Extensible SPARQL Functions With Embedded Javascript

Gregory Todd Williams

University of Maryland, College Park, MD, USA
gtw@cs.umd.edu

Abstract. The SPARQL Query Language allows filtering of query results through arbitrary predicate expressions. Such expressions may invoke custom functions identified with IRIs, but the SPARQL implementation used must support the identified function. We present an extensible approach to allowing arbitrary function implementations using functions identified with URLs. We provide an example using Javascript functions, show how such a system can be implemented in the Perl based RDF::Query SPARQL implementation and discuss security concerns.

1 Introduction

The SPARQL query language[7] has provided a powerful, standardized way to query RDF models. Many query engine implementations exist that support SPARQL[4], and they allow consistent querying regardless of implementation language or platform. One of the most exciting features of SPARQL is its support for extension functions. These functions, identified by IRI, allow SPARQL queries to rely on domain-specific approaches to filtering query results.

Extension functions provide a means of extending the SPARQL language, but support for such functions remains sparse. Furthermore, to allow SPARQL query engines to share support for specific extension functions, implementations of the functions must be provided in each of the programming languages used.

We present a system in which extension functions are implemented in an agreed upon programming language and implementations may be shared among query engines by using an embedded interpreter for the language. Functions are identified by URLs (a subset of IRIs) and the source code may be retrieved at run time by dereferencing the URL. Such a system reduces duplicated effort in implementing extension functions, and allows reference implementations to define the semantics of functions.

2 Background

To make use of SPARQL extension functions, SPARQL query engines must implement functionality to allow users (or administrators) to register functions with the query engine, mapping an IRI to a function. Unfortunately, many query engines don't yet support this feature.

One popular query engine that does support this feature is ARQ[8] which allows querying of Jena[6] models. Functions may be registered with ARQ by installing a mapping between a URI and and a Java factory class. As a convenience shortcut to avoid the registering process, ARQ allows functions to be automatically loaded by using a URI with the "java:" IRI scheme. For example, using functions in the `java:com.hp.hpl.jena.query.function.library.` namespace will automatically make use of functions in the `com.hp.hpl.jena.query.function.library` package. Figure 1 shows an example SPARQL query with an extension function using the "java:" IRI scheme. Under ARQ, this example would dynamically load and execute the `com.ldodds.sparql.Distance` function, filtering any potential results whose corresponding return value was not less than a constant value.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX geo:     <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX ldodds:  <java:com.ldodds.sparql.>
SELECT ?image ?placename
WHERE {
  ?image a foaf:Image .
  ?image dcterms:spatial ?point .
  ?point foaf:name ?placename .
  ?point geo:lat ?lat ; geo:long ?long .
  FILTER(
    ldodds:Distance(?lat, ?long, 38.9937, -76.933) < 10
  ) .
}
```

Fig. 1. An example ARQ SPARQL query finding all images taken within 10 kilometers of a known point using the `com.ldoddds.sparql.Distance` extension function.

Another query engine that supports extension functions is the Perl-based RDF::Query[10] package. In RDF::Query, IRIs may be mapped to user functions using the `add_function` object or class method to register the function either per query or globally for all queries, respectively.

Extension functions may be used in several different ways. Some extension functions are used as shorthand for accessing complex RDF structures. `jena:listMember` is used to test for membership in an `rdf:Seq` sequence[6]. Other extension functions are used to transform scalar values; `jena:sha1sum` is used to return the SHA-1 cryptographic hash of a value. A third type of function generates a new value from sets of values. `ldodds:Distance` can be used to compute the distance in kilometers between two sets of latitude/longitude values[5].

Despite having broad usefulness, these functions and many others are only available on one or a few query engine implementations, restricting their use.

In some situations, the need to use such a function may be addressed by using a specific query engine. However, in situations where a (potentially large) database is hidden behind a SPARQL query engine "endpoint" (where an existing SPARQL engine is the only point of access to the underlying data), the user may be forced to use the query engine that is provided.

What is needed is an approach to extension functions that allows functions to be implemented once, and shared between query engines without *a priori* knowledge of any particular function.

3 Methodology

Our system extends the RDF::Query query engine with the ability to retrieve Javascript extension function implementations via URL, run the extension functions in an embedded Javascript interpreter, and return the resulting value to the RDF::Query engine as a native Perl object. Javascript was chosen for its wide use on the internet, existing RDF APIs for Javascript, and the availability of several high quality, free implementations suitable for embedding [1, 9]. However, our approach is general, and could work with any embeddable language.

Our system makes use of an RDF schema to describe SPARQL extensions, a SPARQL extension function API for Javascript, and optional cryptographic signatures to distinguish "trusted" implementations. We discuss these components below, together with notes on the actual implementation in RDF::Query.

3.1 Extension Function Schema

The extension function namespace is:

```
http://www.mindswap.org/2007/owl/sparql#
```

The RDF document describing functions is used as a central location for information about the implementation of the functions. It should contain data relating to the location of implementation files and any cryptographic signatures, a description of the referenced functions, a name for each function (to be used as an entry point into the code and that will return the function value), and any other information relevant to the implementation.

An example RDF document describing two of the jena functions using the extension function schema is shown below in the Turtle[2] syntax:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ex: <http://www.mindswap.org/2007/owl/sparql#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<java:com.hp.hpl.jena.query.function.library.>
    a ex:Namespace;
    ex:hasFunction
        <java:com.hp.hpl.jena.query.function.library.sha1sum>,
```

```
        <java:com.hp.hpl.jena.query.function.library.now> .

<java:com.hp.hpl.jena.query.function.library.sha1sum>
    a ex:Function;
    dc:description "SHA-1 Hash";
    ex:source <http://example.com/sha1.js>;
    ex:signature <http://example.com/sha1.js.asc>;
    ex:function "sha1" .

<java:com.hp.hpl.jena.query.function.library.now>
    a ex:Function;
    dc:description "Start time of query execution";
    ex:source <http://example.com/now.js>;
    ex:signature <http://example.com/now.js.asc>;
    ex:function "now" .
```

The important statements of this RDF description for purposes of implementation are those using the following predicates:

- `ex:source` declares the location of the implementation source code.
- `ex:function` defines the function name in the source code that should be called to execute the function.
- Optionally, `ex:signature` declares the location of any cryptographic signatures that can be used by query engines in order to run only trusted functions. This is discussed in more detail in sections 3.3 and 4.1.

3.2 Extension Function API

The implementation of extension functions must take as input RDF nodes passed as function arguments and output an RDF node value. Therefore, a simple API for interacting with RDF nodes is needed in the implementation language. We use the AJAR[3] Javascript API, providing the classes `RDFLiteral`, `RDFBlankNode`, and `RDFSymbol`, the common property `termType`, the common method `toString`, and the node-creating function `makeTerm`. For a more detailed description of the API, refer to the AJAR documentation.

Figures 2 and 3 show corresponding RDF description and Javascript implementation of a distance function using the Javascript API. This function is similar to `ldodds:Distance`, taking two latitude/longitude pairs, and returning the distance between them in kilometers as computed using the great circle distance algorithm.

3.3 Cryptographic Signatures

For security and performance reasons, it may be desirable for query engine instances to only allow running extension functions that have been identified as trusted. This can be accomplished by designating known third-parties as trusted,

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ex: <http://www.mindswap.org/2007/owl/sparql#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://example.com/functions/>
    a ex:Namespace;
    ex:hasFunction <http://example.com/functions/distance> .

<http://example.com/functions/distance>
    a ex:Function;
    dc:description "Geographic (great circle) distance in kilometers";
    ex:source <http://example.com/distance.js>;
    ex:function "gcdistance" .
```

Fig. 2. RDF description of a distance extension function.

```
function gcdistance( lat1, lon1, lat2, lon2 ) {
  lat1  = deg2rad( makeTerm(lat1).toString() );
  lat2  = deg2rad( makeTerm(lat2).toString() );
  lon1  = deg2rad( makeTerm(lon1).toString() );
  lon2  = deg2rad( makeTerm(lon2).toString() );

  var londiff = Math.abs(lon1 - lon2);
  var s1    = square(Math.sin((lat2 - lat1) / 2));
  var s2    = square(Math.sin( londiff / 2 ));

  var sq  = Math.sqrt(
            s1
            + Math.cos(lat1)
            * Math.cos(lat2)
            * s2
          );

  var adist = 2 * Math.asin( sq );
  var r     = 6372.795;
  return r * adist;
}

function square (x) { return x * x; }
function deg2rad(d) { return Math.PI*d/180 }
```

Fig. 3. Javascript implementation of a distance extension function.

and trusting any implementation that has a valid cryptographic signature from the trusted third-party.

If configured to allow only trusted functions, our implementation takes a list of trusted GPG public key fingerprints, attempts to verify all available signatures of a requested implementation, and only proceeds if there exists a signature created with a trusted key.

The need for a possibly growing list of arbitrary signatures in the RDF description is the primary reason why function metadata and implementation source code must reside at distinct URLs. As new signatures are generated, metadata describing the signatures must be added to the RDF description. However, this addition must not invalidate existing signatures. Therefore, signatures must be made against the implementation source code and referenced in the seperate function metadata RDF document.

3.4 Extending RDF::Query

We implemented this system in the Perl-based RDF::Query query engine using an embedded Javascript interpreter. During query execution, an extension function is retrieved as a code reference via a lookup method. This method first looks for a native implementation of the extension function. If no implementation is found, and the function is identified with a URL, the URL is dereferenced and the resulting content is interpreted as an RDF document describing the function implementation. Using this RDF data, the implementation source code is retrieved, compiled, and verified against any signature files. If this process is successful, a code reference is returned that will behave just as a native implementation would.

Figure 4 shows example code for constructing an RDF::Query query engine, enabling the use of URL-based extension functions, and restricting their use to only those signed by a known key. In this example, a Javascript implementation of the SHA-1 hashing function is used to find all people with an email address that hashes to a known value.

In the construction of the query engine object, two new constructor fields are introduced to support the URL-based extension functions:

- `net_filters` is passed with a true value to enable URL-based extension functions.
- `trusted_keys` is passed with an array reference containing a list of the trusted signing keys' cryptographic fingerprints. This field is optional.

After the first use of an extension function, the source code and signatures are cached to prevent unnecessary repeated downloads. Using features of HTTP/1.1, the content of the source and signatures need only be downloaded again if these documents change. On repeated requests where the documents have not changed, a response with only a small header is transfered. This approach saves bandwidth while ensuring that changes to the source code and signatures are not ignored due to arbitrarily long caching – an important feature for long running query

```
my $sparql = <<"END";
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  PREFIX func: <http://example.com/functions.rdf#>
  SELECT ?person
  WHERE {
    ?person a foaf:Person ;
    foaf:mbox ?mbox .
    FILTER(
      func:sha1(?mbox) = "f80a0f19d2a0897b89f48647b2fb5ca1f0bc1cb8"
    ) .
  }
END
my $query = RDF::Query->new( $sparql, undef, undef, 'sparql',
  net_filters  => 1,
  trusted_keys => ['1150 BE14 FF91 269F 398B 0F4E 0253 5AF9 A2B9 659F'],
);
```

Fig. 4. Setting up an RDF::Query query engine with support for network-based extension functions only when signed by a known key.

engines. Furthermore, if the source code was verified against a signature, the verified status of the code may be cached when using the same trusted key and source code to avoid unnecessary repeat verifications.

4 Results and Analysis

The implementation of URL-based extension functions in RDF::Query was relatively straightforward. Just sixteen lines of existing code were modified, and three new methods were added to the main query engine class to initialize an embedded Javascript interpreter and to compile and call the extension function. There are several issues of concern with such an implementation relating to potential security and performance degredation. These issues are discussed below.

4.1 Security and Performance Concerns

With many SPARQL endpoints being made publicly available for use by the public, thought must be given to the potential to abuse these systems. Relating to URL-based extension functions, such abuse could affect overall system performance, or even cause a denial of service attack against external machines.

Query engines that may run unknown code loaded as an extension function must be aware of the potential problems doing so may cause. A denial of service attack may be initiated against a foreign server by using an extension function URL pointing to a very large resource (in which case available local bandwidth

may be a concern) or a resource that is computationally intensive to return (where the total system load of the remote server may be a concern). Similar attacks may be initiated if the embedded Javascript environment allows the creation of network connections.

Another potential problem of running arbitrary code is local system load. If an extension function runs for an unexpectedly long time, it may prevent the server from answering other requests or may overrun a time limit on the request (as might be likely if the request was made using HTTP). Even if the running time of a function is bounded, system load may still be of concern if the overhead of switching contexts between the query engine and the embedded language is high. If the potential result set before filtering is large, and many calls to the extension function are necessary, the context switching overhead may overwhelm the total running time of the query.

Public endpoints may prevent many potential problems by allowing only extension functions signed by trusted keys. By relying on a trusted third party to sign a function, the signature may be used to indicate the correctness of the function (by a lack of syntactic and semantic errors) as well as the absence of malicious code. This approach may work well for domain-specific functions where a central organization or interest group could provide the trusted signatures.

To address denial of service attempts and performance concerns, a query engine may also define configurable maximums for source code size and run time. In situations where a maximum run time isn't acceptable (where query results must be provided despite run time), total run time and the overhead of context switching may be addressed by providing a native implementation of commonly used extension functions. Native implementations may also be important for functions where a native implementation can provide much better performance (due to more efficient data structures, access to hardware acceleration, etc.).

5 Conclusion and Future Work

We believe this approach to allowing arbitrary extension functions can provide commonly used functions to many query engines with comparatively little work. However, to realize this potential and to be generally useful, our approach would need implementation in more than just one query engine.

Future work is needed to implement parts of the Javascript API to allow extension functions to access the underlying RDF model, allowing implementation of model-dependant functions such as jena:listMember. Future work may also include extending the RDF schema to allow for alternate programming language and cryptographic signature types. A language designed specifically for embedding (such as Lua) may prove more desirable for the implementation of functions, while allowing other signature types (x.509 certificates, for example) could allow leveraging existing key infrastructures. Finally, we hope to follow this work with a deeper investigating of the real-world applications and benefits of such a system.

This work was supported by grants from Fujitsu, Lockheed Martin, NTT Corp., Kevric Corp., SAIC, the National Science Foundation, the National Geospatial - Intelligence Agency, DARPA, US Army Research Laboratory, and NIST.

References

1. Webkit. http://developer.apple.com/opensource/internet/webkit.html.
2. Dave Beckett. Turtle - Terse RDF Triple Language. http://www.dajobe.org/2004/01/turtle/.
3. Tim Berners-Lee, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, and David Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *3rd International Semantic Web User Interaction Workshop*, November 2006.
4. Christian Bizer and Daniel Westphal. Update on semantic web toolkits for scripting languages. In *2nd Workshop on Scripting for the Semantic Web (SFSW2006)*, June 2006.
5. Leigh Dodds. Sparql geo extensions. http://xmlarmyknife.org/blog/archives/000281.html.
6. Brian McBride. An introduction to rdf and the jena rdf api. http://jena.sourceforge.net/tutorial/RDF_API/.
7. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF (Working Draft). http://www.w3.org/TR/rdf-sparql-query/.
8. Andy Seaborne. ARQ - A SPARQL Processor for Jena. http://jena.hpl.hp.com/ afs/ARQ/.
9. Jens Thiele. Embedding spidermonkey - best practice. http://egachine.berlios.de/embedding-sm-best-practice/embedding-sm-best-practice.pdf.
10. Gregory Todd Williams. RDF::Query. http://search.cpan.org/dist/RDF-Query/.