SWOWS and Dynamic Queries to build Browsing Applications on Linked Data

Paolo Bottoni  
Sapienza, University of Rome  
Department of Computer Science  
bottoni@di.uniroma1.it

Miguel Ceriani  
Sapienza, University of Rome  
Department of Computer Science  
ceriani@di.uniroma1.it

Abstract

The Linked Data Initiative is pushing dataset maintainers to publish data online in a highly reusable way through a set of open standards, such as RDF and SPARQL. The amount and variety of available structured data in the Web is increasing but its consumption is still quite limited. In particular applications used to explore linked data are mostly either generic linked data browsers or applications with hard-coded logic tailored for specific needs. SWOWS is a platform for declarative specification of applications consuming linked data. In this paper we describe the use of the platform for creating browsing applications tailored to specific contexts, and show how the declarative paradigm preserves flexibility of the application. To this end, the platform has been extended to allow the dynamic generation of SPARQL queries. An example of a linked data browser created with the platform is given.

1. Introduction

In recent years the Web is evolving from an interlinked set of documents to an interlinked web of data and services. The structured data available online is increasing both in quantity and diversity [7]. The comprehensive data model proposed by the World Wide Consortium (W3C) is the Resource Description Framework (RDF) [14]. Other de facto standards, e.g. the Freebase model [8], or HTML embedded formats, such as Microformats [25], Microdata [22] or RDFa [1], can be mapped to the Resource Description Framework (RDF) model [14] for interoperability (for example using Any23 1).

One of the key advantages of the linked data model is that it allows the serendipitous exploration and reuse of existing data. Potentially, any expert of a specific domain can build a fully customized visualization from a set of possibly different linked data sources.

In practice, building such a visualization currently requires advanced programming skills. Moreover it involves a number of system choices that constrain the effective reuse of a visualization. To mitigate this problem, we have developed the Semantic Web Open datatafloW System (SWOWS) [11, 10], a platform that allows the declarative construction of interactive linked data applications. Applications are built from a basic set of operators (based on SPARQL, the standard query language over the RDF model [21]) adopting the pipeline metaphor.

In this paper we discuss the use of SWOWS to build applications to browse specific sets of linked data. We show how such applications can be built based purely on Web and Semantic Web standard technologies and how the declarative approach can help in organizing flexible visualizations.

The SWOWS platform has been extended to allow manipulation of SPARQL queries as data. A developer using SWOWS can thus write SPARQL queries which take as input other SPARQL queries and/or produce new queries in output. The queries written with SWOWS have the role higher functions have in other programming languages (Scala [30] being possibly the most popular nowadays). The use of this functionality (together with other extensions to the original SWOWS platform) will be showcased through an example application.

In the rest of the paper, Sect. 2 introduces the technology background and Sect. 3 discusses related work, while Sect. 4 specifically describes SWOWS. New features are shown in Sections. 5 and 6, where dynamic query generation is illustrated. An example application is described in Sect. 7 and Sect. 8 discusses conclusions and future work.

2. Technologic Background

The relational model is widely used to represent virtually any kind of structured information. The Resource Description Framework (RDF) [14] generalises it to the universe of structured data in the World Wide Web, better known as the Semantic Web [5]. In the RDF data model, knowledge is represented via RDF statements about resources, where a
resource is an abstraction of any piece of information about some domain. A RDF statement is represented by a RDF triple, composed of subject (a resource), predicate (specified by a resource as well) and object (a resource or a literal, i.e. a value from a basic type). A RDF graph is therefore a set of RDF triples. Resources are uniquely identified by a Uniform Resource Identifier (URI) [4], or by a local (to the RDF graph) identifier if they have not meaning outside of the local context (in which case they are called blank nodes). The resources used to specify predicates are called properties. A resource may have one or more types, specified by the predefined property rdf:type. A RDF dataset is a set of graphs, each associated with a different name (a URI), plus a default graph without a name. We use RDF through the framework to represent any kind of information and its transformations. In RDF, prefixes can be used in place of the initial part of a URI, representing specific namespaces for vocabularies or set of resources.

We extensively use SPARQL2 [21], the standard query language for RDF datasets. SPARQL has a relational algebra semantics, analogous to those of traditional relational languages, such as Structured Query Language (SQL). The SPARQL CONSTRUCT, one of the SPARQL query forms, takes as input a RDF dataset and produces a RDF graph. While the SPARQL Query Language is “read-only”, the SPARQL Update Language [33] defines a way to perform updates on a Graph Store, the “modifiable” version of a RDF Dataset. A SPARQL Update request is composed of a number of operations. The current version of the standard is SPARQL 1.1, but much of the existing work refers to the previous version, SPARQL 1.0 [32]. SPARQL 1.1 algebra offers an expanded set of operators, effectively allowing the expression of queries that were not expressible before.

The ubiquity of Web browsers and Web document formats across a range of platforms and devices drives developers to build applications on the Web and its standards. Requirements for browsers have dramatically changed from the first days of the Web. Now a browser is an interface to an application to build applications on the Web and its standards. Indeed, ECMAScript libraries for interactive applications via special functions and syntaxes, but the execution model remains request oriented.

3. Related Work

Several languages were proposed to define SPARQL views in a way analogous to SQL views. A SPARQL view is a graph intensionally defined by a SPARQL CONSTRUCT query; the input dataset can be composed by both “real” (extensionally defined) graphs and views. RVL [28] is an early effort, using an imperative language for defining views based on an independently defined query language (RQL [24]). vSPARQL [35] is an extension of SPARQL 1.0 grammar allowing named views defined with CONSTRUCT queries and reusable in other queries. Schenk and Staab, working on Networked Graphs [34], propose an RDF-based syntax to define views, which are graphs defined in terms of SPARQL 1.0 CONSTRUCT queries on explicitly defined graphs and other views. Although powerful enough to define read-only applications (possibly together with visualization tools described below), network of views do not easily model interactive applications. In particular, they face the problem of how to represent events and time-dependent information, including the application state.

Two pipeline languages have been proposed to define RDF transformations, namely DERI Pipes [27] and SPARQLMotion [26]. They offer a set of basic operators on RDF graphs to build the pipelines and they are both endowed with a graphical environment to create the pipelines using the available operators (free in the case of DERI Pipes, in a commercial software for SPARQLMotion [13]). In SPARQLMotion, both pipelines and queries are represented in RDF (for queries using the SPIN-SPARQL [18] syntax).

Visualbox [20] and Callimachus [2] have been proposed for linked data visualisation. In their two-step model/view approach, SPARQL queries select data and a template language generates the (XML-based) visualisation. SPARQL Web Pages (SWP) is a RDF-based framework (to be used with SPARQL Motion or on its own) to describe and render HTML+SVG visualisations of linked data. HTML and SVG are mapped to two corresponding vocabularies and together with the UISPIN Core Vocabulary allow the association of a RDF resource with the description of its visualisation. The description may be also statically associated with a class of resources, with each specific resource mapping defined through a SPARQL query. In all these proposals the execution model corresponds to managing a single HTTP request, as with typical application server technologies like Java Servlet or PHP. Persistence and logical relationships between requests and client state must be managed explicitly (e.g. saving/loading data related to a session and encoding parameters in requests)3.

Generation/manipulation of queries at runtime is widely

2 Originally a recursive acronym SPARQL Protocol and RDF Query Language, the extended form has then been dropped from W3C documents.

3 Commonly called JavaScript, the dialect from Mozilla Foundation.
used both in SQL and in SPARQL. Several systems (for a SPARQL example, see Jena [29]) provide basic support through parameterized queries, in which some parameters are bound (through some external mechanism) to actual (scalar) values at execution time. This mechanism is not sufficient when the structure of the query must be changed dynamically (e.g., for multiple or complex search criteria and/or ordering rules). Generic dynamic query generation is usually achieved through string-based, semantically-unaware, manipulation or through a programmatic interface offered by the host language. There have been efforts to represent queries using semantically rich structures not tied to a specific host language [37], specifically for SPARQL the already mentioned SPIN-SPARQL [18] vocabulary. While this vocabulary potentially allows dynamic query generation, its use for this purpose is documented only for a specific case of query rewriting [17].

4. SWOWS

We set the following requirements for SWOWS: to be based on a dataflow language in which data transformations are represented as pipelines; to use pipelines through cascading declarative views on the input or other views; to represent data as RDF; to be able to connect to existing RDF sources; to exhibit interactivity through Web interface input/output; to represent pipelines as RDF to share and reuse; to support interoperability with XML; to use existing standards whenever possible.

The platform we propose allows users to define linked data applications through a pipeline language based on RDF. Users create pipelines using a visual representation, through a Web-based editor, in turn interacting with a pipeline repository. Other software, called the dataflow engine, executes the pipeline (after downloading the corresponding RDF Graph from the repository) on a possibly separate server, with a Web-based interface as well. Figure 1 shows a simplified lifecycle of the pipeline, from editing and saving it in a directly controlled repository, to eventually sharing it for use “as it is” or reuse in other pipelines.

A SWOWS pipeline is a side-effect-free dataflow programming module, taking as input an RDF Dataset and returning another RDF Dataset. The available components (shown in Figure 2) are: the default input graph and the (named) input graphs; the default output graph and the (named) output graphs; the transform processors, that execute a SPARQL 1.1 [21] query against a RDF dataset; the single graph stores, whose content is incrementally modified during an execution of the pipeline by executing a SPARQL 1.1 Update [33] on it each time one of its input graphs changes (the update takes as input a RDF dataset composed by the previous snapshot of the store as default graph and a set of input named graphs); existing pipelines which can be used as components in the current pipeline; file data sources, i.e. RDF graphs generated by loading local or remote files (serialized in one of the standard RDF formats); another way to access data from outside the pipeline is through SPARQL Federated Queries used in transform processors or simple graph stores.

Figure 1. A schematic view of the platform

Figure 2. The pipeline base component types

A pipeline can be designed just for reuse by other pipelines. If a pipeline has to be executed (i.e. it is a top level pipeline), its default output graph must comply with an XML DOM Ontology in RDF. It will represent a HTML or SVG document, to be rendered by the user interface. Its default input graph will receive the DOM Events generated in the interface, described with a DOM Events Ontology.

The main blocks of the implemented application are the editor, the pipeline repository and the dataflow engine. The editor is a rich Web application with its client side logic coded in HTML+CSS+JavaScript and embedded

5SPIN-SPARQL vocabulary is widely used as a way to attach SPARQL rules to RDF resources.

6http://www.swows.org/2013/07/xml-dom

7http://www.swows.org/2013/07/xml-dom-events
in the Callimachus Web application [2], used as pipeline repository. The dataflow engine is a Java-based (using Apache Jena [29]) Web application maintaining the state of each running pipeline instance; when a new instance is launched (e.g., from the editor) the engine initialises the pipeline and returns its output to the client, along with a piece of JavaScript logic to report handled events back to the server; each time an event is fired on the client, the dataflow engine is notified and answers with the changes to be executed on the client content. On the client side, any browser supporting JavaScript can use both the editor and the generated application. The software is freely available\(^8\).

5. Transform Processors

In the previous version of the platform the behaviour of a transform processor was always specified through a CONSTRUCT query. It has been changed to allow also for a SELECT query or a UPDATE request, maintaining the fact that the transform processor executes a stateless operation on a RDF dataset, resulting in a single RDF graph.

The SELECT queries are run according to the corresponding semantics, but the result set is represented as a RDF graph, containing the bindings, their order and each value associated with a binding. This allows the derivation of some order (depending on the ORDER BY clause) over a set of items. Ordering could also be achieved through CONSTRUCT queries but in a contrived and potentially inefficient way.

The UPDATE requests are run considering the input dataset as a Graph Store; the output is given by the content of the default graph after the UPDATE (the changes to the other graphs are discarded). The UPDATE is useful if most of the input default graph must be copied to the output, as using UPDATE only the changes need to be expressed.

6. Dynamic Queries

One of the advantages of a declarative platform based on a flexible model such as RDF is to easily generalise functions. For example if one wants to generalise an operation of filtering or aggregation on some property, one can take the property from a configuration graph or as result of another query: properties in RDF are resources and can be treated as such. In practice, due to some limitations in SPARQL expressivity (e.g., there is no direct way to express a FORALL operator), it can be tricky to build generalised queries; moreover, due to current limits in SPARQL engines and protocol infrastructure, they can be highly inefficient. For these reasons we propose dynamic query generation, using a RDF vocabulary to represent any SPARQL query. The vocabulary

![http://www.swows.org/](http://www.swows.org/)

\(^8\)The usual approach for faceted navigation is to use the different categories to build multi taxonomy filters for searching items. This approach is indeed established and effective; we want to experiment instead with the use of different dimensions to guide the user in free exploration of dataset item by item. This kind of navigation can be useful when the user has no previous complete knowledge of the used taxonomies and/or the user interface will be used for casual exploration, wandering through a data set. Usually this is provided through related items links, that provide a single dimension for this kind of exploration.\(^9\)

The usual approach for faceted navigation is to use the different categories to build multi taxonomy filters for searching items. This approach is indeed established and effective; we want to experiment instead with the use of different dimensions to guide the user in free exploration of dataset item by item. This kind of navigation can be useful when the user has no previous complete knowledge of the used taxonomies and/or the user interface will be used for casual exploration, wandering through a data set. Usually this is provided through related items links, that provide a single dimension for this kind of exploration.\(^9\)

The user interface for the application is composed of: a main area in which an image is shown together with information about the currently selected item; a variable number of related artworks areas, in which links to related artworks

\(^9\)In the present work we present the idea of multifaceted visual exploration just to show the feasibility of the proposed approach for design of browsing applications. We think that the idea on its own deserves to be studied and tested in future research.
(visually represented as thumbnails) are shown according to different axes (in Figure 3 one of the possible configurations of the application is shown\(^\text{10}\)). The pipeline design has been based on the separation between presentation elements (position and sizes of the different areas) and content elements (different filters used for related artworks). A specific graph defines the association between the two kinds of elements.

For this example we consider the Freebase dataset to be available from a SPARQL Endpoint, considering the standard mapping defined by Freebase maintainers. It is possible to use the public FactForce endpoint\(^\text{11}\) (aggregating also several other datasources [6]), or to install a cloud-based distribution as the Sindice Freebase Distribution\(^\text{12}\) for a more reliable access.

The pipeline, shown in Fig. 4, is composed by the following components: some Data Sources corresponding to queries: one for each available filter (here four, but could be any number) and one, filters-common.rq, to be used as template to dynamically build the query; Framing that represents the wireframe of the user interface, both as SVG structure and as specification of the areas for the other components; FiltersQuery that generates the dynamic query for the areas based on AreaFilter, the association between areas and filters; SelArtwork, that stores the selected artwork, setting the initial one and changing it when the thumbnail for another one is clicked (events coming from Default Input); RelatedArtwork, that runs the dynamic query to obtain the sets of related artworks; View, that creates the dynamic part of the Default Output (the SVG) to be merged with the static part of the visualization from Framing.

The “heart” of the pipeline is the generation and execution of the query that retrieves the data of related artworks for each area based on the defined areas, the defined filters (the queries) and the association between them. The areas are defined like the following (Turtle syntax):

\begin{verbatim}
SELECT DISTINCT ?relatedArtwork WHERE {
    ?selectedArtwork aw:artist ?artist;
    aw:artist_genre ?genre;
    ?relatedArtwork aw:artist ?diffArtist;
    aw:artist_genre ?genre;
    aw:art_form ?diffForm .
    FILTER (?diffForm != ?form).
    FILTER (?diffArtist != ?artist).
    FILTER (?diffForm != ?form).
}
\end{verbatim}

Any kind of SELECT query will be accepted as long as it projects the variable ?relatedArtwork and use the variable ?selectedArtwork corresponding to the selected artwork.

Finally the association between areas and filters is defined through a graph like the following one:

\begin{verbatim}
PREFIX spx: <http://sindicetech.com/freebase>
sps: resultVariable var:area, var:relatedArtwork;
sp: where (@<#FilterUnion>)
\end{verbatim}

The FiltersQuery component generates the dynamic query and has as inputs various named graphs: <#filters>, from the file data sources corresponding to the queries defining the filters (the RDF graphs, being connected to the same input, are merged, but each query is still distinguishable, as each query root element corresponds to the original query URI); <#common>, from a file data source corresponding to <filter-common.rq>, a query used to hold constraints that should be satisfied by any filter (e.g. having at least an image and a minimum set of information); <#areas>, from AreaFilters component, giving the association between areas and filters described above; <#framing>, from Framing component, holding the definition of the areas, also described above. The default graph is given by the union of <#common> and <#filters>. FiltersQuery is defined an Update request; it is useful to recall that the output graph is obtained from the default input graph, applying on it the Update request. The Update request is the following (prefix declaration omitted):

\begin{verbatim}
INSERT {<#FilterUnion>}
\end{verbatim}

\(\text{10}\)Some images are not shown due to broken URLs in the dataset.

\(\text{11}\)http://factforge.net/sparql

\(\text{12}\)http://sindicetech.com/freebase
The Update request is composed by two separate Update operations. The first one (lines 1–13) is constant (in fact the \texttt{WHERE} clause is empty) and builds the fixed part of the query to be generated. The second Update request depends from the association between areas and filters (line 38) to build the part of the query corresponding to each filter and consisting of a common part (lines 16–21) and having as subquery the query corresponding to the filter (lines 22–24). That filter query is also enriched in some ways: the \texttt{filter-common.rq} query is added as a subquery (lines 25–26); a \texttt{LIMIT} clause (\texttt{sp:limit}) is added to retrieve exactly the number of items that will be shown in each area (lines 34–37, 39, 31); the corresponding area is added to the variables projected by each filter query (lines 28, 19); the queries are joined to the selected artwork through projecting the variable for each query (line 31) and joining each query with a constant part that reads the selected artwork from the input of the generated query (lines 8–11, 19).

As the output of this component is connected to the input named \texttt{#query} of \texttt{RelatedArtwork}, this query is executed in that component, taking as input the selected artwork. The purpose of using a \texttt{SELECT} query is having the items ordered. Even in the case the order of the related artworks in an area is undefined (the tuples are ordered only with respect to \texttt{?area}), the \texttt{SELECT} result gives an arbitrary order that can be used to draw the corresponding thumbnails. Finally, the \texttt{CONSTRUCT} query in the component \texttt{View} (not shown), builds the SVG page based on an image and information on the selected artwork and the thumbnails of related artworks organized by areas.

8. Conclusions and Future Work

We discussed the use of the platform SWOWS and dynamic query generation to build interactive linked data applications, especially in the case of flexible browsing applications. Dynamic query generation and other extensions to the platform as originally conceived were presented, justified by application to concrete cases.

We want to keep experimenting in this direction, and possibly also to leverage this experimentation to build...
higher level interfaces, designed also for usage by non-expert users, as a way to flexibly interact with linked data.

References


