RESTful SPARQL? You Name It!

Aligning SPARQL with REST and Resource Orientation

Erik Wilde
dret@berkeley.edu
School of Information, UC Berkeley, USA

Michael Hausenblas
michael.hausenblas@deri.org
DERI, National University of Ireland, Galway

ABSTRACT

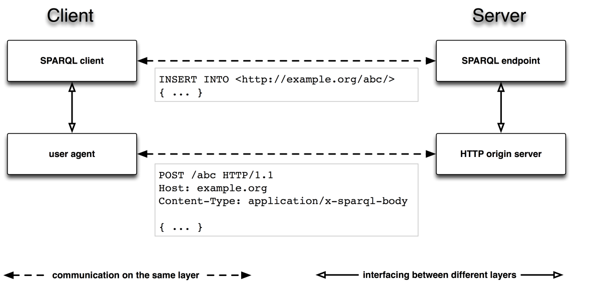
SPARQL is the standard query language for RDF, but currently is a read-only language defined in a way similar to SQL: Queries can be formulated, are submitted to a single processing facility, which then returns a result set. In this paper, we examine the shortcomings of this approach with regard to Web architecture, and propose a path towards a language that is more in line with basic principles of Web architecture. While this work has been done in the context of a proposed update extension for SPARQL, our focus is on how to apply the principles of *Representational State Transfer (REST)* to SPARQL. Our claim is that a RESTful redesign of SPARQL allows the Semantic Web to evolve in a more decentralized and openly accessible way than the current RPC-style design of SPARQL.

1. MOTIVATION

SPARQL [6] is the standard query language for RDF, widely adopted and is currently extended [5] into several directions, such as aggregates and update functions. While adding features is certainly desired by the Semantic Web community at large, we observe the need to discuss the design of the protocol [1]. We will show in the following that the current SPARQL protocol design is not RESTful in the sense of the *Resource-Oriented Architecture (ROA)* [7] and propose a way to overcome this situation. Our proposed solution additionally addresses the planned SPARQL update features and hence provides a route towards a RESTful Semantic Web.

As we have pointed out previously [3, 4], the realization of a read-write Semantic Web depends heavily on a scalable and sound update mechanism concerning RDF graphs. However, as we will show in the following, there is a schism between the RDF, graph-based world, and the document-oriented Web at large. The fundamental reason for that is that Semantic Web approaches often operate on a big graph of RDF triples, either virtual or actually available from one central place, whereas the Web and its underlying principle of *Representational State Transfer (REST)* [2] is based on a decentralized model where interactions with individual resources are possible, based on a uniform interface, and questions of data provenance, authenticity and integrity, and deeply intertwined with this general interaction model.

Our work explores the possibilities and limitations of mapping SPARQL query forms[1] (`SELECT`, `CONSTRUCT`, etc.), as well as update forms (`INSERT`, `DELETE`, etc.), that are subject to standardization as time of writing, to HTTP methods, such as *`GET`*, or *`PUT`* (note that, in order to avoid confusion we will keep this formatting, i.e., italic fonts for HTTP methods, normal typewriter for SPARQL forms). Figure 1, though illustrative rather than descriptive, depicts the focus of our work at hand, which is the *interfacing between the layers*. It is illustrated how the current protocol approaches HTTP as a transport protocol, instead of using it as the application protocol.



```
INSERT INTO <http://example.org/abc/>
{ ... }
```

```
POST /abc HTTP/1.1
Host: example.org
Content-Type: application/x-sparql-body

{ ... }
```

Figure 1: SPARQL over HTTP conceptual layering

The paper is structured as follows: In Section 2 we briefly review both the REST architecture style and SPARQL. Then, in Section 3 we highlight the central issues around how to redesign SPARQL to be more RESTful. We present our solution to the alignment problem on a theoretical and practical level in Section 4, and finally discuss our findings as well as outline future work in Section 5.

2. SPARQL AND REST

RDF is the core specification of the Semantic Web. It is a graph-based metamodel that is based on triples of *subject*, *predicate*, and *object*, where *subject* and *predicate* are always URIs, whereas the *object* can be a value or a URI. RDF itself makes no assertions about how triples or graphs are accessible, and it has no higher-level concept of *collections* or

[1]`http://www.w3.org/TR/rdf-sparql-query/#QueryForms`

WEWST 2009, November 9, 2009 Eindhoven, Netherlands
Copyright 2009 ACM 978-1-60558-776-9/09/11 ...$5.00.

documents. SPARQL introduces such a concept in the form of *named graphs*, but this concept only exists in the context of SPARQL, is not necessarily supported in all SPARQL implementations, and is not part of any interaction model around SPARQL-accessible collections of RDF data.

REST is the architectural style underlying the Web itself, it has been distilled from the core specifications of the Web, and the most important constraint is that in RESTful systems, everything is exposed as a resource via a uniform interface. This is different from traditional middleware and other distributed systems approaches, which often start from the procedural model of local programming approaches, and then extend this to *Remote Procedure Call (RPC)* models in an attempt to hide distribution and failure and to allow programmers to work with a distributed system as they would with a local system. REST, on the other hand, is built around the idea of decentralized systems, which means that it is not a design goal to hide distribution; on the contrary, accessing remote resources is made explicit through using their identifiers, and programmers have to take into account that remote accesses can fail.

While REST is simply an architectural style, various acronyms have been proposed to name the instantiation of that style in the Web technologies we have today, most importantly URIs, HTTP, HTML, media types, and XML. *Resource Oriented Architecture (ROA)* and *Web Oriented Architecture (WOA)* both are terms that are being used, but for the purpose of this paper, we will continue to use the term REST, but will refer to the specific technologies used in today's Web, and not just the general architectural style.

2.1 RESTful Architectures

RESTful design revolves interactions around resources; resources identify all relevant concepts in the application space (information resources as well as state), interactions are based on a uniform interface implemented for all resources, resources can have various media types (identification of the media type is part of uniform interface), and applications interact with RESTful services by traversing a network of resources interconnected by links.

RESTful architectures differ from RPC-style architectures mainly in their more explicit way of dealing with distribution (instead of trying to hide distribution, applications must be aware of the fact that they are interacting in a distributed, unreliable, and possible hostile environment), and in turning the traditional RPC model upside down [9]. In RPC, the prime abstraction is the function that can be called, and objects in services often are identified, but this identification typically is only within the context of a particular service or application. In REST, on the other hand, identification via URI is the primary architectural concern, and functions are not all that important because the usage of a uniform interface for interactions makes it much easier to understand the possible interactions with a resource.

The debate around the advantages of RESTful vs. RPC-style services (mostly in the form of SOAP/WS-* services) is still ongoing. The biggest difference is probably the main focus: RPC-style services focus on development-time advantages, have good support for tooling (such as generating stubs from interface descriptions), and generally try to provide programmers with the convenience of an environment that tries to hide distribution as much as possible, creating the perception of programming a single system. REST, on the other hand, focuses more on loose coupling, runtime issues such as the ability to rely on a uniform interface, and forces programmers to accept the fact that they are developing in an environment that is heterogeneous, unreliable, and possible hostile. This different "world view" is well reflected in the two biggest user groups of RPC vs. REST: RPC is most popular in enterprise IT, which often starts with the explicit goal of building one integrated system, and traditionally has been using middleware platforms. REST is most popular in the world of smaller players in the context of the Web, where it is necessary to quickly expose, use, reuse, and repurpose existing services, and where there is not even an implicit goal of building one integrated system: the world of the Web is loosely coupled, heterogeneous, and the easier services can be used, the more they will be used. This is highlighted by the fact that many of the major players on the Web (such as Google and Flickr) have seen much less uptake of their SOAP APIs than of their REST APIs, and sometimes even stopped supporting the SOAP API due to a lack of interest.

Another important issue about RESTful services is that they can be easily mixed and remixed, because they implement the same interface and resources are addressed in a way that is consistent across services. This makes it easy to combine RESTful services, but as a prerequisite for this, these services must properly expose all relevant resources via URIs, and must support HTTP interactions with those URIs, so that the services can be used in a RESTful fashion.

2.2 The SPARQL Protocol

SPARQL [6] can be regarded as being on the same level as SQL: It is a query language based on a certain metamodel, that operates on a database storing data of that metamodel, and returns a query result in that metamodel. SPARQL as such has little to do with the Web's RESTful underpinnings, it is just a back-end language for querying a database. The SPARQL protocol [1] is providing a means how to package a request and a response in SOAP or in HTTP messages, but it does not change SPARQL's basic approach of an "endpoint" which is returning a result that is not embedded into a network of URI-identified resources that support HTTP interactions.

2.2.1 The Current State

The SPARQL Protocol defines a means of conveying SPARQL queries from query clients to query processors. The protocol essential comprises two parts, (i) an abstract interface independent of any concrete binding to another protocol, and (ii) HTTP and SOAP bindings of the abstract interface. The SPARQL Protocol is described abstractly with the *Web Services Description Language (WSDL).* We will focus on the HTTP binding in the following, as this seems to be the more Web-oriented approach.

In fact, there are two HTTP bindings defined in the SPARQL Protocol, `queryHttpGet` and `queryHttpPost`, where the latter should be used in cases where the URL-encoded query exceeds practical limits. This design violates basic REST principles, since the request method used in the uniform interface (in this case, the HTTP protocol) has to be based on the interaction semantics, not on implementation details. In REST, this difference is important because *`GET`* is supposed to be idempotent and safe, whereas *`POST`* has none of those properties. Since the current SPARQL version is read-only, the best conceptual choice would be to use *GET*. *POST* has been chosen because for longer queries the URI-based query syntax may exceed length limitations of HTTP implementations (often around 4KB). While it is possible to use *GET* requests with a message body (which could be used to submit the query), the current HTTP specification unfortunately is silent about whether this is allowed or disallowed, and consequently, implementations differ in their handling of *GET* requests with message bodies.

Conclusion.

The above discussed, "read-only" version of the SPARQL Protocol is (almost) RESTful.

2.2.2 The Planned Extensions

As outlined in the planned features of the W3C SPARQL Working Group[2]:

> The Working Group has resolved to specify a SPARQL/Update language, but may also pursue a HTTP based graph update via the protocol. This issue is orthogonal to the SPARQL/Update language. Whether or not there will be a concrete mapping between SPARQL/Update and HTTP based graph update is currently under discussion in the working group.

The current discussion seems to tend into a POST-only direction. We will discuss issues with this solution and outline potential solutions in the following. The underlying problem is that a design based on a single endpoint and interactions with that endpoint only, or on RESTful resources exposed through actionable URIs is not a mere interface design issue; it has implications on the data model and the data that a query operates on and returns as a response.

REST considerations become much more important in scenarios beyond read-only access, because a richer set of interactions with RDF services would require well-designed mappings of how to expose these services' RDF through actionable URIs, and how it is possible to *PUT*, *POST*, *GET*, and *DELETE* RDF data using HTTP methods and interaction with URIs that have a well-defined relationship with the RDF triples managed by the service.

Conclusion.

RESTful SPARQL is not just a design issue of the SPARQL protocol. For the SPARQL protocol to become RESTful, SPARQL itself must be able to support such a protocol; specifically, it must support interactions with URI-identified resources.

3. ISSUES

The most important step in REST is to identify the resources that services want to build their interactions around, so that they can then be exposed as hyperlinked resource representations. We hence start from the *information unit* concept and claim that this is something that always has to be done on a per-applications basis, however it can be defined in ways which make it easy to find those information units. This might be achieved simply by locating them with a SPARQL query, and then assigning identifiers to them, which can be used in an URI template mechanism.

What this does is essentially add a new level of RDF granularity; it introduces a concept which in many other scenarios is referred to as a *document*: A unit of data that is self-contained (but contains links to other resources) and can be interacted with as a whole. We claim that this level of granularity not only is required to support RESTful interactions in RDF-centered scenarios; it is also required when it comes to issues such as data assurance, authenticity, integrity, security, and provenance, where is becomes essential to be able to talk about datasets and be able to talk about them by using RESTful identifiers.

The *discovery of information units* has to be addressed [8], that is, a method is needed to find out which information units are available. Additionally, the identification process could also expose this information by using feeds, so that the list of identifiers would be serialized as a feed [10], which is a well-known and widely supported RESTful interaction pattern on today's Web.

Our main goal is to reuse as much of today's Web technologies and Web practices, and the biggest missing part we have identified is the fact that SPARQL in its current design lacks the identifiable units that have to be identified (logically as well as by URI) to build a RESTful protocol around them.

Based on the current state and the premises given above, we have identified three main issues:

- SPARQL's grounding in RDF makes it harder to think of more meaningful coarse-grained units that can be exposed and accessed in a RESTful way; in SQL and XML one has certain "grouping" features built into the metamodel (tables/rows or documents), but this not the case in RDF, natively.
- The HTTP binding itself (see Section 2.2 for the current bindings and the current direction of the discussion).
- Eventually, there is no identity on RDF triple level, which makes it hard to realize the information unit concept.

With these three problems in mind, we submit that transforming SPARQL into a RESTful protocol requires an effort to be more explicit on the RDF metamodel level to name both triples and information units (i.e., graphs). Identifying and naming entities that are important for interactions is the first and most fundamental activity in designing RESTful services.

Since SPARQL is based on the RDF metamodel, these fundamental questions of how RDF data is named and actionable on the Web level needs to be resolved on the RDF level, but we propose to first use an implementation approach that will put a naming layer on top of an RDF store in an attempt to expose the store's contents in ways more aligned with Web architecture.

4. TOWARDS A RESTFUL SOLUTION

4.1 Potential Solutions

[2]`http://www.w3.org/TR/sparql-features/#sparql-update`

To overcome the above discussed issues, we propose to use *named graphs* (NG)[3] to identify information units. For certain RDF serializations such as *RDFa*, there is an implicit NG available (the URI of the hosting HTML document is the NG), but this is not true for all serializations of RDF data. We will hence operate under the following assumptions:

- Every RDF triple is in a NG; also the default graph has a URI. It is up to the back-end to decide how to assign triples to NGs; this can be based on explicit assignments, rule-based assignments such as authorship or ownership, or algorithmic decisions such as explicit triples vs. inferred triples.
- NG URIs are under the control of the SPARQL service, so that the data made available at the URIs can be interacted with using HTTP methods.
- Serializations of NGs can use any representation that allows interactions with the data in the NG; these can be various RDF serialization formats, as well as non-RDF formats such as XML or JSON.
- The mapping between SPARQL and exposed serializations must be bijective to ensure a lossless transformation in both directions, or if there is some loss of information (such as losing sequence information when going from XML to RDF), it must be guaranteed that this information is not relevant for the RDF model.

In a first attempt, we have identified the following possible solutions for a RESTful mapping of SPARQL forms:

1. **Information unit-based:** One NG per information unit (say, a book, a person, etc.). This option is easy to implement, however the chosen granularity might be to coarse-grained, and hence inefficient for large-scale update operations. This option also depends on concepts to be explicitly identified and exposed, which might be appropriate in some scenarios, but might be too static and inflexible in others.
2. **Subject-level:** The subject resource is used as the HTTP resource. The advantage here is that there is no overhead concerning NG, however this (i) does not address RDF blank nodes, and (ii) it enforces an *all or nothing* operation on the subject level.
3. **Triple-level:** Every RDF triple is in a NG on its own (and can additionally be in an NG of an information unit it belongs to). This solution offers the advantage that all operations can be carried out effectively, however the overhead can be enormous for a huge number of triples.

4.2 An Early Demonstrator

To this end we have implemented a demonstrator using *Jersey*[4], an open source JAX-RS (JSR 311) reference implementation. We have implemented the first option (information unit-based solution) in a client/server setup and tested it with social network data represented in the *Friend-Of-A-Friend (FOAF)* vocabulary. See also Figure 2 for screenshots. The source code of our demonstrator is available online[5].

In the current version of the demonstrator we have implemented `SELECT`, `INSERT`, and `DELETE`. An exemplary mapping for a SPARQL `INSERT` is shown in listing 1. Based on the inserted data, a follow-up SPARQL `SELECT` and its RESTful mapping is given in the code fragment in listing 2.

```
INSERT INTO
 <http://localhost:8083/sparestfulql/default/erik> {
  <http://http://dret.net/netdret/foaf.rdf#me>
  <http://xmlns.com/foaf/0.1/knows>
  <http://sw-app.org/mic.xhtml#i> .
}

PUT http://localhost:8083/sparestfulql/default/erik

 <http://http://dret.net/netdret/foaf.rdf#me>
 <http://xmlns.com/foaf/0.1/knows>
 <http://sw-app.org/mic.xhtml#i> .
```

Listing 1: An exemplary RESTful SPARQL mapping for INSERT.

```
SELECT ?who ?whom
 FROM NAMED
 <http://localhost:8083/sparestfulql/default/erik>
 {
  ?who <http://xmlns.com/foaf/0.1/knows> ?whom .
 }

GET http://localhost:8083/sparestfulql/default/erik?
query=SELECT+%3fwho+%3fwhom+FROM+NAMED+%3c
http%3a%2f%2flocalhost%3a8083%2fsparestfulql%2f
default%2ferik%3e+%7b%0a+%3fwho+
%3chttp%3a%2f%2fxmlns.com%2f
foaf%2f0.1%2fknows%3e+%3fwhom+.
```

Listing 2: An exemplary RESTful SPARQL mapping for SELECT.

Though the experiments with our demonstrator are encouraging, they also highlight issues with this approach. For example, how should one deal a SPARQL `DELETE`? Obviously, the intention is to remove the referred triples, however, this contradicts with the HTTP semantics, which are about removing the entire resource itself.

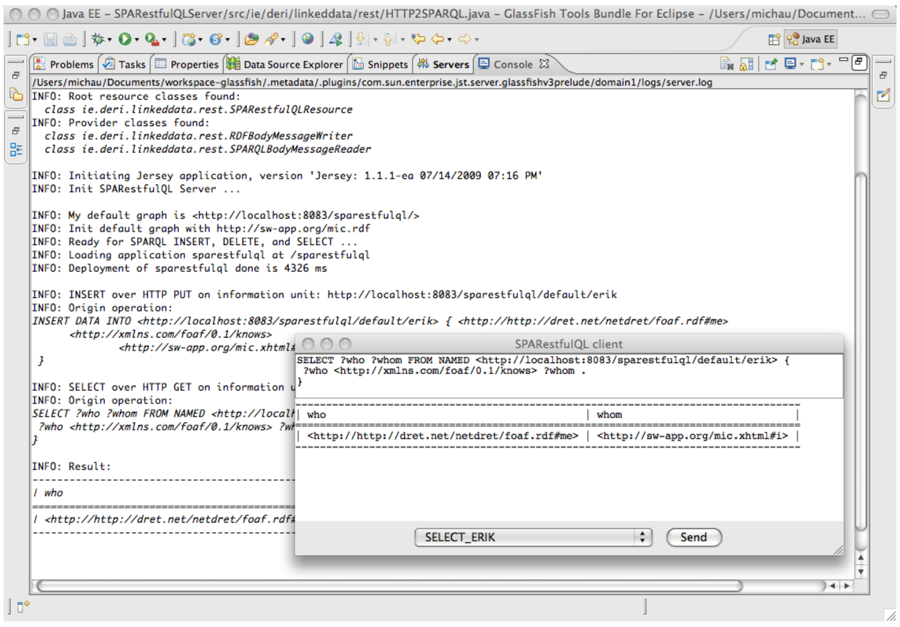
5. CONCLUSIONS

In this position paper we have discussed the RESTful approach and highlighted issues with SPARQL in this respect. We have indicated potential solutions and started to work on a demonstrator that allows us to experiment with potential solutions. We aimed to tackle the high level mismatch between a REST API and one based on a query/update language with its three dimensions: (i) resources, (ii) verbs, and (iii) state transfer, with a strong focus on the first one. Regarding verbs we note that one of the defining characteristics of the RESTful approach is that the verbs of the interaction are given by HTTP, that is, in particular *`GET`*, *`PUT`*, *`POST`*, and *`DELETE`*. This contrasts to the RPC-style,

[3]`http://www.w3.org/2004/03/trix/`

[4]`https://jersey.dev.java.net/`

[5]`http://bitbucket.org/mhausenblas/sparestfulql/`


Java EE – SPARestfulQLServer/src/ie/deri/linkeddata/rest/HTTP2SPARQL.java – GlassFish Tools Bundle For Eclipse – /Users/michau/Document...
Java EE
Problems
Tasks
Properties
Data Source Explorer
Snippets
Servers
Console
SPARestfulQL client
SELECT_ERIK
Send

```
/Users/michau/Documents/workspace-glassfish/.metadata/.plugins/com.sun.enterprise.jst.server.glassfishv3prelude/domain1/logs/server.log
INFO: Root resource classes found:
  class ie.deri.linkeddata.rest.SPARestfulQLResource
INFO: Provider classes found:
  class ie.deri.linkeddata.rest.RDFBodyMessageWriter
  class ie.deri.linkeddata.rest.SPARQLBodyMessageReader

INFO: Initiating Jersey application, version 'Jersey: 1.1.1-ea 07/14/2009 07:16 PM'
INFO: Init SPARestfulQL Server ...

INFO: My default graph is <http://localhost:8083/sparestfulql/>
INFO: Init default graph with http://sw-app.org/mic.rdf
INFO: Ready for SPARQL INSERT, DELETE, and SELECT ...
INFO: Loading application sparestfulql at /sparestfulql
INFO: Deployment of sparestfulql done is 4326 ms

INFO: INSERT over HTTP PUT on information unit: http://localhost:8083/sparestfulql/default/erik
INFO: Origin operation:
INSERT DATA INTO <http://localhost:8083/sparestfulql/default/erik> { <http://http://dret.net/netdret/foaf.rdf#me>
      <http://xmlns.com/foaf/0.1/knows>
              <http://sw-app.org/mic.xhtml#
 }

INFO: SELECT over HTTP GET on information u
INFO: Origin operation:
SELECT ?who ?whom FROM NAMED <http://localh
 ?who <http://xmlns.com/foaf/0.1/knows> ?wh
}

INFO: Result:
-------------------------------------------
| who
===========================================
| <http://http://dret.net/netdret/foaf.rdf#
-------------------------------------------
```

```
SELECT ?who ?whom FROM NAMED <http://localhost:8083/sparestfulql/default/erik> {
 ?who <http://xmlns.com/foaf/0.1/knows> ?whom .
}
```

```
-----------------------------------------------------------------------------------
| who                                           | whom                            |
===================================================================================
| <http://http://dret.net/netdret/foaf.rdf#me> | <http://sw-app.org/mic.xhtml#i> |
-----------------------------------------------------------------------------------
```

Figure 2: Screen-shot of the Demonstrator for information unit-based RESTful SPARQL mapping.

where the verbs are defined as methods by the developers and to query/update languages where the verbs are part of the query/update language. If these mismatches are not addressed, this leads to redundant information and ambiguous requests. Concerning the "state transfer": in REST, resource states a transferred between client and server, unlike the situation with the SPARQL protocol where queries/updates and responses are send between client and server. A RESTful change to an RDF repository might, for example, send a new NG as RDF/XML as a PUT request to some URI. These issues are subject to further research.

From the architecture perspective, the biggest issues around a more RESTful design of SPARQL are deeply intertwined with the way of how RDF handles identify and granularity of RDF data, and beyond our experimental implementation, we hope that our work informs and influences the design of SPARQL itself (making named graphs a more central part of the language design) and maybe even RDF (adding concepts such as named graphs to the basic metamodel of RDF).

6. REFERENCES

[1] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. SPARQL Protocol for RDF. World Wide Web Consortium, Recommendation REC-rdf-sparql-protocol-20080115, January 2008.

[2] Roy Thomas Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, May 2002.

[3] M. Hausenblas. Exploiting Linked Data to Build Web Applications. *IEEE Internet Computing*, 13(4):68–73, 2009.

[4] M. Hausenblas. Linked Data Applications. First Community Draft, DERI, 2009.

[5] Kjetil Kjernsmo and Alexandre Passant. SPARQL New Features and Rationale. World Wide Web Consortium, Working Draft WD-sparql-features-20090702, July 2009.

[6] Eric Prud'Hommeaux and Andy Seaborne. SPARQL Query Language for RDF. World Wide Web Consortium, Recommendation REC-rdf-sparql-query-20080115, January 2008.

[7] Leonard Richardson and Sam Ruby. *RESTful Web Services.* O'Reilly & Associates, Sebastopol, California, May 2007.

[8] J. Umbrich, M. Hausenblas, P. Archer, E. Hammer-Lahav, and E. Wilde. Discovering Resources on the Web. Technical Report, DERI, 2009.

[9] Erik Wilde. What are you talking about? In *Proceedings of the 2007 IEEE International Conference on Services Computing*, pages 256–261, Salt Lake City, Utah, July 2007.

[10] Erik Wilde. Feeds as Query Result Serializations. Technical Report 2009-030, School of Information, UC Berkeley, Berkeley, California, April 2009.