility to choose a suitably unguessable key and to take care in sharing/distributing it.

Each incoming request is of the form (URI, key) – the system simply looks up the URI, verifies that the key matches the appropriate facet for the requested operation and then, assuming it matches, performs the requested operation. Notice that the system doesn't need to verify the identity of individuals. Instead it just verifies that each access is accompanied by the correct key. Anyone with the right key gains access.

Each content producer may choose how to re-use keys. For very secure items, he or she might choose to use a unique key for each facet of each snippet. For less secure operations, he or she might re-use keys for objects and operations. Clearly there is a trade-off. The less is done with each key, the more secure the system is, but the more keys must be managed. It is the content producer's choice.

In our implementation the security facets and keys for a snippet are stored as RDF statements in a meta-snippet. To support the transmission of keys and support secure operation we require secure communication and trusted servers.

8 Practical Example

To show the use of snippets, we'll use an example suggested by Bob MacGregor during a discussion of Quads on the RDF_Interest mailing list [12]. The idea was to store data so one could later answer questions like:

> "Retrieve freighters that visited Antwerp on April 2003 whose cargo included aluminum pipes"

Here's an example of data that we've made up for a single freighter and stored using a snippet:

f1 type Freighter .

s1 about f1 .
s1 location Antwerp .
s1 hasCargo _1 .
_1 consistsOf Pipes .
s1 context c1 .

c1 beginDate "Jan 2003" .
c1 endDate "Mar 2003" .

And here is an example query over that data - note that so long as we know we're querying a snippet, we can use standard query languages to query the snippet without needing to expand it:

```
SELECT ?f
WHERE ((?f type Freighter),

       (?s about ?f),
       (?s location Antwerp),
       (?s cargo ?cargo),
       (?cargo consistsOf Pipes)
       (?s context ?c)

       (?c beginDate ?begin),
       (?c endDate ?end),

       (?begin < "April 2003" ),
       (?end > "April 2003"),
```

Recently, MacGregor and Ko proposed an alternative representation [13] called a snapshot. Representing the same data in their snapshot format gives:

f1 type Freighter .

_:ssf1 theRealThing f1 .
_:ssf1 location Antwerp .
_:ssf1 hasCargo _:cargo1 .
_:cargo1 consistsOf Pipes .
_:ssf1 inContext _:cxt1 .

_:cxt1 beginDate "Jan 2003" .
_:cxt1 endDate "Mar 2003" .

This is very similar to the snippet. The difference is in the details. They define the meaning of "theRealThing" as being a modified form of owl:sameIndividualAs, whereas our snippets are a compact representation for a bag of reified statements.

In the above we've tried to closely follow MacGregor's style. Our own preferred approach is to treat the spatial and temporal constraints similarly:

f1 type Freighter .

s1 about f1 .
s1 location Antwerp .
s1 arrival "Jan 2003" .
s1 departure "Mar 2003" .
s1 hasCargo _1 .
_1 consistsOf Pipes .

and the query then simplifies slightly to:

```
SELECT ?f
WHERE ((?f type Freighter)

       (?s about ?f),
       (?s location Antwerp),
       (?s hasCargo ?cargo),
       (?s arrival ?begin),
       (?s departure ?end),
       (?cargo consistsOf Pipes))
```

```
(?begin < "April 2003"),
(?end > "April 2003"))
```

In addition, by further modifying the example, we can make use of immutability and security. Specifically, if we remove the bNode, by creating a snippet for the cargo as well:

f1 type Freighter .

s2 type Snippet .
s2 consistsOf Pipes .
s1 about f1 .
s1 location Antwerp .
s1 arrival "Jan 2003" .
s1 departure "Mar 2003" .
s1 hasCargo s2 .

then we can create an immutable copy of each instance of each snippet[2], forming an immutable arrival/departure record whose value depends on its contents (including the cargo).

f1 type Freighter .

s2?sumsha1=114...4 type Snippet .
s2?sumsha1=114...4 consistsOf pipes .

s1?sumsha1=2bb...b" about f1 .
s1?sumsha1=2bb...b" location Antwerp .
s1?sumsha1=2bb...b" arrival "Jan 2003" .
s1?sumsha1=2bb...b" departure "Mar 2003" .
s1?sumsha1=2bb...b" hasCargo s2?sumsha1=114...4 .

Now we can make statements about that immutable record, knowing that anyone to whom we give those statements can verify that their copy of the freighter's record exactly matches the one that we saw.

Furthermore, to associate security information with those snippets we can introduce a SystemInformation context and use snippets about snippets:

[2]Note that the ordering is important here. We must compute the identity of the instance of the cargo record first so that the arrival/departure record may refer to it

s3 about s2?sumsha1=114...4 .
s3 context SystemInformation .
s3 hasReadKey “sfjlsjl24jlj” .
s3 hasQueryKey unlocked .

s3 about s1?sumsha1=2bb...b” .
s3 context SystemInformation .
s3 hasReadKey “sfjlsjl24jlj” .
s3 hasQueryKey “2sjfljwerjwlj” .

9 Implementation

We are currently experimenting with APIs for easing interaction with snippets. A preliminary implementation has been built in Java on the Jena [6] API.

This is presented again using the freighter example. First we construct a snippet describing the cargo:

```
Snippet s2 = myWorld.createSnippet();
s2.addProperty(consistsOf, pipes);
```

and create an immutable copy of that instance of the snippet:

```
Snippet cargoRecord = myWorld.freeze(s2);
```

Using similar code an immutable arrival/departure record in stored:

```
Snippet s3 = myWorld.createSnippet(f1);
s3.addProperty( location, "Antwerp");
s3.addProperty( arrival, "Jan 2003");
s3.addProperty( departure, "Mar 2003");
s3.addProperty( hasCargo, cargoRecord);
Snippet shipsRecord = myWorld.freeze(s3);
```

We are currently experimenting with distributed stores. The availability of immutable snippets and the means to secure them is expected to prove beneficial. In particular there is an appealing similicity to storing all shared content as immutable snippets.

10 Conclusions

Snippets provide the means to represent a set of statements made about a particular subject and in a particular context. They may be stored conventionally using a bag of reified statements, or in a non-standard compact form using a much smaller number of triples. This compact representation is somewhat analogous to the use of quads to represent individual reified statements. The difference is that, while quads fall completely outside the specification, snippets may still be queried and transported in their compact form. The application making the query needs to know about snippets but the underlying query tools and stores do not.

By collecting a number of related statements together, the snippet provides a convenient level of granularity - small enough to be cached and transported easily, but large enough that any overhead can be amortized over several statements.

The use of immutable snippets, whose identifiers are dependent upon their content, should prove helpful when distributing and caching content. In addition, the use of a split capabilities security system appears to provide a reasonable tradeoff between security and complexity.

11 Acknowledgements

Special thanks to Alan Karp for a number of discussions which contributed to this document (especially the suggestion of using a set hash and pointing us to the split capabilities work) and to Dave Reynolds and Ian Dickinson for suggesting significant improvements to a early draft. Thanks also to Bernard Burg, Kave Eshegi, Harumi Kuno, and Kevin Smathers for discussions which contributed to this document. We're also very appreciative of the advice and assistance provided by the entire HP Labs Bristol semantic web team. Several of those acknowledged here have not yet seen this final document, so please blame the authors for anything disagreeable.

References

[1] D. Banks. Some thoughts on metadata caching. Personal communication, February 2003.

[2] D. Banks, S. Cayzer, I. Dickinson, and D. Reynolds. The ePerson snippet manager: a semantic web application. Technical Report HPL-2002-328, Hewlett Packard Labs, Bristol, England, November 2002.

[3] J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. The Jena semantic web platform: Architecture and design, 2003.

[4] Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, Virginia. *FIPS 180-1: Secure hash standard*, April 1995.

[5] S. Hawke. Identification via secure definition hash: A solution to the semantic web identification problem. http://www.w3.org/2002/09/sdh/, October 2002.

[6] Hewlett Packard Laboratories, http://www.hpl.hp.com/semweb. *Jena: A Java API for RDF*, 2002.

[7] J. K. Hollingsworth and E. L. Miller. Using content-derived names for configuration management. In *ACM Symposium on Software Reusibility, Boston, MA*, May 1997.

[8] I. Inc. Intellidimension. http://intellidimension.com/.

[9] Intellidimension Inc. RDF Gateway developer guide, Context based security. http://www.intellidimension.com/pages/rdfgateway/devguide/security/context.rsp, October 2003.

[10] A. Karp, G. Rozas, A. Banerji, and R. Gupta. Using split capabilities for access control. *IEEE Software*, 20(1), January 2003.

[11] O. Lassila and R. Swick. Resource description framework (RDF) model and syntax specification, W3C recommendation. http://w3.org/TR/1999/RED-rdf-syntax-19990222, February 1999.

[12] B. MacGregor. Representing temporal data in RDF. http://lists.w3.org/Archives/Public/www-rdf-interest/2003Sep/0055.html, September 2003.

[13] B. MacGregor and I.-Y. Ko. Representing contextualized data using semantic web tools. In *First International Workshop on Practical and Scalable Semantic Systems*, October 2003.

[14] F. Manola and E. Miller. Resource description framework (RDF) primer, W3C working draft. http://w3.org/TR/2002/WD-rdf-primer-20020319, March 2002.

[15] C. Sayers and K. Eshgi. The case for generating URIs by hashing RDF content. Technical Report HPL-2002-216, Hewlett Packard Laboratories, Palo Alto, California, August 2002.