Semantic Web 0 (0) 1
IOS Press

SPARKLIS: An Expressive Query Builder for SPARQL Endpoints with Guidance in Natural Language

Editor(s): Eero Hyvönen, Aalto University, Finland
Solicited review(s): Norbert E. Fuchs, University of Zurich, Switzerland; Vanessa Lopez, IBM; Eetu Mäkelä, Aalto University, Finland

Sébastien Ferré *,**
IRISA, Université de Rennes 1, campus de Beaulieu, 35042 Rennes cedex, France

Abstract. SPARKLIS is a Semantic Web tool that helps users explore and query SPARQL endpoints by guiding them in the interactive building of questions and answers, from simple ones to complex ones. It combines the fine-grained guidance of faceted search, most of the expressivity of SPARQL, and the readability of (controlled) natural languages. No knowledge of the vocabulary and schema are required for users. Many SPARQL features are covered: multidimensional queries, union, negation, optional, filters, aggregations, ordering. Queries are verbalized in either English or French, so that no knowledge of SPARQL is ever necessary. All of this is implemented in a portable Web application, SPARKLIS, and has been evaluated on many endpoints and questions. No endpoint-specific configuration is necessary as the data schema is discovered on the fly by the tool. Online since April 2014, thousands of queries have been formed by hundreds of users over more than a hundred endpoints.

Keywords: semantic search, SPARQL endpoint, query builder, faceted search, natural language

1. Introduction

A wealth of semantic data is accessible through SPARQL endpoints. DBpedia alone contains several billions of triples covering all sorts of topics (e.g., people, places, buildings, species, films, books). Although different endpoints may use different vocabularies and ontologies, they all share a common interface to access and retrieve semantic data: the SPARQL query language [24]. In addition to being a widely-adopted W3C standard, the advantages of SPARQL are its *expressivity*, especially since version 1.1, and its *scalability* for large RDF stores thanks to highly optimized SPARQL engines (e.g., Virtuoso, Jena TDB). Its main drawback is that writing SPARQL queries is a tedious and error-prone task, and is largely unaccessible to most potential users of semantic data.

Our motivation in developing SPARKLIS[1], shared by many other developers of Semantic Web tools and applications, is to unleash access to semantic data by making it easier to define and send SPARQL queries to endpoints. The novelty of SPARKLIS is to combine in an integrated fashion different search paradigms: Faceted Search (FS), Query Builders (QB), and Natural Language Interfaces (NLI). That integration is the key to reconcile properties for which there is generally a trade-off in existing systems: user guidance, expressivity, readability of queries, scalability, and portability to different endpoints.

Section 2 discusses related work on making semantic search more usable. Section 3 presents the principles and architecture of SPARKLIS, and Section 4 illustrates them on a concrete navigation scenario on DBpedia. Section 5 precisely states the capabilities and

*Corresponding author. E-mail: ferre@irisa.fr.
**This work is supported by ANR project IDFRAud.

[1] Online at `http://www.irisa.fr/LIS/ferre/sparklis/`

limitations of SPARKLIS, and Section 6 reports experimental evaluation and impact on real usage worldwide. Finally, Section 7 concludes and draws perspectives. This paper focuses on the functional and non-functional properties of SPARKLIS, as perceived by users. Technical details about its internal working can be found in previous papers [6,5].

2. Related work

There are mainly two approaches to make semantic search more usable: user interaction (UI) and natural language (NL). UI-based systems reuse and adapt UI paradigms to semantic data: hypertext browsing (e.g., Fluidops Information Workbench[2]), query builders (e.g., SemanticCrystal [15]), faceted search (FS) [23], or OLAP [3]. Query builders generally offer more *expressivity*, but lack *readability* because they are based on formal languages. Moreover, their guidance is mostly based on syntax, and sometimes on a data schema, but not on actual data, like in FS. Most FS-based systems do not claim for a contribution in term of *expressivity*, and contribute either to the design of better interfaces and visualizations, or to methods for the rapid or user-centric configuration of faceted views: e.g., Ontogator [18], mSpace[3], Longwell[4]. Similarly, OLAP-based systems emphasize visualization, and require substantial amount of configuration to extract cubic views over RDF graphs: e.g., Cubix [21], Linked Data Query Wizard [12]. Therefore, their contributions are somewhat orthogonal to ours, and could certainly complement them. A few FS-based systems extend faceted search *expressivity*: e.g., SlashFacet [11], BrowseRDF [22], gFacet [9], VisiNav [8], SemFacet [1], OpenLink FS[5], Vinge Query&Explore[6]. While more expressive than classical FS, those systems are still much less expressive than SPARQL 1.1, and approximately cover basic graph patterns. None of them support union, negation, or aggregation. All except Vinge Query&Explore present only lists of results, rather than tables. That expressivity is reflected by the frequent choice to use trees and graphs to represent the query. Those representations have a good match with SPARQL graph patterns, but do not scale well to express union, negation, or aggregations, unlike natural language.

Natural Language Interfaces (NLI) [16] use NL in various forms, going from full natural language (e.g., PowerAqua [17]) to mere keywords (e.g., NLP-Reduce [15]). In between, there are also controlled natural languages (e.g., Ginseng [15], SQUALL [4]). Systems based on full NL or keywords devote most of their effort to bridging the gap between lexical forms and ontology triples (mapping and disambiguation), and process only the simplest questions, i.e., they generate SPARQL queries with only one or two triples. Most of them support none of aggregations (e.g., counting), comparatives, or superlatives, even though those features are relatively frequent (see QALD-3 challenge [2]).

Some systems integrate the UI approach in NLIs to alleviate the *habitability problem* [15], in which users have not a precise knowledge about what can be understood by the NLI system, and therefore can be frustrated by syntax errors or empty results. Ginseng [15] uses auto-completion based on the grammar and an ontology. Atomate [25] uses a Controlled Natural Language (CNL) and dynamic forms to guide users in the definition of reactive rules. Those systems can be seen as query builders based on a controlled natural language. They improve the former with readability, and the latter with guidance, but they still lack the fine-grained guidance of FS that is necessary to fully solve the habitability problem.

In our objective to reconcile guidance and expressivity, we wanted to avoid some unsatisfactory effects that generally result from the loose integration of two existing approaches. First, we wanted guidance from the beginning, rather than asking users to start with a category (e.g., BrowseRDF), an entity (e.g., Vinge Query&Explore), or a keyword search (e.g., OpenLink FS). Guidance from the beginning avoids the *writer's block* (blank field with no suggestion), and initial suggestions provide an overview over the whole dataset to users, which is especially valuable when the dataset is unknown. Second, we wanted to directly explore RDF graphs, rather than first extracting a dataset on which standard techniques can then be applied, e.g. faceted search on extracted facets over a selected collection of items (e.g.. Ontogator), OLAP on an extracted data cube (e.g., Linked Data Query Wizard, Cubix). The latter approaches generally require an expert for the extraction part, and strongly limit expressivity for end users. Finally, we want to avoid the *habitability problem* [15], found in NLIs, in which users have not a pre-

[2] http://iwb.fluidops.com/
[3] http://mspace.fm/
[4] http://simile.mit.edu/wiki/Longwell
[5] http://dbpedia.org/fct/facet.vsp
[6] http://www.vingefree.com/querybyexplore/

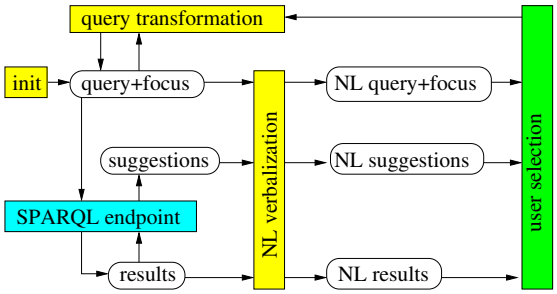


Fig. 1. The system architecture of SPARKLIS

cise knowledge about what can be understood and answered by the system, and therefore can be frustrated by syntax errors or empty results.

3. Principles and architecture

This section explains how SPARKLIS integrates the principles of Faceted Search (FS), Query Builders (QB), and Natural Language Interfaces (NLI). That integration is summarized by Figure 1, which shows the system architecture of SPARKLIS. The state of the system is determined by the *query* and *focus*, where the focus refers to an entity in the query, and is used as an insertion position for applying query transformations. The SPARQL endpoint is used to compute *results* from the query, and to compute a set of *suggestions* from the query and results. Each suggestion is a query element that can be inserted at the focus to refine the query: i.e. an entity, a class, a property, or an operator (e.g., an aggregation). The query, suggestions, and results are verbalized in NL for rendering to the user in a readable form. The user can select a suggestion, or activate a control in the query or results, to trigger a query transformation, and reach a new system state.

SPARKLIS re-uses and generalizes the interaction model of FS [23], where users are *guided step-by-step* in the selection of items. In FS, at each step, the system gives a set of suggestions to refine the current selection, and users only have to pick a suggestion according to their preferences. The suggestions are specific to the selection, and therefore support exploratory search [19] by providing overview and feedback during the search process. To overcome *expressivity* limitations of FS, and of existing FS extensions for the Semantic Web (e.g., gFacet [9], VisiNav [8], SemFacet [1]), we have generalized it to *Query-based Faceted Search (QFS)* [6], where the selection of items is replaced by a structured query. The latter is built step-by-step through the successive choices of the user. This makes SPARKLIS a kind of Query Builder (QB), like SemanticCrystal [15]. QBs have the advantage to allow for a high expressivity while assisting users about syntax by listing eligible constructs at each step. In particular, this enables to completely avoid syntax errors. However, the FS-based guidance of SPARKLIS is more fine-grained than in QBs. In fact, SPARKLIS only allows the building of queries that *do* return results, preventing users to fall on empty results. Note that non-empty results necessarily imply no syntax error, and no vocabulary error as a consequence. That is because system suggestions are computed for the individual results, not for their common class. In fact, SPARKLIS is as much about building answers as about building questions: e.g., adding columns to the table of results by selecting a property, filtering out rows by selecting a value.

To overcome the lack of *readability* of SPARQL queries for most users, SPARKLIS queries, suggestions, and results are verbalized in natural language so that SPARQL queries and variables never need to be shown to users (see [5] for more details). This makes SPARKLIS a kind of Natural Language Interface (NLI), like PowerAqua [17]. The important difference is that questions are built through successive user choices in SPARKLIS instead of being freely input in NLIs. SPARKLIS interaction makes question formulation more constrained, slower, and less spontaneous, but it provides guidance and safeness with intermediate answers and suggestions at each step. Moreover, it avoids the hard problem of NL understanding: i.e., ambiguities, out-of-scope questions. A few NLI systems, like Ginseng [15], are based on a controlled NL and auto-completion to suggest the next words in a question. However, their suggestions are not fine-grained like with FS, and less flexible because they only apply to the end of the question. In SPARKLIS, questions form complete sentences at any step of the search; and suggestions are not words but meaningful phrases (e.g., **that has a director**), and can be inserted at any position in the current question. That current insertion position is called *focus*, and suggestions are specific to each insertion position.

In order to address *scalability* issues, only a limited number of results are retrieved from the query, and only a limited number of suggestions are computed from the partial results. No ranking is applied to either results or suggestions because it would defeat scalability. Indeed, ordering SPARQL results forces the query engine to iterate over all solutions. How-

Table 1
Navigation scenario in SPARKLIS over DBpedia

Step	Query
1	Give me something
2	Give me **a writer**
3	Give me a writer **that has a nationality**
4	Give me a writer that has nationality **Russians**
5	Give me a writer that has nationality Russians **and that has a birth date**
6	Give me a writer that has nationality Russians and whose birth date is **after 1800**
7	Give me a writer that has nationality Russians and whose birth date is after 1800 **and that is the author of something**
8	Give me a writer that has nationality Russians and whose birth date is after 1800 and that is the author of **a book**
9	Give me a writer that has nationality Russians and whose birth date is after 1800 and that is the author of **a number of** book
10	Give me a writer that has nationality Russians and whose birth date is after 1800 and that is the author of **the highest-to-lowest** number of book
11	Give me a writer that has nationality Russians **or something** and whose birth date is after 1800 and that is the author of the highest-to-lowest number of book
12	Give me a writer that has nationality Russians or **Russia** and whose birth date is after 1800 and that is the author of the highest-to-lowest number of book

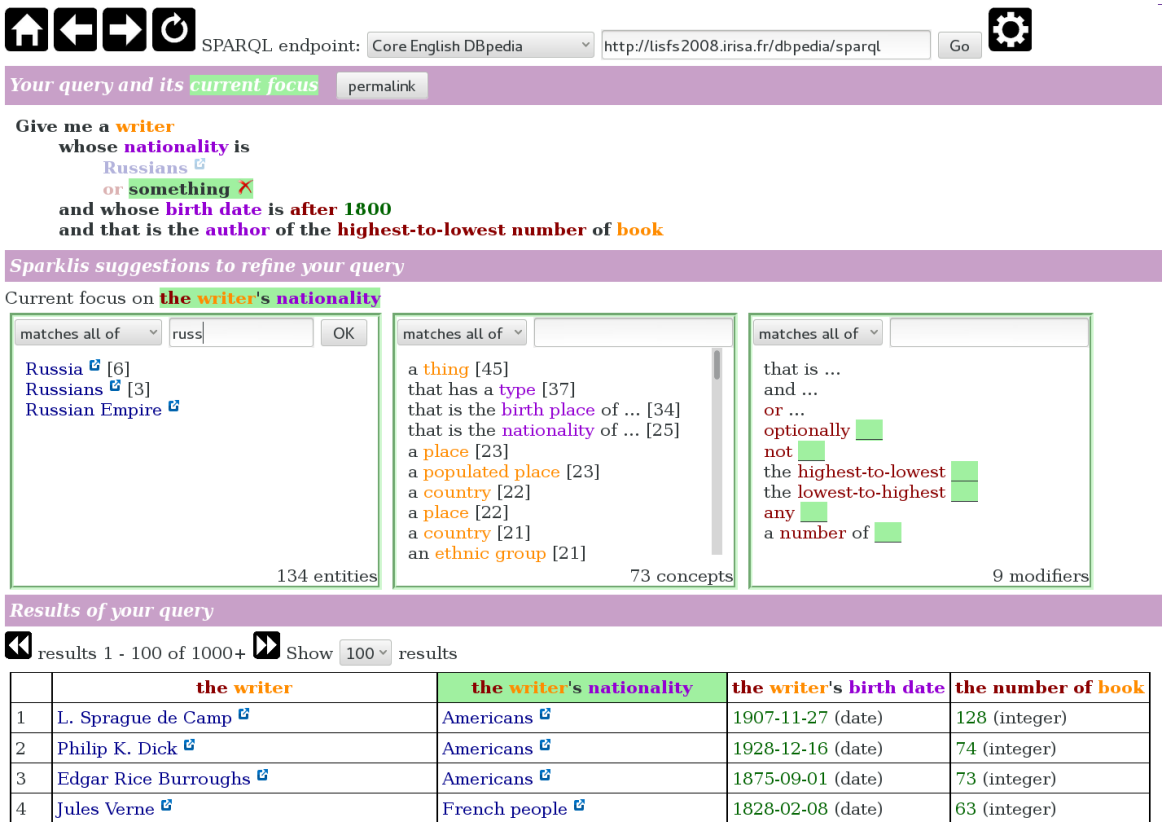
ever, a SPARQL engine may exploit an internal ranking to produce best-ranking solutions first. Now, even without ranking, the most frequent classes and properties are more likely to belong to the selected suggestions, and this generally produces good enough selections. The negative consequence of partial results and suggestions is that suggestions may be incomplete, making some queries unreachable. To restore some form of completeness, we have designed an *intelligent auto-completion*. Auto-completion is a well-known user interface mechanism that provides guidance and feedback, and has already been adapted to semantic contexts [14,7]. SPARKLIS auto-completion is directly available at the top of each suggestion list, and dynamically filters suggestion lists at each keystroke for immediate feedback. It is intelligent in that it uses a cascade of three stages to ensure completeness relative to user input. At stage 1, the partial list of suggestions is filtered on the client side, which can be done efficiently. At stage 2, if the filtered list gets empty, the list of suggestions is re-computed by sending to the SPARQL endpoint a new query that includes the user filter. This means that the same partial query results are used, but a constraint is put on the expected classes and properties. At stage 3, when the filtered list is still empty, new queries are again sent to the SPARQL endpoint, using the full SPARQL query instead of the partial results, in addition to the user filter. This ensures that all query results are used in the computation of suggestions.

To promote *portability*, SPARKLIS is entirely based on Web standards. It uses SPARQL endpoints for RDF storage and querying, HTTP requests to query them, JavaScript (JS) for the application code, and HTML5/CSS3 for the user interface. SPARKLIS only needs the URL of an endpoint to explore it, without any further required configuration. Users can however customize the number of results to be retrieved, or the labelling properties to verbalize URIs. Queries to SPARQL endpoints are sent directly from the client browser, using AJAX requests. It makes SPARKLIS independent from a server, hence trivial to deploy, and efficient because all application code runs on the client. When a SPARQL endpoint does not enable cross-domain AJAX requests, another endpoint can be used as a proxy, based on the `SERVICE` feature of SPARQL. For code safety and development speed, the JS code is compiled from a high-level strong-typed language (OCaml using `js_of_ocaml`[7]). The source code counts about 5000 lines of code, and the minimized JS code weights about 250kB.

4. Navigation scenario on Core DBpedia

We detail in this section a navigation scenario using SPARKLIS on Core English DBpedia, the subset of DBpedia limited to classes and properties of the DBpedia ontology, and to English labels. The same scenario can be played on the full DBpedia but with more noisy suggestions because it includes all data from Wikipedia infoboxes. When SPARKLIS is loaded,

[7] `http://ocsigen.org/js_of_ocaml/`



	the writer	the writer's nationality	the writer's birth date	the number of book
1	L. Sprague de Camp	Americans	1907-11-27 (date)	128 (integer)
2	Philip K. Dick	Americans	1928-12-16 (date)	74 (integer)
3	Edgar Rice Burroughs	Americans	1875-09-01 (date)	73 (integer)
4	Jules Verne	French people	1828-02-08 (date)	63 (integer)

Fig. 2. SPARKLIS screenshot at step 11 of scenario in Table 1

the Core English DBpedia is the default endpoint, and users can switch to DBpedia, Live DBpedia, or a few other predefined endpoints. Users can also enter the URL of the endpoint of their choice (see at the top of Figure 2).

Table 1 shows the successive queries, as verbalized in SPARKLIS, of a navigation scenario that leads the user in 12 steps to a list of "Russian writers born since 1800, and ordered by decreasing number of written books". That scenario is only one of several possible scenarios leading to the same results: e.g., the birth date could have been constrained before the nationality. At each step, the bold part represents the newly inserted query element, chosen by the user at the previous step among system suggestions, and the underlined part represents the query focus that is used for the next query transformation. The query focus is moved simply by clicking on different parts of the query. The query elements that are suggested for insertion at query focus can be entities (e.g., **Russians**), classes (e.g., **a book**), properties in both directions (e.g., **is the author of**, **has birth date**), filters (e.g., **after 1800**), and various modifiers (e.g., **number of**, **or**).

Figure 2 is a SPARKLIS screenshot at step 11, during the specification of an alternative nationality for the writer (**Russia** as a synonym nationality of **Russians**). The user interface is made of three parts: top, middle, bottom. The top part shows the current query, and highlights the current focus, here **something**. The first branch of disjunction is transparent to reflect the fact that it is ignored during the construction of the second branch. The bottom part is the result table of the current query, with a column for each entity/value in the query (here: writer, nationality, birth date, and number of books). The focus column, here the nationality, is highlighted. The middle part contains relevant query elements for insertion at the query focus. It is split in three lists. The first list contains entities (URIs) and values (literals) found in the focus column. It also enables the construction of filters over values. The second list contains *concepts* (classes and properties) that apply to entities/values found in the focus column. The third list contains modifiers that are applicable to the

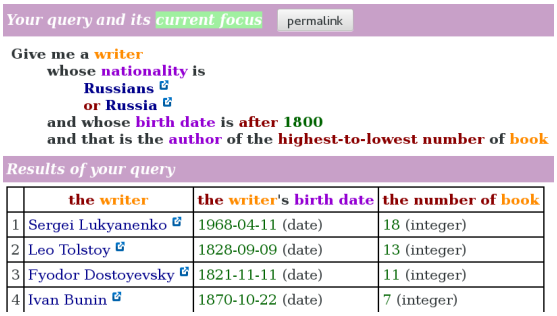


Fig. 3. SPARKLIS screenshot at final step 12, hidding suggestions

```
PREFIX n1: <http://dbpedia.org/ontology/>
PREFIX n2: <http://dbpedia.org/resource/>
SELECT DISTINCT ?Writer_1 ?birthDate_3
    (COUNT(DISTINCT ?Book_4) AS ?number_of_Book_5)
WHERE {
    ?Writer_1 a n1:Writer .
    { ?Writer_1 n1:nationality n2:Russians . }
    UNION { ?Writer_1 n1:nationality n2:Russia . }
    ?Writer_1 n1:birthDate ?birthDate_3 .
    FILTER ( str(?birthDate_3) >= "1800" )
    ?Book_4 a n1:Book .
    ?Book_4 n1:author ?Writer_1 . }
GROUP BY ?Writer_1 ?birthDate_3
ORDER BY DESC(?number_of_Book_5)
LIMIT 1000
```

Fig. 4. The SPARQL translation of the query at step 12

query focus, such as Boolean connectors, aggregation operators, and ordering. Each list provides filtering for quickly locating a query element, and auto-completion for retrieving more matching elements from the endpoint. In addition to selecting a query element to insert it at query focus, the query part under focus can be deleted by clicking the red cross in the query.

Figure 3 is another SPARKLIS screenshot at final step 12. The current focus and suggestions have been hidden by clicking on the query head "Give me", in order to emphasize the query results. Figure 4 shows the SPARQL translation of the current query, which is sent to the endpoint, and is displayed at all times at the bottom of the page for the curious users or for reuse in other Semantic Web tools. Note that the layout, URI abbreviations, and variable names have been designed so as to make those SPARQL queries more human-readable. When users want to come back frequently to the current query, they can get a *permalink* to reach the current query and results in one click (see button above the query in Figure 3). Entities and values in the table of answers can be inserted in the query, at the focus corresponding to their column. For example, selecting the birth date **1968-04-11** in Figure 3 would replace the constraint **after 1800** by the value **1968-04-11**. It also works for aggregated values, so that retrieving all writers having written a given number of books is possible.

5. Capabilities and limitations

We here detail the capabilities and limitations of SPARKLIS w.r.t. five important properties for user experience: expressivity, guidance, readability, scalability, and portability.

5.1. Expressivity: large subset of SPARQL 1.1

SPARKLIS is concerned with the `SELECT` and `ASK` forms of SPARQL queries. It covers many features of SPARQL: basic graph patterns, including cycles; simple filters on strings, numbers, dates and times, language tags, and datatypes; `UNION` patterns; `OPTIONAL` patterns; `NOT EXISTS` constraints; `ORDER BY` clauses; multiple aggregations with `GROUP BY` and `HAVING` clauses. Triple patterns with a variable in predicate position are also covered through the suggestions **that has a relation to ...** and **that has a relation from ...**, where **a relation** stands for a property placeholder. The focus can then be put on it, and all kinds of constraints can be applied to it through SPARKLIS navigation. Blank nodes are correclty handled. They are shown in results, and suggestions are given about them, but they cannot be inserted in the query.

From a linguistic point of view, SPARKLIS verbalized queries include: noun phrases, verb phrases, and relative clauses for graph patterns; coordinations for relational algebra (`UNION`, `NOT EXISTS`, `OPTIONAL`); superlatives for ordering; determiners for aggregations; prepositions for some filters; and anaphoras for cycles. All 100 training questions over DBpedia of the QALD-3 challenge [2] fall in the expressivity scope of SPARKLIS. In practice, 9 questions could not be built because they are "out of scope", i.e. have no answers in DBpedia.

SPARKLIS does not address SPARQL updates, nor queries that return graphs (`CONSTRUCT`). We have proposed a guidance for those, based on query relaxation [10], but it cannot be made scalable on top of a SPARQL endpoint, unless some kind of `RELAX` operator is added to the SPARQL language [13]. The SPARQL query features that are not yet covered are: `GRAPH` patterns to select named graphs; nested queries, and in particular nested aggregations; expres-

sions to perform computations; and transitive closures of property paths. Nested queries are useful for doing analytics, i.e. to answer questions like "Give me the *average number of* children *per woman*, in *each country*" or "Give me *the number of* countries *per number of* official languages". They are difficult because, to avoid ambiguity, one has to specify for each aggregator its nesting level, and its grouping variables (e.g., "per woman", "for each country"). Expressions are also useful for analytical queries, as well as for spatio-temporal queries (e.g., computing distances and durations). For alll uncovered features, it is difficult to make their NL verbalization readable, especially transitivity. We think that transitivity may be best handled by defining new properties with rules: e.g., "An ancestor of a person is a parent of the person or an ancestor of a parent of the person".

5.2. Guidance: safeness and completeness

There are two important properties for guidance in query building: safeness and completeness. A *safe* guidance avoids dead-ends (i.e., empty results) by providing only relevant suggestions, i.e. suggestions that match actual data. A *complete* guidance fulfills the expressivity potential by providing *all* relevant suggestions. In a previous work [6], we formally proved the theoretical safeness and completeness of Query-based Faceted Search (QFS), on which SPARKLIS is founded. For scalability reasons, only a subset of results and suggestions might be computed and displayed in SPARKLIS. Guidance completeness is restored through the intelligent auto-completion mechanism (see Section 3). The query focus plays a crucial role in combining high expressivity (see Section 5.1) and guidance completeness. It determines an insertion position, and the suggestions that are relevant to it. It also provides flexibility in the query construction process, allowing query elements to be inserted in many orderings. When the focus is in the scope of an *opaque* operator (e.g., negation, aggregation), that operator is temporarily ignored in the translation of the query in SPARQL so that results and suggestions can be computed.

The main limitation to guidance completeness lies at the lexical level. If a suggestion (e.g., property **spouse**) is out of the selected subset, and the user uses auto-completion with a synonym (e.g., "married with"), then the desired query becomes unreachable from the point of view of the user. The schema can have a similar impact when there are several possible representations: e.g., a class **is a German** vs a property and value **has nationality Germany**. For best experience, we encourage data designers to limite the number of classes and properties, e.g., by factorizing classes like **AmericanChristianScientists** into descriptions like **a person whose occupation is Scientist and whose nationality is America and whose religion is Christianism**; and we encourage users to start by freely exploring an unfamiliar dataset in order to get accustomed to its schema and vocabulary.

Another limitation in guidance is that, for a given query, it gives the same suggestions in the same order to everybody, ignoring any user preferences or past usage. This can be a difficulty when the vocabulary is large, like in DBpedia. One could imagine that suggestions that have been selected by the user (or another user of his community) in similar circumstances were ranked higher in the list of suggestions.

5.3. Readability: verbalization in natural language

All content elements of the user interface of SPARKLIS are verbalized in natural language for readability by users unaware of RDF and SPARQL: the query, the suggestions, the column headers, and the entities in the table of answers. By default, URIs are verbalized by uncamelizing their local name (e.g., `dbo:birthDate` ⇝ **birth date**), or by replacing underscores by spaces (e.g., `dbr:Barack_Obama` ⇝ **Barack Obama**). Alternately, users can specify in the configuration panel a property, typically `rdfs:label`, and a language tag to retrieve URI verbalizations from the endpoint. Class and property local names or labels are assumed to be nouns, but a few common patterns are taken into account: e.g., `hasParent` ⇝ **parent**, `partOf` ⇝ (inverse of) **part**.

To verbalize queries, each query construct is first verbalized locally using a fixed syntactic pattern, and syntactic transformations are then applied globally to make the verbalization look more natural. For example, the first verbalization **that has a birth date that is after 1800** is transformed into **whose birth date is after 1800**. In case of cyclic graph patterns, it is necessary for one part of the query to refer to another part of the query. In order to completely avoid the use of unnatural variables, we verbalize variables as anaphoras. For example, in the SPARKLIS query **Give me a film whose director is an actor of the film**, the noun phrase **the film** is implicitly the SPARQL variable used for **a film**. The same variable verbalizations are used to label column headers. SPARKLIS so far offers verbalization in both English and French, and it should be relatively easy to extend

it to other languages that have the same syntactic categories, and no declensions (e.g., Spanish, Italian).

It is certainly possible to improve verbalization at the syntactic level, but the main room for improvement lies at the lexical level. For a real improvement, it is necessary to know for each URI all its possible lexical forms, along with their syntactic category (e.g., noun, verb), number (singular/plural), and gender when relevant. The LEMON vocabulary [20] has been defined to that purpose, but lexicons are rarely available. We think that SPARQL endpoint should include their ontology and lexicon so that lexical information can be retrieved along with data, and so that better readability does not come at the price of lesser portability.

5.4. Scalability: billions of triples

SPARKLIS is responsive on the largest well-known endpoint, DBpedia, which has a few billions of triples. For a language as expressive as SPARQL, short response times cannot be guaranteed, but from our experience, time-outs are rarely encountered. Among the 91 QALD-3 questions having answers, half can be answered in less than 30s, and the worst one required 109s (wall-clock time including user interaction and system computations but not user thinking). Only one question could not be answered because of a time-out when searching for a YAGO class by auto-completion.

In fact, the scalability of SPARKLIS is mostly limited by the scalability of the endpoint, because SPARQL requests account for most of the computation time. In particular, the lack of a standard way to perform state-of-the-art full-text search over resources, rather than using `FILTER`, is an important limitation for efficient auto-completion. Note that it may also be possible to optimize SPARQL engines for the particular kind of queries used by SPARKLIS, especially in the case where many users use it on a same endpoint. So far, federated search is not supported by SPARKLIS, i.e. a single endpoint can be queried at any given time. Distributed evaluation of queries over several endpoints, although supported by the `SERVICE` feature of SPARQL, raises the scalability issue one order higher.

5.5. Portability: no required configuration

SPARKLIS conforms to the SPARQL standard, and requires no preprocessing or configuration to explore an endpoint. It entirely relies on the endpoint to discover data and its schema. However, a few settings can be configured to adapt to differences between endpoints. When an endpoint does not allow cross-domain requests from Web browsers (same-origin policy), a default endpoint located at our institution IRISA is used as a proxy. The use of a proxy can be deactivated, and another proxy endpoint can be specified. By default, results to SPARQL queries are cached for improved performance, but caching can be deactivated, for example in case of frequently changing data. The default limit for query results is 200 and can be changed to adapt to the responsiveness of the endpoint. Similarly for the number of suggested classes and properties. The default language of the interface and verbalization is English, and can be changed to French. By default, URIs are verbalized from their local names. For each of entities, classes, and properties, a label property and language tag can be specified for the verbalization of URIs. The latter is more demanding on the endpoint, and requires it to support `BIND` patterns.

The limits that we have encountered by trying SPARKLIS on many endpoints come either from the implementation of the endpoint, or from its contents. Some endpoints do not support HTTP requests with the POST method or do not support some essential SPARQL features, such as `UNION` or `BIND`. A number of endpoints only contain terminological knowledge, and no instances. We recall that SPARKLIS heavily relies on instances for guidance. Another limitation is when the endpoint does not apply RDFS or OWL inference, so that some suggestions are missing. From its client side, SPARKLIS has no easy way to compensate for this lack of inference.

6. Evaluation and impact

In previous papers [6,5], we have presented controlled experiments about the expressivity, scalability, and portability of SPARKLIS, and the guidance of query-based faceted search. The main results are recalled in Section 5. A user study in [6] has shown that students without any prior knowledge of semantic technologies and data were able to answer questions involving several classes and properties, disjunction and negation, tree-shape patterns, and cycles in patterns.

In this paper, we analyze real usage of SPARKLIS worldwide. SPARKLIS has been developed since December 2013, and was put online and publicized since April 2014. It continues to evolve – the last version

was released in September 2015 – but its user interface has remained very stable over time. Several people became regular users, and provided valuable feedback and suggestions. Important improvement came from needs expressed by those people, e.g., support of blank nodes in answers, use of labelling properties for verbalizing URIs. We collected an anonymous usage log[8] from 19/06/2014 to 16/08/2015, where entries are navigation steps. Each step is described by the timestamp, the endpoint URL, the current query, and the user IP. Because IPs are not reliable to identify users and sessions, especially from wireless connection, we also include since 29/10/2014 a session ID that is randomly generated when the application page is loaded. For privacy sake, the user can deactivate the logging (checkbox in the configuration panel), so that the following analysis may underestimate real usage.

As an additional proof of the capabilities of SPARKLIS, we want to emphasize that SPARKLIS itself was used to explore its own log, and to produce most results below. The log was translated into RDF, and loaded in a local SPARQL endpoint using the Fuseki server. The most useful features were filters, ordering, and aggregation. What we found missing sometimes are nested aggregations (e.g., the average number of steps per session), and the direct visualization of results by charts rather than numeric tables.

6.1. Global statistics

Over the 424 days of the log period, SPARKLIS has been loaded 7379 times, hence 17.4 times per day on average. Over the 291 days with session IDs, there have been 2970 sessions with at least one non-empty query, hence about 10 effective sessions per day on average. The total number of steps is 26739, hence 63 steps per day on average. Although only seven endpoints are suggested in SPARKLIS, 176 endpoints[9] have been explored with SPARKLIS. About 12000 different queries have been built overall, which means that queries are visited about twice on average. Finally, we have counted 950 different user IPs. This is a rough estimate of unique users because several people may hide behind a same IP (e.g., wireless networks), but it is consistent with the number of views of the YouTube demo (612 on 16/08/2015). It is noticeable that the latter was seen from 49 countries: France (175), United States (82), Germany (45), Italy (37), UK (25), Russia (23), etc. All those figures demonstrates that SPARKLIS is effectively usable and portable. Usage has been relatively regular over the log period with spikes when we gave presentation talks at various venues.

Table 2
Most popular endpoints (number of user IPs)

endpoint URL	#user
http://lisfs2008.irisa.fr/dbpedia/sparql	544
http://dbpedia.org/sparql	370
http://live.dbpedia.org/sparql	88
http://data.nobelprize.org/sparql	86
http://lod.euscreen.eu/sparql	40
http://rdf.insee.fr/sparql	33
http://cu.drugbank.bio2rdf.org/sparql	32
http://datos.bcn.cl/sparql	30
http://drugbank.bio2rdf.org/sparql	22
http://cu.kegg.bio2rdf.org/sparql	18

6.2. User statistics

In this section, we show and discuss the distribution of users according to several criteria: explored endpoints, built queries (size and used features), and number of navigation steps.

Endpoints. Table 2 shows a shortlist of the 10 endpoints (out of 176) that attracted the most users, as counted by the number of user IPs. The most popular endpoint is the default endpoint, the Core English DBpedia hosted at IRISA. It is closely followed by the reference DBpedia endpoint and its *live* version. Other popular endpoints cover various topics, in particular bioinformatics with many bio2rdf datasets, and are hosted in countries worldwide (e.g., Chile, Japan, France). There are also 19 endpoints with URL `http://localhost...`, demonstrating that people are using SPARKLIS for their own local datasets. The other way around, 312 users (33%) have explored more than one endpoint, 181 users (19%) three or more endpoints, and 2 users have explored 11 different endpoints. This demonstrates that SPARKLIS is used as a general purpose tool by a significant number of people. Those figures may be overestimated because of shared IPs, but they are consistent with the fact that many sessions (193, 6.5%), which are necessarily associated to a single user, explore several endpoints (up to 6) one after the other.

[8] Available online without user IPs, and restricted to DBpedia and bio2rdf endpoints at `http://www.irisa.fr/LIS/ferre/-pub/sparklis_querylog.dat`.

[9] We only count endpoints that were responsive, and for which at least one navigation step could be performed.

Fig. 5. Percentage of users building queries up to some size (truncated at size 13, maximum size is 29)

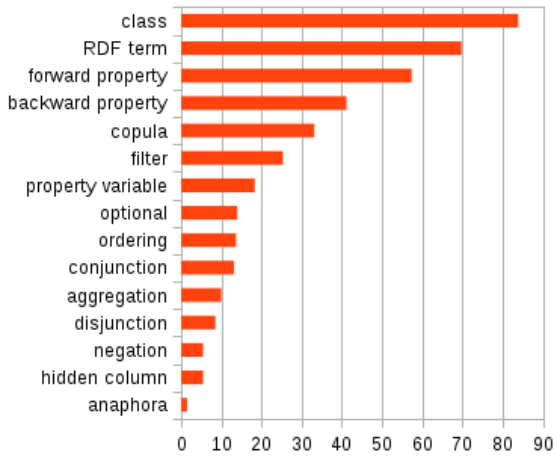


Fig. 6. Percentage of users building queries with some feature

Queries. Figure 5 shows for each query size the percentage of users who have managed to build a query with a size greater of equal. The size of the query is the number of query elements that must be selected to build the query, and is therefore the minimal number of navigation steps to reach it from the empty query (size 0). The query size ranges from 0 to 29, and most queries have a size under 10. The figure shows that about 50% of users have built a query with size 4 or higher, that still nearly 20% users have reached size 7, and 5% (50 users) have reached size 10. Note that 37 queries out of the 50 QALD-4 challenge have a size less or equal to 4, and the longest has size 8. Those figures demonstrate that most users understood how to interact with the tool, and managed to build non-trivial queries (beyond 1 or 2 elements). Here is an example of query with size 3 on DBpedia: **Give me a celestial body that is a galaxy and that has a ngc**. It retrieves galaxies along with their NGC number. Another more complex example with size 9 built on the bio2rdf endpoint by 11 different users: **Give me a drug that has a title and is the drug of a target-relation that has action inhibitor and whose target has a title and has a specific-function**. It returns not onlys drugs, but also the related targets and the specific-functions of each target. The query elements **that has a title** were not used to filter results but to get a readable label for drugs and targets as additional columns in the result table.

Figure 6 shows for each feature of SPARKLIS' query language the percentage of users that have used it at least once. The SPARQL counterpart of most features should be clear from their names. *Copula* (modifier **that is**) corresponds to starting a new constraint (triple pattern or filter) with an already used variable. *Property variable* (modifier **has a relation to/from**) corresponds to adding a triple pattern with a new variable in predicate position. *Hidden column* (modifier **any**) corresponds to omitting a variable in the SELECT clause. *Anaphora* corresponds to picking an already used variable. The 5 most popular features correspond to basic graph patterns in SPARQL, and make up the main stuff of queries. Then, we observe that filtering, ordering, and optional come next, and are more popular than logical operators and aggregations, which are used by less than 10% of users. *Hidden column* is hardly used because it is most useful with aggregation to specify groupings. *Anaphora* is also hardly used because anaphora suggestions can only occur when the query contains two entities of same type (e.g., two persons related to a film), and when those two entities can be made equal. We think that expressive features are hardly used because they rarely occur in search needs, and because most SPARKLIS' users have only tried a few queries so far. However, we think those are essential, and we dare an analogy with the vocabulary of a natural language: most words are rarely used (cf. Zipf law), but without them a language would be badly empoverished.

Navigation steps. Figure 7 shows for each number of step the percentage of users who have performed at least that number of steps (over one or several sessions). It therefore measures the capability of the tool to retain users in the exploration experience, and to encourage them to come back later. We have not used the number of sessions because a session can vary from the mere loading of SPARKLIS to a long sequence of complex queries. The number of steps is a more reliable measure of user activity. The proportion of steps done by bots seems negligible as the number of steps made by Google and MSN bots is only 39 (0.15%), and only consists in loading SPARKLIS or following

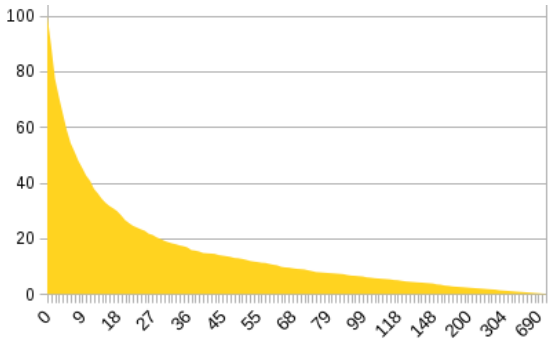


Fig. 7. Percentage of users having performed at least some number of steps

permalinks. More than 50% of users make at least 7 steps, but 12% of users make no step at all. One explanation for the latter is that half endpoints do not allow cross-domain requests from browsers, even though they are public, and other explanations are that some endpoints are not responsive enough or do not support essential SPARQL features or have no instances. The cross-domain issue has been solved with a proxy endpoint on mid-December 2014, and as a result the proportion of no-step users has decreased from 15.3% to 8.6%. At the other end of the spectrum, 10% of users have performed 60 steps or more, and one user has even reached 1400 steps. Those numerous steps are generally partitioned in several sessions over different days: 56 (6%) users have used the tool on 3 different days or more, and one of them has used it on 57 different days! Looking at the queries that they have built, it appears that those active users are clearly human, and pursuing sensible search goals. The above figures show that adoption by users is still relatively low, and that most users have only "played" with the tool. While a few users have adopted it for encyclopedic searches on DBpedia, many adopters use it on domain-specific datasets (e.g., bioinformatics, local datasets). We therefore think that adoption is strongly linked to the availability of endpoints interesting users.

6.3. Comparing user profiles

An interesting question is how different kinds of users compare in their usage of the tool. Unfortunately, the anonymous log provides no personal information on users such as sex, age, or technical background. It is also difficult to discriminate users according to their competency because they used the tool with different purposes: just trying the tool, reproducing the given examples, answering simple information needs on encyclopedic endpoints, or making professional use on domain-specific endpoints. We have therefore decided to compare two groups of users that differ on a clear criteria: the target endpoint.

	DBpedia	bio2rdf
nb. of users	781	71
avg. maximum query size	3.8	4.6
avg. nb. of steps	21.2	22.5

Table 4
Statistics about two groups of users: DBpedia and bio2rdf

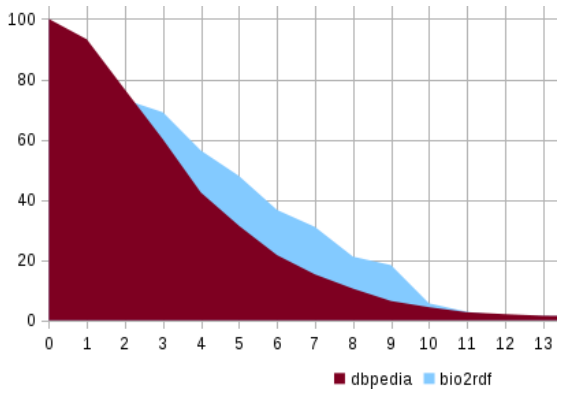


Fig. 9. Compared percentage of users building queries up to some size

We compare users on any DBpedia-related endpoint (hobby usage) with users on any bio2rdf-related endpoint (professional usage). Note that a same person may belong to either group at different times. Table 4 gives a few statistics on the two groups. It shows that users of the two groups made a similar number of steps on average, but that bio2rdf-users built more complex queries: on average, the most complex query each user built had 4.6 elements, compared to 3.8 for DBpedia-users. We also performed the same statistics as in previous section for each group, and compared their charts. For the area charts about query size and number of steps, the bio2rdf group is consistently above the DBpedia group. Most notably, Figure 9 shows that the proportion of users producing queries with size 7-9 is double (e.g., from 15% to 31%). Similarly, the proportion of users performing 15-20 steps is 1.5 times higher (e.g., from 26% to 40%); and the proportion of users having used the tool on several days jumps from 12% to 24%. Looking at query features, bio2rdf-users used slightly more properties (which can be explained by the higher query sizes), more filters and disjunctions, but less ordering and aggregation. Note that the frequency of aggregation in the DBpedia group is raised

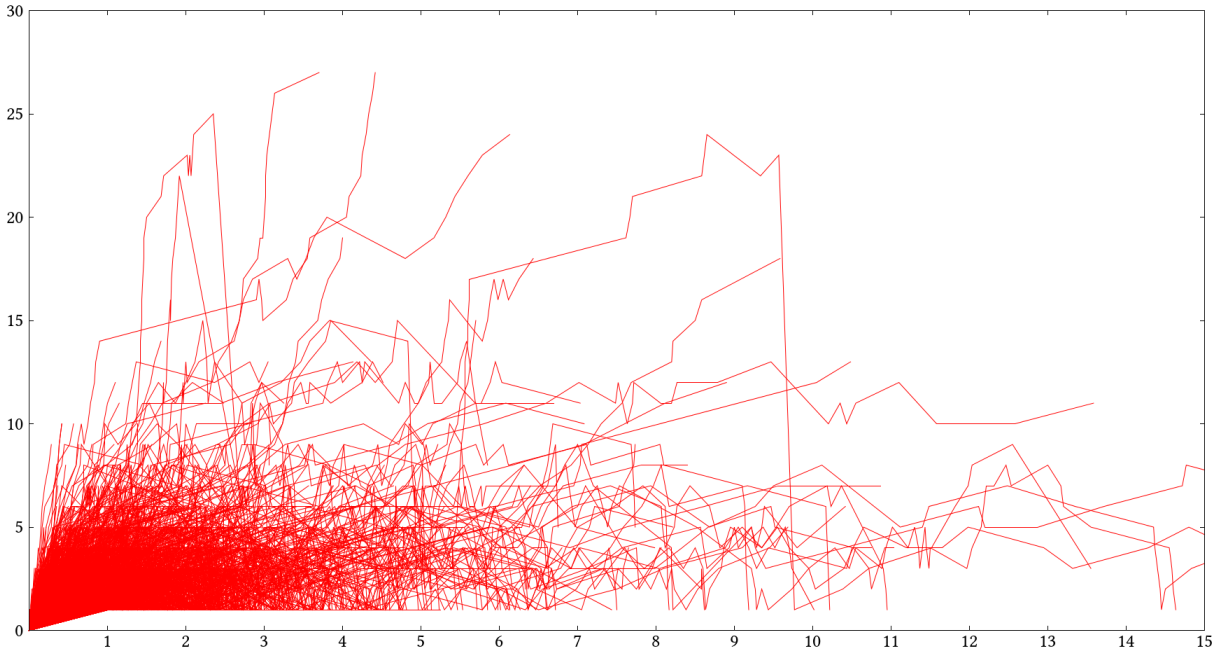


Fig. 8. Query size as a function of elapsed time (min) for all search episodes

Table 3
An example search about cars on Core English DBpedia

Step	Time	Query
1	00:00	**Give me something that matches car**
2	00:09	**Give me something**
3	00:42	**Give me an automobile**
4	01:11	**Give me an automobile that has a class**
5	01:53	**Give me an automobile whose class has a thumbnail**
6	02:20	**Give me an automobile that has a class**
7	02:32	**Give me an automobile whose class has a manufacturer**
8	02:56	**Give me an automobile that has a class**
9	02:58	**Give me an automobile**
10	03:12	**Give me an automobile that has a manufacturer**
11	03:20	**Give me an automobile whose manufacturer is Toyota**
12	03:50	**Give me an automobile whose manufacturer is Toyota and that has a transmission**
13	04:25	**Give me an automobile whose manufacturer is Toyota and whose transmission is Super ECT 4-Speed automatic**
14	04:35	**Give me an automobile whose manufacturer is Toyota and whose transmission is Super ECT 4-Speed automatic and that has a thumbnail**

by users reproducing example queries or testing the different modifiers. In total, the differences in the use of features are small, and not much can be concluded from them.

6.4. Search profiles and examples

In this section, we analyze more in depth user searches as sequences of queries over time. We define searches as normalized sessions. A search is a maximal subsequence of a session starting with the empty query, and having at least one step. This results in 2353 searches. Some users make long breaks (e.g., lunch break) in their search, so, to ease comparison of searches, we compress breaks to 1 minute. Figure 8 shows the query size as a function of elapsed time (in minutes) for all searches. Most searches cannot be read

individually, but the chart clarifies the correlation between user time, and query size, hence query complexity:

- queries with size 10 can be generated in less than a minute (short searches);
- queries with size 20 or more can be generated in 2-5 minutes (complex searches);
- a number of searches last up to 15 minutes, but generate queries with size less than 10 (long searches);
- most searches are less than 3 minutes, and generate queries with 5 elements at most.

The *short searches* can be interpreted as directed searches, where the user has a clear information need in mind. For example, one of the most frequent users built a query with size 9 in only 9 steps and 17s. The query is in bioinformatics, and searches for "studies where miRNAs show a decreased expression w.r.t. transcripts". The *long searches* can be interpreted as exploratory searches, where the user either gets familiarized with the dataset or the tool, or is answering a row of related questions. For example, a 3min search explores volcanos in DBpedia, finding their eruption years, ordering them to get the most recent and most ancient eruption years, then looking at places that have a volcano as highest place, and finally retrieving their elevation, and finding the highest ones. The *complex searches* are directed search, like short searches, but, because of their complexity, they require some exploration, and hence more steps and more time. They only concern domain-specific datasets, and correspond to actual information needs. We contend that the queries generated by those searches would have been very difficult or impossible to produce with other tools. For example, a user has built a query with size 19 in 36 steps and 4 minutes. It is applied to his/her own dataset about real lecturers, students, and courses. The search starts from a lecturer, retrieves the courses it teaches, and then some students of a Semantic Web course, and finally finds PhD students among them along with their thesis subject.

Searches less than 3 minutes and less than 5 elements can be either non-complex directed searches or short exploratory searches. They can also be unsuccessful searches with lots of trial-and-errors, and hesitations. Unfortunately, it is difficult to tell apart exploratory searches and unsuccessful searches because they have similar profiles. By examining in detail a number of searches, it appears that user mishandlings are not so frequent during interaction, and that the main obstacles are at start time. There are many void sessions because of unresponsive endpoints, and many very short sessions because users apparently test the tool with only a few random steps. The most common user mishandlings are:

- **redundant steps**, i.e. steps that does not make the query wrong but unnecessarily complex (e.g., **a thing that is a thing that ...**, **a thing or a thing**);
- **bad logic**, i.e. misinterpretation or wrong combinations of logical operators (e.g., **not optionally**, **optionally not**, **optionally** used to represent uncertainty);
- **ordering/aggregation confusion**, i.e. using the **minimum** aggregator instead of the **lowest-to-highest** ordering;
- **overuse of string filters**, i.e. inclination to use the tool like a search engine or a natural language interface.

Based on those observations, we have improved SPARKLIS' guidance with a tighter selection of suggested modifiers based on the query syntax and current focus, in order to eliminate most redundant steps and bad logic. For example, when the focus is on the phrase **a thing**, the disjunction modifier **or** is no more suggested; and when the focus is on (or in the context of) a negation or optional, the modifiers **not** and **optionally** are no more suggested either.

Table 3 details an example session about cars on Core English DBpedia. It shows an untrained user first learning by trial and error, and finally reaching a useful query of size 6 after 14 steps and less than 5 minutes. At step 1, he tries to filter individual entities with keyword "car", which does not work as expected. At step 3, he finds the class that interests him, **an automobile**. After exploring tentatively from property **class**, he seems to really get the trick at step 10, from which he reaches the final query in a straight line. He gets the cars he is looking for, along with a picture of them.

7. Conclusion and perspectives

SPARKLIS provides a stable and working answer to a frequent question in the Semantic Web: *"How to explore and query a large unknown endpoint beyond the most simple queries without reading or writing any SPARQL, and without preprocessing or configuration ?"*. Because of a novel interaction paradigm combining faceted search and query builders, user studies have shown that users need to learn how to use it, but

that after a short training, they can answer complex queries. Complex queries such as analytical queries involving multiple dimensions, aggregations, and filters, are only a few clicks away, under safe and complete guidance. Moreover, guidance is entirely mediated through natural language.

In the future, we will continue to maintain and improve SPARKLIS, especially w.r.t. expressivity and readability. For expressivity, we plan to cover almost all of SPARQL 1.1, i.e. to add expressions for computations, nested aggregations for rich analytics, and graphs as results for `CONSTRUCT` queries and updates. For readability, we plan to improve verbalization with lexicons, to extend multi-lingual support, and to better display results by generating full sentences enriched with multimedia objects (e.g., pictures), and graphical visualizations (e.g., charts, maps).

Acknowledgements. We thank the reviewers for their insightful suggestions that helped to improve this paper, in particular the evaluation section. We also thank the users who contributed to improve the tool itself with their feedback.

References

[1] M. Arenas, B.C. Grau, E. Kharlamov, Š. Marciuška, D. Zheleznyakov, and E. Jimenez-Ruiz. SemFacet: Semantic faceted search over YAGO. In *World Wide Web Conf. Companion*, pages 123–126. WWW Steering Committee, 2014.

[2] P. Cimiano, V. Lopez, C. Unger, E. Cabrio, A.-C. Ngonga Ngomo, and S. Walter. Multilingual question answering over linked data (QALD-3): Lab overview. In P. Forner, H. Müller, R. Paredes, P. Rosso, and B. Stein, editors, *Information Access Evaluation. Multilinguality, Multimodality, and Visualization - Int. Conf. CLEF Initiative*, LNCS 8138, pages 321–332. Springer, 2013.

[3] E.F. Codd, S.B. Codd, and C.T. Salley. *Providing OLAP (Online Analytical Processing) to User-Analysts: An IT Mandate*. Codd & Date, Inc, San Jose, 1993.

[4] S. Ferré. SQUALL: a controlled natural language for querying and updating RDF graphs. In T. Kuhn and N.E. Fuchs, editors, *Controlled Natural Languages*, LNCS 7427, pages 11–25. Springer, 2012.

[5] S. Ferré. Expressive and scalable query-based faceted search over SPARQL endpoints. In P. Mika and T. Tudorache, editors, *Int. Semantic Web Conf.* Springer, 2014.

[6] S. Ferré and A. Hermann. Reconciling faceted search and query languages for the Semantic Web. *Int. J. Metadata, Semantics and Ontologies*, 7(1):37–54, 2012.

[7] H. Haller. QuiKey – an efficient semantic command line. In *Knowledge Engineering and Management by the Masses (EKAW)*, pages 473–482. Springer, 2010.

[8] A. Harth. VisiNav: A system for visual search and navigation on web data. *J. Web Semantics*, 8(4):348–354, 2010.

[9] P. Heim, T. Ertl, and J. Ziegler. Facet graphs: Complex semantic querying made easy. In L. Aroyo et al., editor, *Extended Semantic Web Conference*, LNCS 6088, pages 288–302. Springer, 2010.

[10] A. Hermann, S. Ferré, and M. Ducassé. An interactive guidance process supporting consistent updates of RDFS graphs. In A. ten Teije et al., editor, *Int. Conf. Knowledge Engineering and Knowledge Management (EKAW)*, LNAI 7603, pages 185–199. Springer, 2012.

[11] M. Hildebrand, J. van Ossenbruggen, and L. Hardman. /facet: A browser for heterogeneous semantic web repositories. In I. Cruz *et al*, editor, *Int. Semantic Web Conf.*, LNCS 4273, pages 272–285. Springer, 2006.

[12] P. Hoefler, M. Granitzer, V. Sabol, and S. Lindstaedt. Linked data query wizard: A tabular interface for the semantic web. In *The Semantic Web: ESWC 2013 Satellite Events*, pages 173–177. Springer, 2013.

[13] C.A. Hurtado, A. Poulovassilis, and P.T. Wood. Query relaxation in RDF. In S. Spaccapietra, editor, *Journal on Data Semantics X*, LNCS 4900, pages 31–61. Springer, 2008.

[14] E. Hyvönen and E. Mäkelä. Semantic autocompletion. In *The Semantic Web (ASWC)*, pages 739–751. Springer, 2006.

[15] E. Kaufmann and A. Bernstein. Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. *J. Web Semantics*, 8(4):377–393, 2010.

[16] V. Lopez, V. S. Uren, M. Sabou, and E. Motta. Is question answering fit for the semantic web?: A survey. *Semantic Web*, 2(2):125–155, 2011.

[17] V. Lopez, M. Fernández, E. Motta, and N. Stieler. PowerAqua: Supporting users in querying and exploring the semantic web. *Semantic Web*, 3(3):249–265, 2012.

[18] E. Mäkelä, E. Hyvönen, and S. Saarela. Ontogator - a semantic view-based search engine service for web applications. In I. F. Cruz *et al.*, editor, *Int. Semantic Web Conf.*, LNCS 4273, pages 847–860. Springer, 2006.

[19] G. Marchionini. Exploratory search: from finding to understanding. *Communications of the ACM*, 49(4):41–46, 2006.

[20] J. McCrae, D. Spohr, and P. Cimiano. Linking lexical resources and ontologies on the semantic web with lemon. In *Extended Semantic Web Conference (ESWC)*, LNCS 6643, pages 245–259. Springer, 2011.

[21] C. Melo, A. Mikheev, B. Le Grand, and M.-A. Aufaure. Cubix: A visual analytics tool for conceptual and semantic data. In *Int. Conf. Data Mining Workshops*, pages 894–897. IEEE computer society, 2012.

[22] E. Oren, R. Delbru, and S. Decker. Extending faceted navigation to RDF data. In I. Cruz *et al*, editor, *Int. Semantic Web Conf.*, LNCS 4273, pages 559–572. Springer, 2006.

[23] G. M. Sacco and Y. Tzitzikas, editors. *Dynamic taxonomies and faceted search*. The information retrieval series. Springer, 2009.

[24] SPARQL11. SPARQL 1.1 query language, 2012. URL `http://www.w3.org/TR/sparql11-query/`. W3C Recommendation.

[25] M. Van Kleek, B. Moore, D.R. Karger, P. André, and M.C. Schraefel. Atomate it! end-user context-sensitive automation using heterogeneous information sources on the web. In *Int. Conf. World Wide Web*, pages 951–960. ACM, 2010.