Web Semantics: Science, Services and Agents on the World Wide Web 26 (2014) 29–35

Contents lists available at ScienceDirect

Web Semantics: Science, Services and Agents on the World Wide Web

journal homepage: www.elsevier.com/locate/websem

The SWGET portal: Navigating and acting on the web of linked data

Valeria Fionda [a], Claudio Gutierrez [b], Giuseppe Pirrò [c,*]

[a] Department of Mathematics, University of Calabria, Italy
[b] Computer Science Department, Universidad de Chile, Chile
[c] Faculty of Computer Science, Free University of Bozen-Bolzano, Italy

ARTICLE INFO

Article history:
Received 29 April 2013
Received in revised form
3 April 2014
Accepted 17 April 2014
Available online 28 April 2014

Keywords:
Semantic navigation
Scripts
Linked open data

ABSTRACT

This paper presents the SWGET portal. By using the portal, users can instruct software modules to (virtually) move from one place (data source) to another on the Web of Data, interpret knowledge and trigger actions much in the same spirit of intelligent *agents*. Instructions are specified via navigational expressions in the NAUTILOD language. Such expressions are included into SWGET scripts that is, RDF documents that can be shared, modified, and reused. We discuss examples with real data in different scenarios showing the usefulness and potentialities of the portal. We also provide an evaluation of the performance of the portal.

1. Introduction

The increasing availability of structured data is evolving the current Web, based on hypertext documents and syntactic links among them, into a Web of Data. Here, Uniform Resource Identifiers (URIs) are used not only to identify Web documents and digital content, but also new kinds of resources such as real world (e.g., people) and abstract (e.g., sport) concepts. The Linked Open Data (LOD) project [1] is one of the driving forces in this direction. It leverages well-established Web technologies like URIs to identify resources, and the HTTP protocol to retrieve their representations available as triples in the Resource Description Framework (RDF) [2] format. RDF data on the Web of Linked Data (WLOD) are commonly accessed via SPARQL endpoints, that is, network locations that can be queried upon by using the SPARQL query language [3]. This approach, however, cannot cope with the highly distributed nature of interconnected RDF data sources. In particular, it does not provide mechanisms to dynamically discover/select (relevant) data sources. The WLOD can be modeled as a distributed (semantic) graph with thousands of RDF data sources (its nodes) and semantic links between them (its edges). An essential feature to retrieve data in graphs is navigation. Despite the large number of graph languages and tools today available (see [4] for a survey), none of them focuses on navigation in the WLOD. "Consuming" data in this graph is challenging due to the intrinsically decentralized data creation/management, the lack of superimposed schema and the unknown topology of data sources and their links. In the WLOD, which is *not known* in its entirety, navigation amounts at going from one data source to another (unknown) data source guided by some navigational chart (i.e., a navigational expression). For instance, from the node R. Johnson in DBpedia (the known) it is possible to navigate toward the nodes of artists he influenced (the unknown) by traversing edges labeled as `influenced` (the navigational chart). A desirable feature of (navigational) languages for the WLOD is the possibility to command actions (e.g., retrieval/update of data, sending of notification messages, etc.). To the best of our knowledge, current languages lack such a feature.

Contributions. This paper describes the SWGET portal that allows users to instruct software modules to (virtually) move across data sources, interpret knowledge, take decisions and trigger actions much in the same spirit of the intelligent agents envisioned in the early days of the Semantic Web [5]. Instructions are given via the navigational language NAUTILOD [6]. We describe the SWGET portal, present examples with real data in different scenarios and evaluate its performance.

2. Related work

Navigation and specification languages of nodes in a graph have a long tradition. Nevertheless, most of the existing approaches assume that data is stored in a central repository (e.g., graph query languages [4], *XPath*, navigational versions of SPARQL [7,8]). They gave inspiration for the navigational core of NAUTILOD.

* Corresponding author. Tel.: +39 0471016122.
E-mail addresses: fionda@mat.unical.it (V. Fionda), cgutierr@uchile.cl (C. Gutierrez), pirro@inf.unibz.it, giuseppe.pirro@unibz.it (G. Pirrò).

http://dx.doi.org/10.1016/j.websem.2014.04.003
1570-8268/

Specification (and retrieval) of collections of sites was early addressed by tools like `wget`.[1] Besides being non-declarative, it is restricted to almost purely syntactic features. There is a solid body of work on query processing and navigation on the WLOD. Three main lines of research can be identified:

(1) Load the *desired* data into a single RDF store (by *crawling* the WLOD [9] or some sub-portions) and process queries in a centralized way. There have been also developments in indexing techniques for semantic data (e.g., Sindice [10], Watson [11], Harth et al. [12]).

(2) Federated query processing like DARQ [13]. SPARQL 1.1 [14], with the SERVICE operator, extends the scope of SPARQL to federated queries.

SWGET has a different departure point: it focuses on *navigational* functionalities, thus departing from querying as in (2); emphasizes the declarative specification of autonomous distributed sources, as opposed to (1). We now provide a more in depth comparison with approaches that extend SPARQL to the WLOD.

Comparison with link-traversal query approaches. Several approaches (e.g., [15]) have been proposed to discover on the fly data sources relevant for answering SPARQL queries on the WLOD. At their core, there is the possibility to navigate toward other data sources while executing a query; SQUIN[2] is an implementation of such a mechanism. We discuss the differences w.r.t SWGET both in terms of scope and expressiveness.

First, we want to point out that SQUIN and SWGET have two different departure points. SQUIN uses "implicit" (not-controlled) navigation to collect information from the WLOD to perform query answering (i.e., obtain a set of variable mappings). Hence, navigation is a "means" to retrieve such information. Indeed, SQUIN is not a navigational language and does not have the features of a navigational language (e.g., closure). SWGET leverages NAUTILOD, which is a pure navigational language that uses (boolean ASK SPARQL) queries to select data sources. Hence, navigation (not querying) is the main actor. While NAUTILOD uses queries to enhance/control the navigation, SQUIN proceeds in the reverse direction. Moreover, the Link-Traversal Based Query Execution mechanism based on non-blocking iterators implemented by SQUIN [15] cannot guarantee that all reachable URIs that may contribute to the final results are discovered. This is because the building blocks of a SQUIN query (i.e., SPARQL basic graph patterns) are evaluated in a fixed order; when an iterator obtains an intermediate solution from the previous iterator, it replaces the previously obtained intermediate solution. Hence, the (replaced) intermediate solution cannot be combined with any data that arrives later. As for SPARQL 1.1's, its navigational core (i.e., property paths) is meant to deal with paths that link RDF triples available in a *local* graph. NAUTILOD (and SWGET) deals with the WLOD graph where paths exist between distributed (and a priori unknown) data sources. We want to point out that NAUTILOD incorporates actions that in some sense generalize procedures implicit in the evaluation over the Web (e.g., "get data" in crawlers and "return data" in query languages).

3. Instructing intelligent applications with SWGET

We start with an overview of NAUTILOD, which is at the core of SWGET scripts. NAUTILOD [6] is a language that enables to write declarative specifications of navigational charts for the WLOD graph. More specifically, NAUTILOD provides a mechanism to declaratively: (i) define *navigational expressions*; (ii) allow *semantic control* over the navigation via test queries; (iii) *perform actions* as side-effects along the navigational path. The syntax of NAUTILOD is reported in Table 1.

Table 1
Syntax of NAUTILOD.

`path::=`	`pred | action | path/path | path⟨l − h⟩`
	`| (path)? | (path)* | (path|path) | path[test]`
`pred::=`	`< RDFpredicate > | < _ >`
`test::=`	`ASK-SPARQL query`
`action::=`	`ACT[Select-SPARQL query::procedure]`

The navigational core of the language is based on regular path expressions, pretty much like Web query languages and XPath. A `path` includes RDF predicates (composed with all the features of regular languages); moreover, `path⟨l − h⟩` denotes repetitions between `l` and `h`. A `path` can be followed by a `test`; an ASK-SPARQL query that allows to redirect the navigation based on the information present at each node (data source) in the navigational path.[3] Finally, `actions` are procedural routines that can be triggered during the navigation according to decisions based on the original specification and information found in the visited data sources. Actions do not interfere with the navigation. A NAUTILOD expression is evaluated in the WLOD starting from a *seed* node and returns a *set of nodes* connected to the seed node via paths matching the expression. An example of evaluation of a NAUTILOD expression is provided in Section 4.

3.1. From set of nodes to subgraphs

Current graph navigational languages (e.g., nSPARQL [8]) enable to retrieve sets of nodes connected by a sequence of edges that match an expression. However, they do not provide information about the structure of the fragment of the graph where these nodes have been found. Such pieces of information are crucial in some contexts like citation networks where one wants to see not only nodes (i.e., papers) but also their connections. Hence, there is the need to augment current navigational languages with capabilities to extract *fragments* (i.e., subgraphs) of the graph being navigated, besides sets of nodes. Solving this problem brings some challenges since: (i) only relevant paths should be kept in the fragment (i.e., paths spelling strings belonging to the language defined by the navigational expression) and (ii) there can be an exponential number of paths connecting the seed node with each node in the results. In the Web setting, the problem becomes even more challenging since the graph structure is discovered during the navigation. The SWGET system described in Section 4 implements an extension of NAUTILOD, which returns not only the expression endpoints but also the (relevant) portion of the WLOD visited during the evaluation of an expression [16]. The idea is to leverage the product between the automaton associated to a NAUTILOD expression and the data graph; then, navigate the product backward from the results to the seed node.

3.2. SWGET scripts syntax

An SWGET script is an RDF document suitable to be shared/exchanged/reused. SWGET scripts are written according to an ontology that supports their semantic specification. The following definition describes the structure of a script.

Definition 1 (SWGET *Script*)**.** An SWGET script $\mathcal{S}$ is a tuple $\langle n, G, s, e\rangle$, where n is the URI that identifies the script, G is an RDF graph, s is the seed URI where the navigation starts and e is a NAUTILOD expression.

[1] www.gnu.org/software/wget.
[2] http://squin.org.
[3] The result of the test depends on how triples are associated to the URI identifying the data source on which the test is performed.

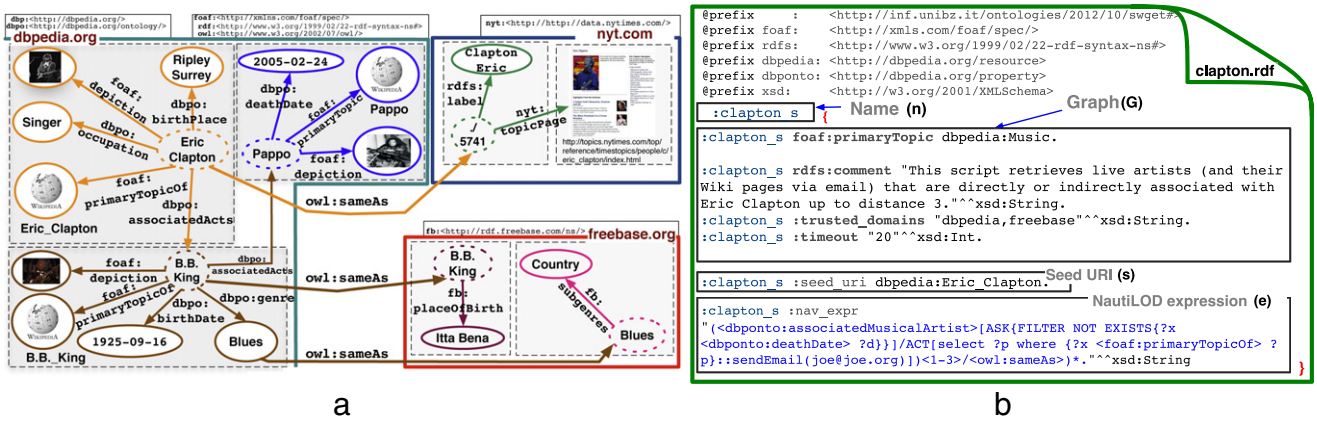


Fig. 1. An excerpt of the WLOD with real data and an SWGET script.

Example. Joe is a fan of Eric Clapton and wants to discover artists (and their aliases) (in)directly associated with Clapton up to distance 3. In particular, he is interested in chains of artists who are still alive and wants to receive via email their Wiki pages.

In order to fulfill this request Joe writes the SWGET script shown in Fig. 1(b). The first thing to do is to create a script and name it (i.e., `clapton.rdf`). Then, a graph G can be defined with triples stating, for instance, the topic of the script (i.e., Music), a comment in natural language to facilitate its reuse, etc. Besides, some parameters to bound the portion of WLOD visited can be also defined; here it is stated (predicate: `trusted_domains`) that only information from `dbpedia.org` and `freebase.org` should be trusted and further processed. Also a timeout has been set (predicate: `timeout`). The complete list of options is available at the SWGET website (http://swget.wordpress.com). The next step consists in specifying (predicate: `seed_uri`) the seed URI where the navigation starts. In the example the navigation starts from the URI associated to Eric Clapton in DBpedia. The last step is to define (predicate: `nav_expr`) the NAUTILOD expression. To profitably use SWGET and NAUTILOD, the user has to have some familiarity with the underlying ontologies. A quick exploration of DBpedia suggests that the predicate `dbponto:associatedActs` connects an artist to associated artists (see Fig. 1(a)).

As Joe is only interested in chains including artists who are still alive, some filtering is necessary. This is done by means of an ASK SPARQL query over the predicate `dbpo:deathDate`. Joe also specifies an action that triggers the sending of an email containing the Wiki pages of such artists (for which the predicate `foaf:isPrimaryTopicOf` can be used). Finally, the `owl:sameAs` predicate is used to combine information about artists from multiple sources. Note that besides `owl:sameAs` one could also use other predicates expressing likeness. A template like (`owl:sameAs` | `skos:related` | `rdf:seeAlso`) captures different likeness predicates defined in different vocabularies.

By executing this script Joe obtains the URIs {`dbpedia:B.B.King`, `freebase:B.B.King`}. Note that from the URI of B.B. King it is also possible to reach the data source associated to the rock musician Pappo (also in DBpedia). However, since Pappo did not pass the test defined in the ASK query (he is not alive) this navigational branch ended. The execution of the script also triggers the sending of an email with the Wiki page of B.B. King.

4. The SWGET web portal

The main objective of the SWGET portal is to enable users to write scripts containing navigational expressions to be evaluated over the WLOD. SWGET implements all the features of

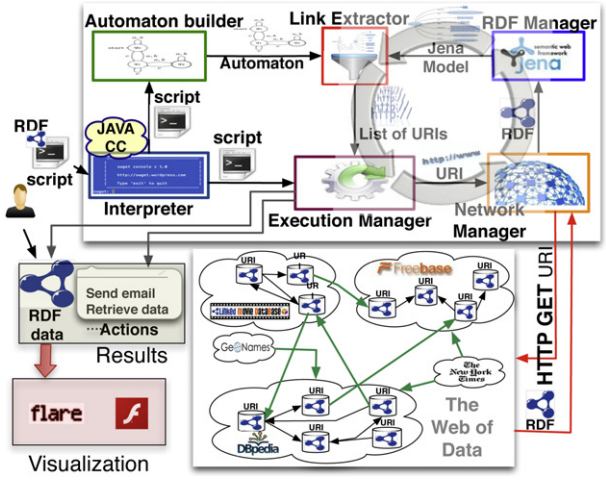


Fig. 2. The SWGET portal architecture.

NAUTILOD and adds a set of additional features to control the navigation from a network point of view (e.g., limiting the amount of data transferred) and a set of actions (e.g., send email messages, retrieve data). SWGET is available in three different releases: (i) an API equipped with a command line tool; (ii) a standalone GUI[4]; (iii) a Web portal.[5] The SWGET portal has been implemented by using Adobe Flex[6] and the Flare data visualization library.[7] The portal builds upon the SWGET API, which is implemented in Java and uses technologies such as the HTTP protocol to retrieve data directly from RDF data sources, JavaCC[8] to parse expressions and Jena[9] to deal with RDF data.

The high level architecture of the portal is shown in Fig. 2. The user submits to the system a SWGET script; the *Interpreter* receives the input, checks the syntax and initializes both the *Execution Manager* (with the *seed* URI) and the *Automaton builder*, which generates the Non Deterministic Finite State Automaton (NFA) associated to the NAUTILOD expression. At this point, the evaluation of the expression consists in a cyclic information exchange between *Execution Manager*, *Network Manager*, *RDF Manager* and *Link Extractor*. To better clarify how the components

[4] http://swget.wordpress.com.
[5] http://swget.inf.unibz.it.
[6] http://www.adobe.com/products/flex.html.
[7] http://flare.prefuse.org.
[8] http://javacc.java.net.
[9] http://jena.apache.org.

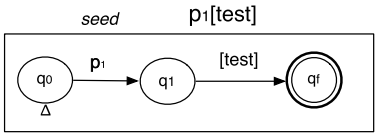


Fig. 3. An example of NAUTILOD expression and its automaton.

of the architecture work and interact we discuss an example of evaluation.

An example of evaluation. NAUTILOD expressions are built on top of regular expressions. If an expression complies with the NAUTILOD syntax, then the NFA that recognizes it is generated by using the classical Thompson construction. Hence, the evaluation of expressions is automaton-based. Consider the simple expression and the automaton in Fig. 3.

When entering the initial state of the automaton (i.e., q_0) the first step consists in dereferencing the *seed* URI. In order to do so, the *Execution Manager* passes this URI to the *Network Manager*, which issues an HTTP GET. The associated set of RDF triples is obtained by converting, via the *RDF Manager*, the stream associated to the HTTP GET into a Jena model M_{seed}. The evaluation continues by looking at transitions originating from q_0. These can be transitions labeled with RDF predicates, actions or tests. In this case there is the predicate p_1. The *Link Extractor* takes in input the NFA and the Jena model M_{seed} and selects a subset of outgoing links L_{q_1} to be traversed at the next step of the navigation. In this example, this is done via the SPARQL query `SELECT ?X WHERE{{seed` p_1 `?x}UNION{?x` p_1 `seed}}` that selects nodes appearing either as a subject or as an object. The set L_{q_1} is passed to the *Execution Manager*, which starts over the cycle. In particular, for each URI $u_i \in L_{q_1}$ a Jena model M_{u_i} is constructed as described above. At this point, the only transition originating from q_1 is labeled with a `test`. Recall that a test is an ASK SPARQL query. The query corresponding to the state transition between q_1 and q_f is executed over each model M_{u_i}. Only those URIs $u_i \in L_{q_1}$ for which the evaluation of the ASK query over M_{u_i} is `true` reach the final state q_f.

The evaluation produces: (i) the RDF graph obtained as the union of all the successful paths that connect the seed node to nodes in the result built according to the algorithm presented in Fionda et al. [16]; the graph is constructed by navigating backward the product automaton (from the final state to the initial) and (ii) the results of the actions fired during the navigation.

4.1. The user interface

The main window of the SWGET portal is shown in Fig. 4(a). The portal is organized in three tabs: (i) *Run script*, where it is possible to load existing scripts, create new scripts, lunch SWGET agents (associated to SWGET scripts), check the status of running agents and retrieve the list of active agents associated to a given email address; (ii) *GraphView*, which shows the RDF graph resulting from the evaluation of a script; (iii) *Learn*, which provides information about the NAUTILOD language and the portal. The create script window is opened when the user loads a predefined script or wants to create a new script. When a script is launched, a new agent is created. Each running agent is assigned an *ID*, which will be used to track its status and retrieve results. If an email address is provided, email notifications are sent when the agent starts and terminates. To check the status of an agent the *Check agent status* form in the *Run script* tab can be used (shown in Fig. 4(b)). When the agent is in the state *Terminated* it is possible to download the results or explore them in the *GraphView* tab.

There are different visualizations that can be changed by using the *Visualization control*. On the top part of the *GraphView* tab the automaton associated to the NAUTILOD expression in the script

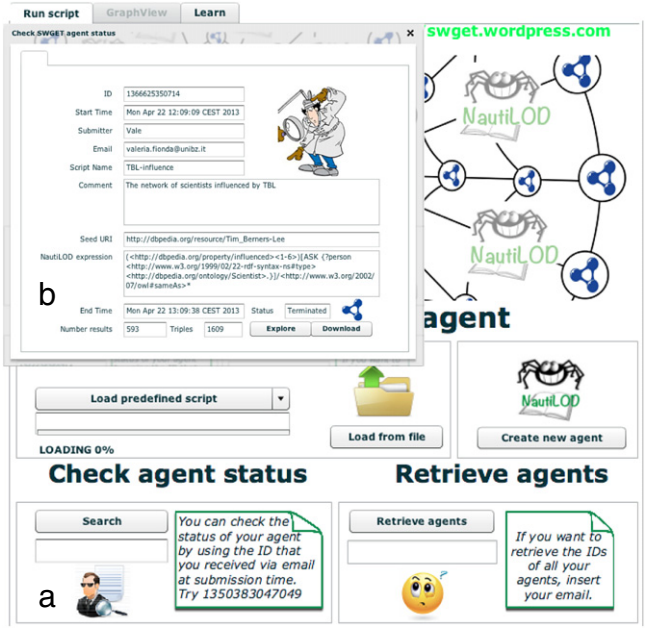


Fig. 4. The SWGET GUI. Visit http://swget.inf.unibz.it.

is reported. When the mouse pointer is placed on a state of the automaton all the nodes reachable in that state are highlighted; when the mouse pointer is placed on an edge, the predicate (or test) corresponding to that state transition is visualized.

5. Application scenario

Valerie is a scientific journalist who is writing an article about the Semantic Web and in particular about the figure of Tim Berners-Lee (TBL) and his cooperation with other people. She thinks that it would be nice to investigate the influence of TBL over other people and also by whom he has been influenced. It would be ideal to have a graphical visualization of this *influence network* and not just a set of "disconnected" nodes. Since Valerie is particularly interested in the scientific community, she wants to restrict the network to scientists only. Data to build such network can be taken from different data sources (e.g., DBpedia, Freebase); the problem is how to access in a clever and automatic way such data and present results in an attractive way. Valerie also thinks *"what if tomorrow I am interested in some other person?"* She realizes that the automation of the whole process and the adaptability of the "command" used to solve this task toward another (similar) task are crucial. Fortunately, Valerie is aware of the SWGET portal! She visits the website and has a quick look at the *Learn* tab where she becomes familiar with the syntax of NAUTILOD. She is ready to create her first SWGET script. She retrieves the URI of TBL[10] from DBpedia (from where the navigation starts) and copies it in the *seed URI* text field. Then, she writes the following expression:

```
(dbpprop:influenced <1-6>)
[ASK{?p rdf:type dbpedia:Scientist.}]/
(owl:sameAs)*
```

This expression considers nodes linked to TBL (by influence relations) up to distance 6. On each node reached, a check is performed to see if it represents a scientist. The last part of the expression considers aliases of such scientists in other data sources.

10 http://dbpedia.org/resource/Tim_Berners-Lee.

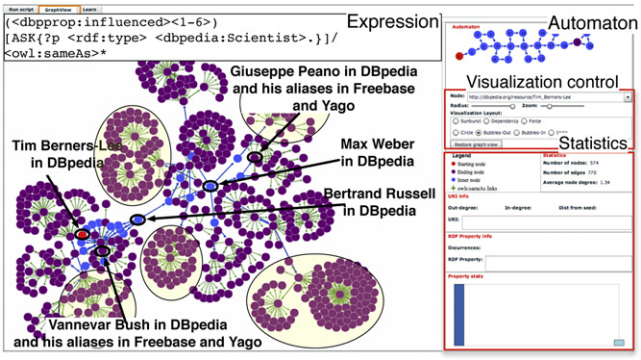


Fig. 5. The SWGET online portal graph visualization.

Valerie saves on her computer the script in RDF for future reuse, launches her agent and is given an *agent id*. Since she provided her email address she is notified when the agent terminates or in case of errors. The results of this script is shown in Fig. 5. Here, some nodes (scientists) belonging to the influence network of TBL are highlighted. Note that *Bertrand Russel* although being in DBpedia a philosopher (and not a scientist) is included in the graph with a different color. This is because he belongs to the influence-path connecting TBL to *Giuseppe Peano* (a scientist who belongs to the results). Clusters of nodes represent the aliases (discovered via `owl:sameAs`) of a node in other data sources.

Embedding in HTML pages. Valerie wants to maintain a Web (HTML) page with the pieces of information collected by executing the previous script. In order to do so, she can use a scripting language such as JavaScript. Moreover, she can launch the script on a regular basis and have always up to date information. Valerie passes the script to her friend Syd, who wants to compute the influence network of Alan Turing. Syd with a slight modification (he changes the seed URI) "centers" the information finding around Alan Turing.

6. Evaluation

We evaluated the portal, and in particular the SWGET engine at its core, on the current WLOD. The objective of the evaluation was to measure: (i) execution time; (ii) number of URIs dereferenced; (iii) number of results retrieved. In particular, we compared SWGET against a new implementation that leverages multi-threading (referred to as SWGETM), SQUIN and SPARQL 1.1's property paths (for which we considered the DBpedia SPARQL endpoint[11]). Further details about the experimental setting and the complete query-set are available at http://swget.wordpress.com/evaluation.

The experiments were performed on a PC with an Intel 2.5 GHz Core i5 processor and 8 GBs of RAM memory. Running times are the average of 5 runs; moreover, number of results and number of dereferenced URIs are rounded to the next integer value. The three experiments performed are: (*exp*1) TBL's influence network (by considering scientists); (*exp*2) FOAF network; (*exp*3) co-director at distance 6 (i.e., reachable via a chain of 12 `director` edges) and influence closure (also with tests). Moreover, *exp1* and *exp2* consider queries with an increasing distance from 1 to 6; this gives a total of 12 queries.

The execution times for *exp*1 are shown in Fig. 6(a). As it can be observed, there is a major improvement between SWGET and its multi-thread version SWGETM. As for SWGETM, we used a thread pool of size 5 to avoid many simultaneous connections requests that may overload servers and generate errors that would result in the lost of results. The improvement in performance for SWGETM is especially true for long queries; for instance, at *dist*6, SWGETM is ∼3 times faster than SWGET.

As compared to SQUIN, SWGETM reported a higher running time at *dist*6. However, by looking at Fig. 6(b) it can be noted that SWGETM performs a much larger number of dereferencing operations (433 vs. 117). This latter aspect had an impact on the number of (valid) results shown in Fig. 6(c). Indeed, SWGETM retrieved 56 results while SQUIN 18. The reason for these differences stems from the Link-Traversal Based Query Execution mechanism based on non-blocking iterators implemented by SQUIN [15]. This approach cannot guarantee to discover all reachable URIs that may contribute to the final results [15] (a more detailed comparison is provided in Section 2). On the other hand, NAUTILOD at the core of SWGETM guarantees that all the parts of the WLOD graph that may contribute to the results are visited. We also want to point out that with SQUIN it was not possible to consider (`owl:sameAs`)* at the end of the TBL's influence expression. To make it possible the comparison, we considered an additional expression (not reported here) where up to 3 levels of `owl:sameAs` are considered. The running times of SWGETM and SQUIN were comparable (∼45 s at distance 6 with 3 levels of `owl:sameAs`). However, the number of results provided by SWGETM was higher.

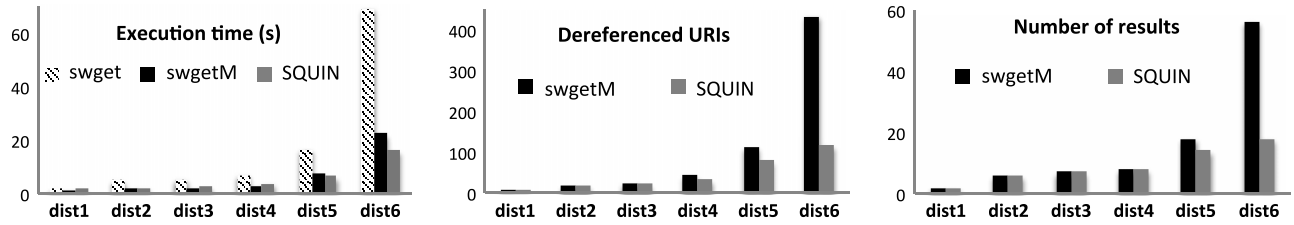
In *exp*2, we used A. Polleres' FOAF profile[12] and the `foaf:knows` predicate with an increasing distance from 1 to 6. The execution time for SWGET (single-thread) varies from ∼15 to ∼500 s. The number of results varies from 43 to 2.6 K. As for SWGETM (multi-thread), the running time varies from ∼2 to ∼60 s (a significant reduction of the running time). For this experiment the size of the thread pool has been increased to 35; this was possible because FOAF profiles are more evenly distributed over different servers and then the risk of overloading a single server with many HTTP requests is lower. As for SQUIN, the running time ranges from ∼4 to ∼250 s. Moreover, for the queries from *dist*2 to *dist*6 the system has thrown a concurrent exception although providing partial results. The running time of SWGET, SWGETM and SQUIN during the evaluation *exp*2 has been affected by several I/O exceptions due to missing FOAF profiles.

In *exp*3, we compared SWGETM with SPARQL 1.1's property paths (PPs) and SQUIN. It is crucial to note that PPs are a means to deal with paths that link RDF triples available in a *local* graph while SWGETM and SQUIN target the WLOD graph where paths exist between distributed (and a priori unknown) data sources. Nevertheless, we compared SWGETM against PPs on the DBpedia SPARQL endpoint (DBse). We first consider the expression: (`dbprop:influenced`)* by considering the URI of TBL in DBpedia as a seed node. Note that SQUIN cannot express the above request since it does not support PPs. And, even if this would be possible, PPs cannot be evaluated over the WLOD graph.

For the previous query, DBse did not provide any result because of a memory overflow after ∼2 min while SWGETM provided 1792 results in ∼172 s. We also compared on another expression (i.e., co-director of S. Kubrick at *dist*6) PPs, SWGETM and SQUIN. This expression took ∼50 s on DBse, ∼180 s on SWGETM while we stopped the execution of SQUIN after ∼2 h. It is interesting to observe that DBse uses a local triplestore while SWGETM and SQUIN work on the Web. Indeed, SWGETM performed a total of ∼3400 dereferencing operations. Finally, we also considered the expression (`dbprop:influenced[ASK ?x rdf:type dbpedia:Scientist]`)* with TBL as a seed node. Note that this request cannot be expressed neither by PPs nor by SQUIN. SWGETM provided 8 results with 21 dereferencing operations in ∼2.5 s.

11 http://dbpedia.org/snorql/.

12 http://www.polleres.net/foaf.rdf#me.



(a) Execution time(s) — *exp1*. (b) Number of URI dereferenced — *exp1*. (c) Number of results — *exp1*.

Fig. 6. Comparison between SWGET single thread, SWGET multi-thread (SWGETM) and SQUIN.

7. Discussion

User audience. The audience of SWGET are Web developers who want to get fresh data directly from the source. The portal Website includes a *Learn* tab where a video is provided, which describes the step for launching SWGET agents and writing NAUTILOD expressions.

Writing expressions. To use SWGET, the user needs some familiarity with the underlying data/schema; similar to other query/navigational languages (e.g., SPARQL). To assist the user in writing expressions one could leverage metadata about datasets (e.g., VoID descriptions [17]). An increasing level of difficulty is represented by the fact that NAUTILOD expressions can span over multiple data sources/schemata. To address this issue, users could write NAUTILOD expressions adopting a meta-schema such as `schema.org` because there exist mappings with other schemata.[13] Hence, NAUTILOD users have only to learn one schema; indeed, expressions written in `schema.org` will be dynamically translated into the schemata of the data sources encountered during the navigation. NAUTILOD could also perform RDFS inference. For instance, if the user chooses a property p_1, NAUTILOD could consider its sub-properties (e.g., p_2 `rdfs:subPropertyOf` p_1) by navigating the schema. Another line of extension could be the interaction of SWGET with engines like Sindice [10].

Performance. SWGET expressions are evaluated over the WLOD without the need of SPARQL endpoints. We want to point out that the main idea underlying the portal (with the notification mechanism via email) is that of freeing the user from the burden to run scripts (with local resources) and wait for the answer. Performance (in terms of execution time) depends on different factors: (i) load of the servers (and polite crawling); (ii) quality of the Internet connection; (iii) use of cache; (iv) complexity of the expression. As for point (iii), SWGET can be configured to use a cache that keeps Jena models associated to dereferenced URIs. The cache has the lifetime of the evaluation of an expression. The cache could be made also persistent, for instance, via a hashtable-like structure with keys being dereferenced URIs and values the corresponding Jena models. This will improve the runtime performance of the system avoiding to multiple dereferencing operations for the same URI. However, the cache introduces problems of data freshness (e.g., how often the cache has to be updated). This issue can be addressed either by setting a cache lifetime or by looking at the HTTP header of HTTP connections when dereferencing a URI and getting data only if changes are detected w.r.t. data in the cache. We are also considering the deployment of the portal on a cloud environment to improve its scalability. As for point (iv), the availability of statistics about the usage of predicates can help in determining their selectivity; this will help in estimating the cost of the evaluation of NAUTILOD expressions. We want to point out that the SWGET engine can be configured (by passing a flag) to stream results as they are available. Hence, for queries that may take long, users can stop the execution and still get some results. Finally, the system also enables to specify a budget in terms of maximum number of dereferencing operations and stop the execution when reaching the budget.

[13] http://schema.rdfs.org/mappings.html.

8. Concluding remarks

In this paper we introduced the SWGET portal, an on-line application that enables to create *Web agents*. These agents can be instructed by means of the NAUTILOD language. NAUTILOD provides a declarative way to express navigational charts into the Web of Linked Open Data graph. Navigation has been deeply studied into the context of graphs; however, NAUTILOD challenges navigation in a context (the Web of Data) where the graph is not known in its entirety. The SWGET Web portal mimics the process of instructing, with a semantic specification given in the NAUTILOD language, a personal agent who traverses semantic data sources looking for relevant information and provides notifications. Although the ambitious dream of the intelligent agent envisioned in the Semantic Web proposal is still far from reality, SWGET gives a hint on the potentialities of declaratively specifying navigation and triggering actions at a Web scale.

References

[1] T. Heath, C. Bizer, Linked data: evolving the web into a global data space, in: Synthesis Lectures on the Semantic Web: Theory and Technology, Morgan & Claypool Publishers, 2011.
[2] G. Klyne, J. Carrol, B. McBride, Resource description framework (RDF): concepts and abstract syntax, 2004.
[3] E. Prud'hommeaux, A. Seaborne, SPARQL query language for RDF, W3C recommendation, 2008.
[4] P.T. Wood, Query languages for graph databases, SIGMOD Rec. 41 (1) (2012) 50–60.
[5] T. Berners-Lee, J. Hendler, O. Lassila, The Semantic Web, Scientific American, 2001.
[6] V. Fionda, C. Gutierrez, G. Pirrò, Semantic navigation on the web of data: specification of routes, web fragments and actions, in: Proc. of the 21st World Wide Web Conference, WWW, 2012, pp. 281–290.
[7] F. Alkhateeb, J.-F. Baget, J. Euzenat, Extending SPARQL with regular expression patterns (for querying RDF), J. Web Semant. 7 (2) (2009) 57–73.
[8] J. Pérez, M. Arenas, C. Gutierrez, nSPARQL: a navigational language for RDF, J. Web Semant. 8 (4) (2010) 255–270.
[9] R. Isele, A. Harth, J. Umbrich, C. Bizer, LDspider: an open-source crawling framework for the Web of Linked Data, in: Proc. of the 9th International Semantic Web Conference, ISWC, Posters & Demo, 2010.
[10] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, G. Tummarello, Sindice.com: a document-oriented lookup index for open linked data, Int. J. Metadata, Semant. Ontol. 1 (3) (2008) 37–52.
[11] M. d'Aquin, E. Motta, Watson, more than a semantic web search engine, Semant. Web 1 (2) (2011) 55–63.
[12] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, J. Umbrich, Data summaries for on-demand queries over linked data, in: Proc. of the 19th WWW Conference, 2010, pp. 495–544.
[13] B. Quilitz, U. Leser, Querying distributed RDF data sources with SPARQL, in: Proc. of the 5th European Semantic Web Conference, ESWC, 2008, pp. 524–538.

[14] S. Harris, A. Seaborne, SPARQL 1.1 Query Language, W3C Recommendation, 2013.
[15] O. Hartig, Zero-knowledge query planning for an iterator implementation of link traversal based query execution, in: Proc. of the 8th European Semantic Web Conference, 2011, pp. 154–169.
[16] V. Fionda, C. Gutierrez, G. Pirrò, Extracting relevant subgraphs from graph navigation, in: Proc. of the 11th International Semantic Web Conference, ISWC, Posters & Demos, 2012.
[17] K. Alexander, R. Cyganiak, M. Hausenblas, J. Zhao, Describing linked datasets with the VoID vocabulary, 2011.