SparqPlug: Generating Linked Data from Legacy HTML, SPARQL and the DOM

Peter Coetzee *
Department of Computing,
Imperial College London
180 Queen's Gate, London,
SW7 2AZ, United Kingdom
plc06@doc.ic.ac.uk

Tom Heath †
Talis Information Ltd
Knights Court, Solihull
Parkway, Birmingham
Business Park, B37 7YB,
United Kingdom
tom.heath@talis.com

Enrico Motta
Knowledge Media Institute,
The Open University
Walton Hall, Milton Keynes,
MK7 6AA, United Kingdom
e.motta@open.ac.uk

ABSTRACT

The availability of linked RDF data remains a significant barrier to the realisation of a Semantic Web. In this paper we present SparqPlug, an approach that uses the SPARQL query language and the HTML Document Object Model to convert legacy HTML data sets into RDF. This framework improves upon existing approaches in a number of ways. For example, it allows the DOM to be queried using the full flexibility of SPARQL and makes converted data automatically available in the Semantic Web. We outline the process of batch conversion to RDF using SparqPlug and illustrate this with a case study. The paper concludes with an examination of factors affecting SparqPlug's performance across various forms of HTML data.

Categories and Subject Descriptors

H.3.1 [**Information Storage And Retrieval**]: Content Analysis and Indexing

Keywords

RDF, SPARQL, DOM, HTML, Conversion, Linked Data, Semantic Web, Data Mining

1. INTRODUCTION

One of the ongoing challenges facing the Semantic Web is not a technical one, but rather a human one; many data providers will not be motivated to make their data available on the Web in formats that are useful to Semantic Web clients (i.e. RDF) until such clients are widely deployed and used. Similarly, these clients are unlikely to become popular until sufficient instance data is available to offer compelling utility value.

For this reason, many Semantic Web-aware developers have an interest in converting existing non-RDF datasets into RDF models for publication on the Web. A noteworthy example of such an effort is the DBpedia Project [1], which has the aim of making existing template-structured data within Wikipedia [2] available on the Web as crawlable Linked Data, and queryable via a SPARQL[3] endpoint. The size of Wikipedia justifies a large-scale custom solution for this conversion, however many smaller datasets will not warrant such an approach despite the value of their data.

To address these issues we have developed SparqPlug, a generic framework for converting legacy HTML-only datasets into RDF, and making this data available on the Web. The SparqPlug approach is targeted at more sophisticated Semantic Web practitioners, and as such requires a fair degree of domain-specific knowledge to be useful. One goal of the approach is to enable the reuse of existing skills that one may expect developers in the Semantic Web field to hold, such as writing SPARQL[3] queries.

Two key processes underlie the SparqPlug approach: parsing the DOM of an HTML document and converting this to RDF, then querying the resulting graph using SPARQL to extract the desired data. A number of existing techniques attempt to extract RDF from legacy HTML data sources. In the following section we will review this existing work, before examining the SparqPlug approach in detail. Later sections of the paper present a case study, using SparqPlug to RDFize example datasets, and a brief evaluation of the system's performance.

2. RELATED WORK

2.1 DOM, XSLT, and GRDDL

The Document Object Model has long been the favored means amongst developers for accessing the parse tree of HTML and XML. It provides methods for getting the attributes and children of nodes, making processing of the tree less difficult and cumbersome than manual parsing. However, it is in general too complicated a method for converting a given document tree into another. For this, XML stylesheets and transformations (in the form of XSLT) are the accepted standard, as embodied in the GRDDL approach [6]. While this is significantly easier to use, it is by no means perfect. One of its greatest limitations is the extent to which it requires perfect lexical representation: the slightest syntactic bug or inconsistency in the source document, or even syntactically correct but misused XSLT tags, and the entire operation will fail. It is challenging to craft a transformation for a given set of source documents, and can

*This work was conducted while the author was a member of KMi, The Open University.

†This work was conducted while the author was a member of KMi, The Open University.

LDOW2008, April 22, 2008, Beijing, China.

even be impossible to convert regular HTML with an XSLT without first serializing it as valid XHTML.

The SparqPlug approach also requires the use of valid syntax; but should make it far simpler to successfully query the DOM. Part of its operation involves the parsing of the HTML DOM through a Tidy-esque system to balance tags and validate the model. This at least adresses one of the issues outlined with XSLT; why not just serialise this DOM to valid XHTML, and continue accordingly? The shortcomings of XSLT have given rise to XQuery as an alternate means of extracting data from the DOM. Some have even attempted to convert SPARQL queries into a valid XSLT / XQuery syntax, to query the DOM directly [7]. While these type of approaches are very simple, and generally rely on standard tools, they lack the full expressivity of the SPARQL language. Integrally, the standards they rely on ignore some key implementation specific extensions (as suggested, albeit not required, in the SPARQL spec), such as ARQ's[8] Property Functions[9].

An alternate approach, SPAT[10], involves extending SPARQL's triple pattern syntax to include a mapping (defined in-query) between a given RDF predicate, and an XPath expression, in order to facilitate conversion of both RDF to XML and XML to RDF. One major issue with this latter approach is that of URI minting; it does not provide a simple way of minting URIs for the document, instead relying on static URIs, or blank nodes on which to root its statements.

The goal of the SparqPlug system is to facilitate the conversion of DOM documents, in particular poorly constructed HTML, into linked RDF data in as *simple* and *automated* a manner as possible.

2.2 Data Mining & NLP Techniques

Researchers such as Welty [11] apply Data Mining techniques from the Artificial Intelligence field to the problem, with varying degrees of success. Given the general lack of provenance and trust awareness in many Semantic Web browsers today, the inherent inaccuracy of this method can be a major issue; slightly inaccurate assertions can be more of a hindrance to machine understanding than a help.

2.3 Thresher

The Thresher [12] system aims to make the process of conversion into RDF as easy as possible, so that even the "Average Joe" user can use it. As a result (by the authors' own admission), it may not be as expressive as other methods. It invites a user to markup a page, extrapolating a pattern in the HTML by analyzing the distance between DOM edit trees. It provides its output to a user through the Haystack semantic web interface. All data produced is held locally in an RDF 'silo'[1].

2.4 MindSwap Web Scraper

Web Scraper [13], from MindSwap, is perhaps the system closest in approach to SparqPlug. It allows users to define the structure of an HTML List or Table in a desktop Java application, and parse these into RDF through their Ontology Browser. It is more expressive than Thresher, but will be much harder (or even impossible) to use with slightly less formally structured data. It requires that its users have much the same level of knowledge about semantic web technologies as SparqPlug. Its output is held locally as an RDF silo.

2.5 Piggy Bank

SIMILIE's Piggy Bank [14] semantic web browser has a Javascript-based plugin system for scrapers. It encourages interested parties to develop and host their HTML scrapers in Javascript, providing the RDF back to Piggy Bank. It is probably one of the most expressive of the RDFizers here, as it requires developers to write code to parse HTML pages. However, a new Javascript application must be written for each site it is applied to. Piggy Bank also requires that the developer have knowledge of Javascript as well as related Semantic Web technologies, effectively closing the door to a large part of the community. All of the screen scrapers' output is held in a local silo, preventing links from this RDF to other data sources.

2.6 Sponger & Triplr

Virtuoso Sponger [15] and Beckett's Triplr [16] work on a similar 'RDF Middleware' style approach. Triplr seeks out GRDDL [6] and RSS data from a given (X)HTML page and converts it into RDF/XML, N-Triples, JSON etc. Virtuoso takes this a step further; given a URI, it will first attempt to dereference it to obtain RDF (through content negotiation with an `accept:` header, as well as following HTML `link` elements). Should this fail, it will try to convert the data source using microformats [17], RDFa [18], eRDF [19], HTML's own metadata tags, the HTTP headers from the server, and even APIs such as Google Base [20], Flickr [21] and del.icio.us [22]. The obvious limitation of such middleware services is that they require that the source data be already marked up for conversion into a machine understandable format, or have hard-coded access methods (in the case of the Sponger's API extraction).

2.7 Round-Up

One of the great disadvantages common to the existing techniques is that of duplication of effort. Often a single user will use the application locally, creating their own personal RDF silo. If another user wishes to convert the same source, they must go through the process (however simple or complex it may be) all over again. Furthermore, the data which is produced is, by its local nature, not necessarily 'good linked data', according to the accepted Linking Open Data community best practices [23].

3. SPARQPLUG

SparqPlug aims to ease the creation of RDF data from legacy HTML-only data sources, and make the resulting data available on the (Semantic) Web. This is achieved by treating the DOM as just another data source to be queried using the SPARQL query language. As the system is targeted at creation of RDF models for exposure on the Semantic Web, it is important that it is accessible to people familiar with these technologies. As such, its design incorporates the SPARQL query language [3] as its means of deriving RDF models from the source data. It is important to note that SparqPlug aims to facilitate this conversion across a particular subset of the process; its area of specialty is the actual conversion of an HTML document into RDF. Previous authors have already researched techniques for ontology selection (for example in Protégé 2000[4]), and UI based extraction (e.g. Huynh's Sifter[5]). SparqPlug would benefit from integration with such technologies to further simplify its use.

[1]The term RDF 'silo', generally refers to a collection of RDF instance data which is not linked to any other data sources.

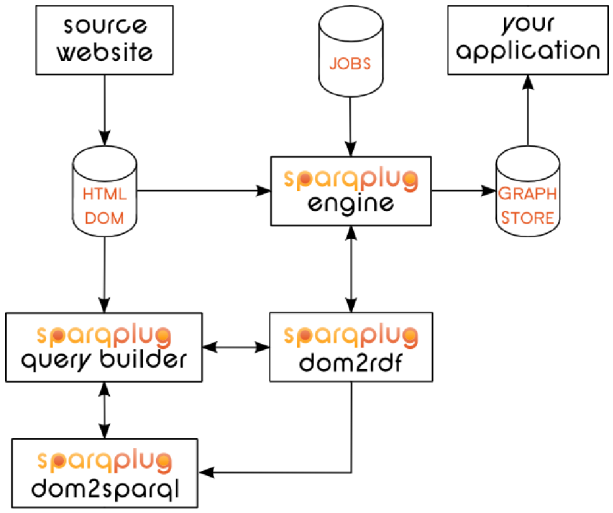

Figure 1: SparqPlug System Architecture

The workflow for conversion of a document is as follows;

1. Get source HTML Document
2. Parse HTML into the Document Object Model in Memory
3. Convert DOM to an RDF Model
4. Query RDF model with a SPARQL CONSTRUCT query
5. Store output model in persistent storage

SparqPlug will be provided as free downloadable software to run on a local machine, and as a web service online (at `<http://sparqplug.rdfize.com/>`), where users may queue up jobs for execution. The major advantage offered by the SparqPlug online service is that data can immediately be made available on the Semantic Web and interlinked with other data sets from the point of conversion.

3.1 SparqPlug Architecture

SparqPlug consists of four main components (Fig. 1):

1. SparqPlug DOM2RDF
2. SparqPlug DOM2SPARQL
3. SparqPlug Query Builder
4. SparqPlug Service Engine

3.2 SparqPlug DOM2RDF

In many respects, this is the heart of the SparqPlug system. It is responsible for getting regular real-world HTML and transforming it into an RDF model in the SparqPlug dom: (`<http://sparqplug.rdfize.com/ns/dom/>`) namespace. It makes use of the following properties when doing this; note that many of these are taken directly from the DOM itself, and share the same semantics. Others are added by SparqPlug to make it easier to query the model. SparqPlug also incorporates dynamic namespaces for tags: (`<http://sparqplug.rdfize.com/ns/tags/>`) and attr: (`<http://sparqplug.rdfize.com/ns/attr/>`). These are for the node names and attribute names respectively.

1. `dom:nodeName` The DOM Name of the current node, e.g. `tags:li` or `tags:body`
2. `dom:nodeValue` The value of this node. For example, a fragment such as `<div>Hello World</div>` has nodeValue "Hello World"
3. `dom:hasAttributes` The object of this property is a BlankNode describing the attributes applied to this node, in the attr: namespace.
4. `dom:subNode` An immediate child node of this one in the parse tree
5. `dom:nodeIndex` An identifier which is unique for this node in the entire model
6. `dom:subNodeNo` An identifier which is unique amongst siblings of a single parent, a convenience property
7. `dom:nextSibling` Another convenience property, this 'points' to the sibling to the 'right' of the current node in the parse tree

The conversion process recursively walks the DOM, adding statements for each node under these properties. This model itself is unsuitable for serving as linked data, as (by necessity) it makes extensive use of BlankNodes, and is, in addition, of a structure poorly related to its content data. Each node in the original DOM becomes a BlankNode in the output graph.

3.3 SparqPlug DOM2SPARQL

This component of SparqPlug complements the DOM2RDF processor. It takes a fragment of HTML as input and follows a similar conversion procedure as that used to convert the DOM to RDF. However, instead of using the fragment to create an RDF graph, DOM2SPARQL uses it to create a SPARQL query for extracting the target data from the RDF graph generated by DOM2RDF. (This is often a useful starting point for a user to build their own queries; it may not always create an optimal query, and its results may not be complete - but it is designed to be significantly easier than starting from scratch.)

Integral to DOM2SPARQL is the ability to insert the variables you wish to use to construct your RDF within the HTML fragment, as well as inserting required literal values. For example, the fragment `<a href="?someLink">"Previous Page"</a>` will produce a SPARQL SELECT query which gets the href of any hyperlink with the text *Previous Page*. If the quotation marks were left off, it would have matched any hyperlink.

This conversion process from HTML fragment to SPARQL query is far from trivial; it uses a set of heuristics to produce what aims to be a *minimal* and *correct* SPARQL query for the input fragment. To do this, it first converts the fragment into a full RDF model using the DOM2RDF component previously described; in this way all normalisations and tidyings that are applied to the input document are also incorporated. This RDF model is then heuristically trimmed and simplified, node by node. Any triples which

are deemed to be unnecessary constraints are removed. Some of the heuristics which are applied include:

1. Remove non-constraining values from nodes; node values describing a SPARQL variable, starting in '?', or enclosed in double-quotes (a SPARQL literal constraint) are kept, and others are removed
2. Remove nodes entirely which have no children, no value constraints, and no attributes.
3. Remove `dom:hasAttributes` predicates from nodes where the number of attributes is zero.
4. Remove `dom:subNodeNo` predicates from nodes which have no siblings in the model.
5. Remove `dom:subNode` constraints for children which do not exist (e.g. if the child was a non-constraining node and was removed).
6. Ensure, to the greatest possible extent, that nodes in the query follow the order in which they will be encountered in the RDF model; through experimentation it was determined that this can afford a 20x increase in execution speed. This ordering was achieved by ensuring that the basic graph patterns in the query are created as their relevant portions of RDFized HTML are processed.

As SPARQL is a declarative pattern-matching query language, the more constraints that can be removed, the faster the queries will run. For this reason, DOM2SPARQL may occasionally be overzealous in removing constraints – it is anticipated that users will be more likely to add these constraints back in, increasing the accuracy of their query, than to remove large numbers of unnecessary ones.

3.4 SparqPlug Query Builder

The Query Builder binds together the features of the DOM2SPARQL and DOM2RDF components to allow a user to prototype their queries, before designing a job for the service to work on. This is usually the best starting point for working with and getting the most out of the SparqPlug web service. It will allow a user to test and optimise their queries before submitting them for processing. It also generates and facilitates testing of a boilerplate query to extract links from an HTML page.

3.5 SparqPlug Service Engine

The final part of SparqPlug is its engine. This does most of the hard work in the system, gluing together all of the components into a usable web service. Once the user has worked out her queries, and found the data she wishes to RDFize, she may setup a new job for SparqPlug. It is the engine's responsibility to create and manage these jobs, and ensure they are processed quickly and fairly. Users may view the current job queue at any time, or view the status of their job; this will tell them how much work the engine has done already, how much it knows it still has to do, or indeed whether or not the job is complete.

SparqPlug requires a few pieces of information from a user in order to process a job. The most important of these are the prototypical *extraction* query, and the *link* query. The former of these is used to `CONSTRUCT` an RDF graph from the input document. No other SPARQL query type is allowed for this field. SparqPlug can also apply the same extraction query to a set of HTML documents, specified using the link SPARQL query (`SELECT` or `SELECT DISTINCT`). This is also run over the input document, and any variables which it `SELECT`s are added to the job's work queue.

The only other pieces of information presently required are a seed URL, a job name, and a regular expression to transform the page URL into the graph name's local-part. Named graphs are given names in the namespace `<http://sparqplug.rdfize.com/jobs/<YourJobName>/graphs/<GraphName>>`, and will be dereferencable at that URI. Group 1 from the regular expression is taken as the local part (`<GraphName>`) of the URI.

3.6 The End Result

As a job finishes each unit of work it must do, it commits the results to the graph store, as a collection of named graphs for this job. The job will have a URI, such as `<http://sparqplug.rdfize.com/jobs/MyJob>`, and all of the graphs belonging to this job will exist in the job's own graphs namespace, `<http://sparqplug.rdfize.com/jobs/MyJob/graphs/>`. The organisation of these graphs are up to the user's Regular Expression query. If two pages are RDFized and result in having the same named graph, the output will be the union of the `CONSTRUCT`ed models.

3.6.1 Linked Data Compliance

Wherever possible, SparqPlug will serve up RDF data according to the Linking Open Data Community's accepted best practices. URIs should be dereferencable, with content negotiation handled by 303-redirects. At the present time, SparqPlug supports content negotiation over the MIME types for HTML, RDF/XML, and N3.

If a user-agent requests the URI of the job, it will be served a description of that job, including details of any graphs it contains. Dereferencing the `/graphs` namespace of a job will return the union-graph of all the named graphs in the job's dataset. Finally, one may dereference Things within a job's graphs, either as `<http://sparqplug.rdfize.com/jobs/MyJob/graphs/SomeGraph/things/SomeThing>`, to get the information about `SomeThing` in a specified named graph (`SomeGraph`), or `<http://sparqplug.rdfize.com/jobs/MyJob/things/SomeThing>`, to get the information about SomeThing in all graphs in a dataset. Note that the SomeThing which is described is the URI resource minted by the SparqPlug job as `<http://sparqplug.rdfize.com/jobs/MyJob/things/SomeThing>`; the addition of `/graphs/SomeGraph/things/SomeThing` is merely an added convenience. These RDF descriptions are simply obtained by means of a SPARQL DESCRIBE query over the dataset – without any prior knowledge of the data to be put in the store, it is not possible to provide any more intelligent descriptions.

In order to connect RDF output produced by SparqPlug into a Web of data, links can be made to external data sets at extraction-time. Where an item's URI in an external data set can be derived from some variable within the extracted data, the target URI used in an RDF link can be minted using the `fn:mint` property function described below. Where this is not the case, additional property func-

tions would need to be defined in order to perform lookups of target URIs for such links.

4. MAINTENANCE

For many data sources, it is not sufficient to simply RDFize them once; they may change frequently. Therefore, it is important that SparqPlug is able to maintain a degree of accuracy in its data. To facilitate this, it records an SHA-512 message digest of each document it downloads and RDFizes. When a maintenance job is run, it is possible for the job worker to check a page for changes by comparing the digest of the page with that stored in the graph metadata store. If they differ, the page has been altered and SparqPlug knows to rerun the protoypical query over it. This is one of the issues in allowing a user to describe multiple input documents in the same named graph; if one of the documents is updated, it is unclear as to the best course of action. Either the entire graph must be regenerated, as there is no statement-level provenance information alongside them, or the data must be left intact, and only new data added to the store. The SparqPlug maintenance system opts for the latter approach, based on the principle that 'disappearing data' is to be avoided wherever possible, and that "Cool URIs don't change" [24]. The link query is executed for all documents run through the maintenance systems, in order to find any new pages that may have been added to this input set since the last RDFization was carried out.

At any time a user may request that a job's data is 'refreshed', by visiting its status page on the SparqPlug website. If the job has finished executing, then it will be queued in the regular work queue for processing.

5. IMPLEMENTATION DETAILS

5.1 Prototyping

Initial prototyping of the SparqPlug system was performed with RAP, the RDF API for PHP [25]. While this was an excellent solution for rapid development and proof-of-concept, it proved to lack the performance required for such an application. In particular, the SPARQL engine was approximately 10x slower than alternative implementations for slightly more complex queries: complex RDFization of a DOM Document could be done in approximately 90 seconds in ARQ [8], as opposed to 1.5 hours in RAP's `MemModel` SPARQL engine.

5.2 Final Selection of Technologies

The final SparqPlug system has been implemented in the Java programming language, selected for its efficiency, scalability, and portability. The RDF API, SPARQL, and graph persistence facilities are provided by the Jena Semantic Web Framework [26]. SparqPlug uses the NG4J extension to the Jena Framework for supporting Named Graphs and Datasets [27]. ARQ [8], the SPARQL engine for Jena, offers so-called 'property functions' [9], in SPARQL queries. These are not explicitly specified by the DAWG (the W3C Data Access Working Group), but can be considered similar to describing custom inference over a model at query-time. Property Functions to allow users to concatenate strings, match regular expressions, and mint URIs have been added to the ARQ library for SparqPlug, in order to allow users to use these powerful constructs in their queries to produce higher quality RDF linked data.

The DOM library selected is JTidy [28], one of a number of open-source tag balancing and tidying libraries. It describes itself as a Java implementation of the standard HTML Tidy, implementing many (but not all) of the W3C-specified DOM functions. The web service is capable of being run in any J2EE compliant servlet container (it makes use of connection pooling and Filter APIs provided by the J2EE spec), and is presently run within Apache Tomcat [29]. Similarly, any JDBC-compliant database server may be used; MySQL [30] is the current database of choice, for scalability and ease of deployment.

5.3 Scalability

The job processing code runs separately to the web service, as a simple command line application. It has been designed to operate concurrently, making use of Jena's inbuilt locking and concurrency support. In this way the SparqPlug system can be scaled up to run on any number of servers; the database and application containers support server clustering, and separate instances of the job dispatcher may run concurrently over this database. A single instance of the job dispatcher may also multithread across multiple jobs on a single server. In this way, the application is ready to scale along whichever parameters are required without modification to its code.

6. CASE STUDY: SPARQPLUG IN ACTION

The SparqPlug system has, as a stated goal, the aim of making the RDFization process simpler for DOM documents. The extent to which this has been a success can be examined through sample use of the system. Due to its nature, it requires a fair degree of Semantic Web-related knowledge to be usable. A user will be able to craft the most efficient and accurate queries only after some experience with the system. In the following example, we will RDFize the database of TimeOut Films [31], as a sample data source.

The HTML describing films in these pages follows a repeating tabular structure, making it particularly suitable for RDFization. For example:

```
<tr class="">
    <td><ul class="bullets">
        <li>
            <a href="/film/reviews/
            69923/cry-freedom.html">
            Cry Freedom</a> (1987)
        </li>
    </ul></td>
</tr>
```

Once we, as a user, have ascertained the structure of the HTML, we may enter our fragment into the Query Builder. Continuing this example, we may wish to attempt to make the query as simple as possible, so describe the films as:

```
<ul class="bullets">
<li><a href="?review">?title</a>?date</li>
</ul>
```

This will create us the following prototypical query:

```
PREFIX  apf:
<java:com.hp.hpl.jena.query.pfunction.library.>
PREFIX  dom:
```

```
<http://sparqplug.rdfize.com/ns/dom/>
PREFIX  tags:
<http://sparqplug.rdfize.com/ns/tags/>
PREFIX  attr:
<http://sparqplug.rdfize.com/ns/attributes/>
PREFIX  fn:
<java:com.rdfize.functions.>

SELECT  ?review ?title ?date
WHERE
  { ?nodeID8  dom:nodeName       tags:ul;
              dom:subNode        ?nodeID9;
              dom:hasAttributes  _:b1.
    _:b1      attr:class         "bullets".
    ?nodeID9  dom:nodeName       tags:li;
              dom:subNode        ?nodeID10;
              dom:subNode        ?nodeID14.
    ?nodeID10  dom:nodeName      tags:a;
              dom:subNodeNo      "1";
              dom:subNode        ?nodeID11;
              dom:hasAttributes  _:b0.
    _:b0      attr:href          ?review.
    ?nodeID11  dom:nodeName      tags:text;
              dom:nodeValue      ?title.
    ?nodeID14  dom:nodeName      tags:text;
              dom:subNodeNo      "2";
              dom:nodeValue      ?date.
  }
```

It is worth noting that it has also constructed our boilerplate link query, although we shan't pay attention to this until later. The query below is a reformatted version of the query, to make it slightly easier to understand (taking advantage of SPARQL's syntactic sugar for querying Collections), as well as for columnar layout purposes.

```
PREFIX  apf:
<java:com.hp.hpl.jena.query.pfunction.library.>
PREFIX  dom:
<http://sparqplug.rdfize.com/ns/dom/>
PREFIX  tags:
<http://sparqplug.rdfize.com/ns/tags/>
PREFIX  attr:
<http://sparqplug.rdfize.com/ns/attributes/>
PREFIX  rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  rfn:
<java:com.rdfize.functions.>
PREFIX  fn:
<http://www.w3.org/2005/xpath-functions#>

SELECT DISTINCT  ?linkCon
WHERE
  { ?nodeID8  dom:nodeName       tags:a;
              dom:hasAttributes  _:b0.
    _:b0      attr:href          ?link.
      { ?x        rfn:cat
("http://www.timeout.com/film/index/film/a/"
?link ).
        ?linkCon  rfn:mint   ?x.
        FILTER (
        !( fn:starts-with(?link, "http")
        || fn:starts-with(?link, "/"))
        )
      }
    UNION
      { ?x        rfn:cat
("http://www.timeout.com" ?link).
        ?linkCon  rfn:mint   ?x.
        FILTER fn:starts-with(?link, "/")
      }
    UNION
      { ?linkCon  rfn:mint  ?link.
        FILTER fn:starts-with(?link,
        "http://www.timeout.com")
      }
  }
```

If we run this RDFization through the Query Builder, we will find it returns a table of results, as we would expect. However, it takes approximately three minutes to do so; this is not terrible efficiency, but it can be improved upon. It may seem counter-intuitive, but by adding *more* constraints to a query, we can reduce the number of cross products ARQ has to construct, and thus improve efficiency of the query. Furthermore, the above query also matches some non-film data; while we could `FILTER` this dirty data out in the SPARQL query, a more specific query may be able to match more accurately. Suppose we change our fragment to:

```
<tr class="">
<td>
<ul class="bullets">
<li>
<a href="?review">?title</a>?date</li>
</ul>
</td>
</tr>
```

This will create us the following prototypical query:

```
PREFIX  apf:
<java:com.hp.hpl.jena.query.pfunction.library.>
PREFIX  dom:
<http://sparqplug.rdfize.com/ns/dom/>
PREFIX  tags:
<http://sparqplug.rdfize.com/ns/tags/>
PREFIX  attr:
<http://sparqplug.rdfize.com/ns/attributes/>
PREFIX  fn:
<java:com.rdfize.functions.>

SELECT  ?review ?title ?date
WHERE
  { ?nodeID8  dom:nodeName       tags:table;
              dom:subNode        ?nodeID9.
    ?nodeID9  dom:nodeName       tags:tr;
              dom:subNode        ?nodeID10;
              dom:hasAttributes  _:b2.
    _:b2      attr:class         "".
    ?nodeID10  dom:nodeName      tags:td;
              dom:subNode        ?nodeID11.
    ?nodeID11  dom:nodeName      tags:ul;
              dom:subNode        ?nodeID12;
              dom:hasAttributes  _:b1.
```

```
    _:b1      attr:class          "bullets".
    ?nodeID12  dom:nodeName       tags:li;
              dom:subNode         ?nodeID13;
              dom:subNode         ?nodeID17.
    ?nodeID13  dom:nodeName       tags:a;
              dom:subNodeNo       "1";
              dom:subNode         ?nodeID14;
              dom:hasAttributes   _:b0.
    _:b0      attr:href           ?review.
    ?nodeID14  dom:nodeName       tags:text;
              dom:nodeValue       ?title.
    ?nodeID17  dom:nodeName       tags:text;
              dom:subNodeNo       "2";
              dom:nodeValue       ?date.
  }
```

This new query will execute in under a second; a significant improvement over the old running times. However, it is returning approximately half the number of results. Returning to the Time-Out page, we notice that many of the listed films do not have a date attached. By altering the query to make the `?date` clause optional, we may attain a much better set of results, thus:

```
PREFIX  dom:
<http://sparqplug.rdfize.com/ns/dom/>
PREFIX  tags:
<http://sparqplug.rdfize.com/ns/tags/>
PREFIX  attr:
<http://sparqplug.rdfize.com/ns/attributes/>
PREFIX  fn:
<java:com.rdfize.functions.>

SELECT  ?review ?title ?date
WHERE
  { ?nodeID8  dom:nodeName       tags:table;
              dom:subNode        ?nodeID9.
    ?nodeID9  dom:nodeName       tags:tr;
              dom:subNode        ?nodeID10;
              dom:hasAttributes  _:b2.
    _:b2      attr:class         "".
    ?nodeID10  dom:nodeName      tags:td;
              dom:subNode        ?nodeID11.
    ?nodeID11  dom:nodeName      tags:ul;
              dom:subNode        ?nodeID12;
              dom:hasAttributes  _:b1.
    _:b1      attr:class         "bullets".
    ?nodeID12  dom:nodeName      tags:li;
              dom:subNode        ?nodeID13;
              dom:subNode        ?nodeID17.
    ?nodeID13  dom:nodeName      tags:a;
              dom:subNodeNo      "1";
              dom:subNode        ?nodeID14;
              dom:hasAttributes  _:b0.
    _:b0      attr:href          ?review.
    ?nodeID14  dom:nodeName      tags:text;
              dom:nodeValue      ?title.
    OPTIONAL {
        ?nodeID17  dom:nodeName tags:text;
                   dom:subNodeNo "2";
                   dom:nodeValue ?date.
    }
  }
```

Now that we are able to `SELECT` a suitable set of results, the next stage is to construct it into a useful RDF model. To do this, we will make use of some of the Property Functions developed for SparqPlug. Some of the desirable alterations to the data would be;

- Minting URIs for each of the films
- Removing the parentheses from around the dates
- Resolving the Review URL to an absolute path
- Minting a URI Resource from the Review URL

6.1 Minting The Film's URI

The basic technique for this will be to first match the film name out of the review URL, then append the ID number of the film (as a disambiguation step between films of the same name), putting it into the SparqPlug namespace, and finally minting it as a URI Resource. This can be done as in the following SPARQL fragment, binding `?r` to the Resource:

```
(?m0 ?m1 ?m2) fn:match
    ( ?review "\\/film\\/reviews
      \\/(.*)\\/(.*)\\.html" ).

?x fn:cat
    ( "http://sparqplug.rdfize.com
      /jobs/TimeOutFilms/things/"
      ?m2 "-" ?m1 ).

?r fn:mint ?x.
```

However, since this is going to be a regular 'incantation' for a user to employ, a property function, `fn:thing`, was introduced, along with a new namespace for URIs. The `fn:thing` function takes two parameters as an object-list, and mints a URI as its subject of the form:
`<http://uris.rdfize.com/JobName/LocalName>`.
This URI is dereferencable in the typical Linked-Data fashion, and will 303-redirect a user to the representation they request in the `http://sparqplug.rdfize.com/jobs/` namespace.

Thus, the above query fragment can be altered, removing the onus of accurate namespace allocation from the user:

```
(?m0 ?m1 ?m2) fn:match
    ( ?review "\\/film\\/reviews
      \\/(.*)\\/(.*)\\.html" ).

?local fn:cat  ( ?m2 "-" ?m1 ).

?r fn:thing ( "TimeOutFilms" ?local ).
```

6.2 Removing Parentheses

Having already seen the use of `fn:match` above, this is a fairly simple process:

```
(?y0 ?year) fn:match
    (?date "\\((.*)\\)").
```

6.3 Resolving and Minting the Review URI

```
    ?rc fn:cat
        ("http://www.timeout.com" ?review).
    ?rev fn:mint ?rc.
```

6.4 Putting it together

We now have all of the pieces ready to `CONSTRUCT` our RDF model. This is the time we also need to choose the ontology / ontologies we wish to use to describe the Films. For this example, we will use the popular Dublin Core [32] ontology, as well as statements from the Review Ontology and the RDFS Yago [33] class hierarchy hosted by DBpedia [34].

```
PREFIX  dom:
<http://sparqplug.rdfize.com/ns/dom/>
PREFIX  tags:
<http://sparqplug.rdfize.com/ns/tags/>
PREFIX  attr:
<http://sparqplug.rdfize.com/ns/attributes/>
PREFIX  fn:
<java:com.rdfize.functions.>
PREFIX dc:
<http://purl.org/dc/elements/1.1/>
PREFIX rev:
<http://purl.org/stuff/rev#>
PREFIX yago:
<http://dbpedia.org/class/yago/>

CONSTRUCT {
  ?r a yago:MotionPictureFilm103789400;
     dc:date ?year;
     dc:title ?title;
     rev:hasReview ?rev.
}
WHERE
  { ?nodeID8  dom:nodeName       tags:table;
              dom:subNode        ?nodeID9.
    ?nodeID9  dom:nodeName       tags:tr;
              dom:subNode        ?nodeID10;
              dom:hasAttributes  _:b2.
    _:b2      attr:class         "".
    ?nodeID10  dom:nodeName      tags:td;
              dom:subNode        ?nodeID11.
    ?nodeID11  dom:nodeName      tags:ul;
              dom:subNode        ?nodeID12;
              dom:hasAttributes  _:b1.
    _:b1      attr:class         "bullets".
    ?nodeID12  dom:nodeName      tags:li;
              dom:subNode        ?nodeID13;
              dom:subNode        ?nodeID17.
    ?nodeID13  dom:nodeName      tags:a;
              dom:subNodeNo      "1";
              dom:subNode        ?nodeID14;
              dom:hasAttributes  _:b0.
    _:b0      attr:href          ?review.
    ?nodeID14  dom:nodeName      tags:text;
              dom:nodeValue      ?title.

    (?m0 ?m1 ?m2) fn:match (?review
    "\\/film\\/reviews\\/(.*)\\/(.*)\\.html"
    ).
    ?uc fn:cat (
        "http://sparqplug.rdfize.com/jobs
         /TimeOutFilms/things/"
        ?m2 "-" ?m1 ).
    ?r fn:mint ?uc.

    ?rc fn:cat ("http://www.timeout.com"
        ?review).
    ?rev fn:mint ?rc.

    OPTIONAL {
        ?nodeID17  dom:nodeName tags:text;
                   dom:subNodeNo "2";
                   dom:nodeValue ?date.
        (?y0 ?year) fn:match
            (?date "\\((.*)\\)").
    }

  }
```

This query can be tested against any of the pages in the TimeOut Films dataset, producing RDF (in under 200ms) for each film similar to:

```
<http://sparqplug.rdfize.com/jobs
/TimeOutFilms/things/cry-freedom-69923>
      a  yago:MotionPictureFilm103789400;
      dc:date "1987";
      dc:title "Cry Freedom";
      rev:hasReview
<http://www.timeout.com/film/reviews
/69923/cry-freedom.html>.
```

6.5 Link SPARQL

The final consideration before designing the job for this is construction of the link query; the boilerplate query that is generated will result in crawling the entire TimeOut website, producing a large number of empty graphs, or worse, bad data. We therefore want to trim its results down to limit it only to URLs that start in `/film/index/film/`, as follows:

```
PREFIX  dom:
<http://sparqplug.rdfize.com/ns/dom/>
PREFIX  tags:
<http://sparqplug.rdfize.com/ns/tags/>
PREFIX  attr:
<http://sparqplug.rdfize.com/ns/attributes/>
PREFIX  rfn:
<java:com.rdfize.functions.>
PREFIX  fn:
<http://www.w3.org/2005/xpath-functions#>

SELECT DISTINCT  ?link
WHERE
  { ?nodeID8  dom:nodeName       tags:a;
              dom:hasAttributes  _:b0.
    _:b0      attr:href          ?l.
    ?x        rfn:cat
        ( "http://www.timeout.com" ?l ).
    ?link     rfn:mint           ?x.
    FILTER fn:starts-with(?l,
        "/film/index/film/")
  }
```

This query will match approximately 30 results from each page; it is important to note that when run in a job, the query is executed against each page that is RDFized, so when (as with TimeOut Films) results are paginated, it will find all of them through the course of the job's execution.

6.6 Designing the SparqPlug Job

Much of the work in preparing the job is now complete. The job now only needs to be named and described, and the Graph RegExp decided upon. This regular expression is used to determine which named graph the data from each page should be put into. It is important to note that a separate RDF Dataset exists for each job, so no named graph collisions will occur between jobs. The default RegExp,

```
^.*/([^/]*)(\.html|\.htm|\.php|\.jsp)$
```

will use the page's name for its graph. In this case, that is less than desirable; it would make more sense to split up the dataset alphabetically, in exactly the same manner as TimeOut organises them. Therefore, we will use the following RegExp:

```
^.*/([^/]*)/[^/]*\.html$
```

The job is now ready to be commited and executed. As soon as it is in the job queue, its URI becomes dereferencable for information about its progress, and a snapshot of the current RDF available. Things in the job's namespaceare also dereferencable as soon as they are in the dataset.

7. EVALUATION

The example job described above was executed on the SparqPlug web service before launch, as a simple performance indicator. Including network latency, the full RDFization of a page (`GET` the page, extract links and add them to the work queue, `CONSTRUCT` the graph, and add the graph to the dataset) takes approximately 5 seconds. The full RDFization of the TimeOut Films dataset took around 25 minutes, and involved RDFizing 228 HTML pages.

In the interests of fair testing, we proceeded to set up a performance testing harness. For each of our performance tests, we execute a SPARQL query against an HTML fragment; in this way we test both the DOM2RDF and SPARQL Query components of the system, in order to best simulate the process of a job's execution. Each test is repeated five times, and the average execution time taken.

The first test we ran was to quantitively evaluate query execution with a single query over a varying length of HTML list. The HTML was constructed as a list of 0 - 1000 elements, as in the following example snippet:

```
<ul>
    <li>
        <p>Row 0, Text Item 0</p>
        <p>Row 0, Text Item 1</p>
..................<SNIP>..................
        <p>Row 0, Text Item 8</p>
        <p>Row 0, Text Item 9</p>
    </li>
```

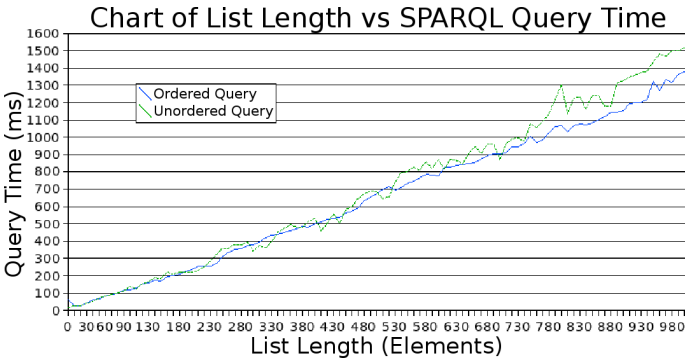

Figure 2: Chart of List Length vs SPARQL Query Time

```
.................<SNIP SNIP>................
    <li>
        <p>Row 99, Text Item 0</p>
        <p>Row 99, Text Item 1</p>
...................<SNIP>...................
        <p>Row 99, Text Item 8</p>
        <p>Row 99, Text Item 9</p>
    </li>
</ul>
```

These data constructs were then each queried with two SPARQL queries, to extract the paragraph text from within the list items. The two queries were semantically identical, but one was ordered intelligently, and the second had its triple patterns put into a random order. These queries remained constant throughout the tests.

```
PREFIX dom:
<http://sparqplug.rdfize.com/ns/dom/>
PREFIX tags:
<http://sparqplug.rdfize.com/ns/tags/>
SELECT ?someValue
WHERE
{
  ?nodeID8  dom:nodeName   tags:ul;
            dom:subNode    ?nodeID9.
  ?nodeID9  dom:nodeName   tags:li;
            dom:subNode    ?nodeID10.
  ?nodeID10 dom:nodeName   tags:p;
            dom:subNode    ?nodeID11.
  ?nodeID11 dom:nodeName   tags:text;
            dom:nodeValue  ?someValue.
}
```

The results of these tests are represented in Figure 2. These figures offer some support for the earlier observation that ordering the triple patterns in the query speeds up the query execution. It also suggests that the system as a whole scales linearly along with HTML fragment size.

8. FUTURE WORK

The SparqPlug service is fully operational in its present state, and fulfils many of its original goals. However, learning from others in this space it may be possible to improve the user experience. One such area is that of optimisation; it can take an exceptionally long

time for certain queries to execute, particularly those that entail calculation of large cross products. By using the standard distribution of ARQ, we have ensured it will be possible to rapidly integrate future releases and optimisations of the SPARQL engine. Even so, it may be possible to decrease the amount of work the engine has to do. With improved understanding of the query planning and execution within ARQ, it should be possible for the Query Builder to produce queries which will execute more efficiently. Streamlining and optimising the RDF model output by the DOM2RDF component of SparqPlug with this query execution and planning in mind may also have a positive impact on performance. An alternative approach to optimisation would be to investigate attaching different SPARQL query processors and triple stores to the SparqPlug service.

Another aspect that could be improved is the user interface. The ability for a user to click-and-drag to select the area of HTML they wish to RDFize could, if coupled with query builder enhancements, allow SparqPlug to combine the ease of use in systems such as Thresher without reducing its query expressivity in any shape or form.

A final area that may prove fruitful to expand research into the SparqPlug approach is that of less formally structured data. SparqPlug can be very efficient and accurate over highly structured DOM documents (e.g. lists of entries, rows of tables etc.) It can, however, be challenging to construct a successful SparqPlug job to, for example, parse the contents of a wiki page into useful RDF. An expansion of the property functions available to the SparqPlug engine, perhaps to normalise the data before processing, may prove valuable here. Another relevant approach here may be that of embedding custom property functions[35] as javascript inside the query.

9. ACKNOWLEDGEMENTS

This research was partially supported by the OpenKnowledge (OK) project. OK is sponsored by the European Commission as part of the Information Society Technologies (IST) programme under grant number IST-2001-34038.

10. REFERENCES

[1] Auer, S; Bizer, C; Kobilarov, G; Lehmann, J; Cyganiak, R; Ives, Z: “DBpedia: A Nucleus for a Web of Open Data”. In Proc. of the 6th Intl. Semantic Web Conference (ISWC2007), Busan, Korea.

[2] http://www.wikipedia.org/

[3] Prud’hommeaux, E; Seaborne, A: “SPARQL Query Language for RDF" http://www.w3.org/TR/rdf-sparql-query/

[4] Gennari, J; Musen, M; Fergerson, R; Grosso, W; Crubézy, M; Eriksson, H; Noy, N; Tu, S: “The evolution of Protégé: an environment for knowledge-based systems development". In International Journal of Human-Computer Studies (January 2003)

[5] Huynh, D; Miller, R; Karger, D: “Enabling Web Browsers to Augment Web Sites’ Filtering and Sorting Functionalities". In Proc. of the 19th annual ACM symposium on User interface software and technology, Montreux, Switzerland.

[6] http://www.w3.org/TR/grddl

[7] Droop, M; Flarer, M; Groppe, J; Groppe, S; Linnemann, V; Pinggera, J; Santner, F; Schier, M; Schoepf, F; Staffler, H; Zugal, S: “Translating XPath Queries into SPARQL Queries". In Proc. of On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops

[8] http://jena.sourceforge.net/ARQ/

[9] Seaborne, A: ARQTick 13 October 2006 - “Assignment Property Function" http://seaborne.blogspot.com/2006/10/assignment-property-function.html

[10] Eric Prud’hommeaux: “SPAT - SPARQL Annotations" http://www.w3.org/2007/01/SPAT/

[11] Welty, C; Murdock, J W: “Towards Knowledge Acquisition from Information Extraction". In Proc. of the 5th Intl. Semantic Web Conference (ISWC2006), Athens, Georgia, USA.

[12] Hogue, A; Karger, D: “Thresher: automating the unwrapping of semantic content from the World Wide Web”. In Proc. of the 14th Intl. World Wide Web conference (WWW2005), Chiba, Japan.

[13] Kalyanpur, A: “RDF Web Scraper" http://www.mindswap.org/~aditkal/rdf.shtml

[14] http://simile.mit.edu/wiki/Piggy_Bank

[15] http://openlinksw.com/virtuoso

[16] http://triplr.org

[17] http://microformats.org

[18] Birbeck, M; Pemberton, S; Adida, B: “RDFa Syntax - A collection of attributes for layering RDF on XML languages" http://www.w3.org/2006/07/SWD/RDFa/syntax/

[19] “ Rdf In Html" http://research.talis.com/2005/erdf/wiki/Main/RdfInHtml

[20] http://base.google.com

[21] http://flickr.com

[22] http://del.icio.us

[23] Bizer, C; Cyganiak, R; Heath, T: “How to Publish Linked Data on the Web” http://sites.wiwiss.fu-berlin.de/suhl/bizer/pub/LinkedDataTutorial/

[24] Sauermann, L; Cyganiak, R; Ayers, D; Voelkel, M: “Cool URIs for the Semantic Web" http://www.w3.org/TR/2007/WD-cooluris-20071217/

[25] Oldakowski, R; Bizer, C; Westphal, D: “RAP: RDF API for PHP". In Proc. of the 1st Workshop on Scripting for the Semantic Web (SFSW2005), 2nd European Semantic Web Conference (ESWC2005), Heraklion, Greece

[26] http://jena.sourceforge.net/

[27] Bizer, C; Cyganiak, R: “NG4J - Named Graphs API for Jena" http://sites.wiwiss.fu-berlin.de/suhl/bizer/ng4j/

[28] http://jtidy.sourceforge.net/

[29] http://tomcat.apache.org/

[30] http://www.mysql.org/

[31] http://www.timeout.com/film/index/film/a/1.html

[32] http://purl.org/dc/elements/1.1/

[33] Suchanek, F; Kasneci, G; Weikum, G: “Yago - A Core of Semantic Knowledge". In Proc. of 16th Intl. World Wide Web conference (WWW2007), Banff, Alberta, Canada.

[34] http://dbpedia.org/

[35] Williams, G: “Extensible SPARQL Functions With Embedded Javascript". From 3rd Workshop on Scripting for the Semantic Web (SFSW2007), 4th European Semantic Web Conference (ESWC2007), Innbruck, Austria.