Expressive and Scalable Query-Based Faceted Search over SPARQL Endpoints

Sébastien Ferré

Expressive and Scalable Query-based Faceted Search over SPARQL Endpoints

Sébastien Ferré

IRISA, Université de Rennes 1
Campus de Beaulieu, 35042 Rennes cedex, France
Email: ferre@irisa.fr

Abstract. Linked data is increasingly available through SPARQL endpoints, but exploration and question answering by regular Web users largely remain an open challenge. Users have to choose between the expressivity of formal languages such as SPARQL, and the usability of tools based on navigation and visualization. In a previous work, we have proposed Query-based Faceted Search (QFS) as a way to reconcile the expressivity of formal languages and the usability of faceted search. In this paper, we further reconcile QFS with scalability and portability by building QFS over SPARQL endpoints. We also improve expressivity and readability. Many SPARQL features are now covered: multidimensional queries, union, negation, optional, filters, aggregations, ordering. Queries are now verbalized in English, so that no knowledge of SPARQL is ever necessary. All of this is implemented in a portable Web application, Sparklis[1], and has been evaluated on many endpoints and questions.

1 Introduction

Linked data is increasingly available through SPARQL endpoints, but exploration and question answering is often tedious for SPARQL practitionners, and largely remains an open challenge for regular Web users. Semantic search can be evaluated according to a number of criteria such as expressivity or scalability. Maximizing any of those criteria in isolation is relatively easy, and what remains a challenge is to reconcile them as far as possible. We consider such a reconciliation as a key to the effective access to semantic data, and hence to the wide adoption of semantic data. Indeed, an easy-to-use system may not satisfy advanced users with more complex information needs, and everybody can become an advanced user in some domain (e.g., profession, hobby). Similarly, an expressive system that does not scale is only of limited use. We shortly define the criteria that motivated this work, and identify for each of them the existing approach that seems to best fulfill it.

Expressivity measures the diversity and complexity of questions that can be answered. It seems clear that the leading approach for maximizing that criteria is formal query languages, and prominently SPARQL.

[1] Sparklis http://www.irisa.fr/LIS/ferre/sparklis/osparklis.html

Guidance measures the level of assistance given to users in their search. It involves interactivity, suggestion, immediate feedback, and dead-ends prevention. We here retain Faceted Search (FS) [11,24]. First, FS is increasingly adopted in e-commerce and multimedia collections. Second, FS has already been adapted to semantic search (see Section 2).

Readability measures the ease for users to read and understand the textual components and controls of the user interface. Natural Language (NL) is obviously more readable than formal languages like SPARQL. If formal queries are used in a system, they should therefore be verbalized into NL, which has already been proposed for SPARQL [22].

Scalability measures the ability of a system to be responsive on large datasets. Most of the scalability effort in the semantic web has gone into RDF stores and SPARQL engines. It therefore seems reasonable to leverage their power.

Portability measures the cost of applying a system to a new dataset. Some systems require manual configuration, and others need an appropriate ontology. SPARQL endpoints have the advantage to provide a standard API.

Guidance and readability are two aspects of *usability* that we address in this paper. Other aspects that we do not consider here are *a smooth learning curve* or *personalization*. Another important criteria for semantic search that is not addressed in this paper is *openness*. It measures the ability to explore distributed linked data (querying several endpoints, following `owl:sameAs` links). In a previous work, we have introduced *Query-based Faceted Search* (QFS) [6,5] as a way to reconcile the expressivity of formal query languages with the guidance of faceted search. The obtained user interaction is similar to query builders that guide users in the construction of queries, but with a more fine-grained guidance that is based on actual data rather than on syntax and data schema.

The first contribution of this paper is to further reconcile QFS with the readability of NL. Questions and system suggestions are verbalized in (a fragment of) English so that no knowledge of SPARQL is ever necessary for users. This makes QFS a kind of Natural Language Interface (NLI) except that questions are not freely input by users, but produced through a user-system dialog. The second contribution is to further reconcile QFS with the scalability and portability of SPARQL endpoints. Question answers and system suggestions are entirely computed by generating and sending SPARQL queries to the endpoint. Portability is ensured by strictly conforming to the SPARQL standard. Compared to previous work, we also improve QFS expressivity. In addition to arbitrary basic graph patterns, unions, and negations, we now also cover multidimensional queries (tables as results), optionals, common filters, aggregations, and orderings. All those contributions are implemented in a Web application, Sparklis, that only needs the URL of the desired endpoint as input.

The main limitation to our work is that it only provides bare views of linked data, without any pre- or post-processing, and therefore exposes any data noise and heterogeneity to users. It uses neither linguistic knowledge (e.g., lexicons), nor external resources (e.g., full-text indexes) to make it more readable and efficient. We made this choice for the sake of portability, genericity, and simplicity, but nothing prevents to customize and improve our approach by using dataset-specific knowledge and resources.

Section 2 is an overview of different approaches and systems in semantic search (Section 2). Section 3 recalls the key principles of QFS to reconcile expressivity and guidance. Section 4 and 5 present the contributions of this paper: NL verbalization and QFS over SPARQL endpoints. Section 6 provides the results of a few evaluations of Sparklis w.r.t. the above criteria. Finally, Section 7 concludes and sketches future work.

2 Related Work

There are mainly two approaches to make semantic search more usable: user interaction (UI) and natural language (NL). UI systems reuse and adapt UI paradigms to semantic data: hypertext browsing (e.g., Fluidops Information Workbench[2]), query builders (e.g., SemanticCrystal [16]), faceted search (FS) [24], or OLAP [2]. Query builders generally offer more *expressivity*, but lack *readability* because 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 [19], 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 [14]. 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 [13], BrowseRDF [23], gFacet [12], VisiNav [9], 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) [18] use NL in various forms, going from full natural language (e.g., PowerAqua [17]) to mere keywords (e.g., NLP-Reduce [16]) through controlled natural languages (e.g., Ginseng [16], SQUALL [4]). Systems based on full NL or keywords devote the most effort to bridging the gap between lexical forms and ontology triples (mapping and

[2] Fluidops Information Workbench `http://iwb.fluidops.com/`

[3] mSpace `http://mspace.fm/`

[4] Longwell `http://simile.mit.edu/wiki/Longwell`

[5] OpenLink FS `http://dbpedia.org/fct/facet.vsp`

[6] Vinge Query&Explore `http://www.vingefree.com/querybyexplore/`

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 birthDate**
6	Give me a Writer that has nationality Russians and whose birthDate is **after 1800**
7	Give me a Writer that has nationality Russians and whose birthDate is after 1800 **and that is the author of something**
8	Give me a Writer that has nationality Russians and whose birthDate is after 1800 and that is the author of **a Book**
9	Give me a Writer that has nationality Russians and whose birthDate is after 1800 and that is the author of **a number of** Book
10	Give me a Writer that has nationality Russians and whose birthDate 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 birthDate 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 **Russian_Empire** and whose birthDate is after 1800 and that is the author of the highest-to-lowest number of Book

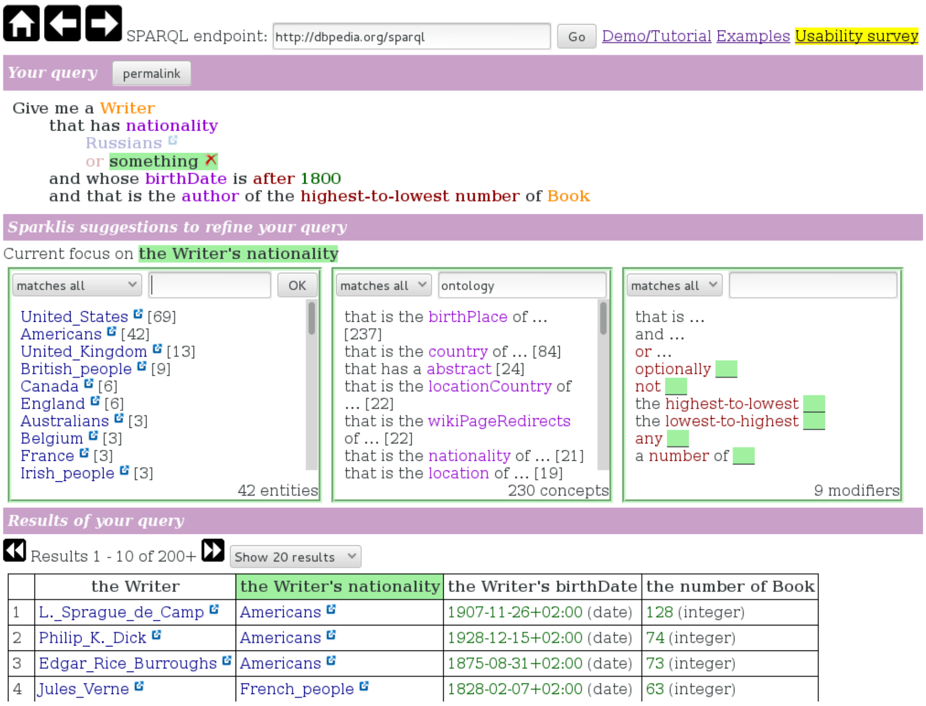
Table 1. Navigation scenario in Sparklis over DBpedia.

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.

Some systems integrate the UI approach in NLIs to alleviate the *habitability problem* [16], 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. Those systems (e.g., Ginseng [16], Atomate [25]) 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 hability problem.

3 Query-based Faceted Search

In this section, we recall the key principles of Query-based Faceted Search (QFS) [6,5], and how they enable to reconcile *expressivity* and *guidance*. QFS guidance relies on the interaction loop of Faceted Search (FS). FS guides users in the iterative refinement of a set of items. The key to QFS expressivity is to replace the *set of items* by a *structured query*, and to define the *focus* as a syntactic part of the query. System suggestions at each navigation step are therefore defined as *query transformations*, rather than as *set-based operations*.



SPARQL endpoint: http://dbpedia.org/sparql Go Demo/Tutorial Examples Usability survey

Your query permalink

Give me a Writer
that has nationality
Russians
or something ✗
and whose birthDate is after 1800
and that is the author of the highest-to-lowest number of Book

Sparklis suggestions to refine your query

Current focus on the Writer's nationality

matches all OK

United_States [69]
Americans [42]
United_Kingdom [13]
British_people [9]
Canada [6]
England [6]
Australians [3]
Belgium [3]
France [3]
Irish_people [3]
42 entities

matches all ontology

that is the birthPlace of ... [237]
that is the country of ... [84]
that has a abstract [24]
that is the locationCountry of ... [22]
that is the wikiPageRedirects of ... [22]
that is the nationality of ... [21]
that is the location of ... [19]
230 concepts

matches all

that is ...
and ...
or ...
optionally 🟩
not 🟩
the highest-to-lowest 🟩
the lowest-to-highest 🟩
any 🟩
a number of 🟩
9 modifiers

Results of your query

Results 1 - 10 of 200+ Show 20 results

	the Writer	the Writer's nationality	the Writer's birthDate	the number of Book
1	L._Sprague_de_Camp	Americans	1907-11-26+02:00 (date)	128 (integer)
2	Philip_K._Dick	Americans	1928-12-15+02:00 (date)	74 (integer)
3	Edgar_Rice_Burroughs	Americans	1875-08-31+02:00 (date)	73 (integer)
4	Jules_Verne	French_people	1828-02-07+02:00 (date)	63 (integer)

Fig. 1. Sparklis screenshot at step 11 of scenario in Table 1.

In short, we propose *query-based* FS as a generalization of classical *set-based* FS. It is a generalization because a set of items can be derived unambiguously from a query and focus (by query evaluation), while many queries may correspond to a given set of items. We have previously demonstrated that set-based FS has strong limits in terms of expressivity, which disappear with query-based FS [6]. In particular, it becomes possible to navigate to arbitrary Boolean combinations of elementary queries.

Table 1 shows the successive queries, as verbalized in Sparklis (see Section 4), of a QFS 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., `Russian`), classes (e.g., `a Book`), properties in both directions (e.g., `is the author of`, `has birthDate`), filters (e.g., `after 1800`), and various modifiers (e.g., `number of`, `or`). Figure 1 is a Sparklis screenshot at step 11, during the specification of an alternative nationality for the writer (`Russian_Empire` as a synonym 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). Note that QFS has relational results (tables) whereas classical FS has sets (lists). 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 query focus, such as Boolean connectors, aggregation operators, and ordering. Each list provides auto-completion for quickly locating a query element (see Section 5.2). 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.

4 NL Verbalization and SPARQL Translation

In this section, we improve previous work on QFS by verbalizing queries in NL for user display, and by translating them in SPARQL for evaluation by SPARQL endpoints. The current query and focus play a central role because they represent the full state of the navigation process. Results and suggestions are entirely computed from them. Their internal representation must be designed to facilitate three processes: (1) NL verbalization, (2) SPARQL translation, and (3) application of query transformations. Using SPARQL for internal representation would make (2) trivial, but (1) difficult as shown by previous work [22]. Using NL for internal representation would make (1) trivial, but (2) and (3) would be tedious because of the many peculiarities of NL. Our choice, inspired by state-of-the-art in the compilation of high-level programming languages, is to use Abstract Syntax Trees (AST) as an intermediate representation between NL and SPARQL. AST leaves are entities, values, and concepts. AST nodes correspond at the same time to NL syntactic structures, such as noun phrases (NP) or verb phrases (VP), and to SPARQL features, such as triple patterns or unions. The query focus, as a syntactic part of the query, is represented as a distinguished node of the AST of the query.

NL verbalization is performed by mapping AST leaves and nodes to NL expressions. Entities and concepts are verbalized by the local name of their URI, i.e. the part after the last sharp or slash. This choice was made for the sake of portability and efficiency, but future work will consider the use of RDFS labels and lexicons, e.g. represented in Lemon [20]. One or a few syntactic patterns are associated to each type of AST nodes. For example, syntactic patterns for relative clauses based on a property p are: `that has` p NP, `whose` p VP, and `that is`

query element	SPARQL feature
concept name, `relation`	triple pattern
entity name	URI in triple pattern
value	literal in triple pattern
`a`, `the`	variable in triple pattern, and after `SELECT`
`any`	variable not put after `SELECT`
`and`, `that`	join
`or`	`UNION`
`optionally`	`OPTIONAL`
`not`	`FILTER NOT EXISTS`
`matches`	`REGEX()`
`higher than`, `after`, `between`, ...	comparators ($<$, $\leq$, ...)
`language`	`lang()`
`datatype`	`datatype()`
`highest-to-lowest`	`ORDER BY DESC()`
`lowest-to-highest`	`ORDER BY ASC()`
`number of`	`COUNT()`
`list of`	`GROUP_CONCAT()`
`total`, `average`, ...	`SUM()`, `AVG()`, ...

Table 2. Mapping from Sparklis query elements to SPARQL features.

```
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:Russian_Empire . }
        ?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)
```

Fig. 2. SPARQL translation for the last query in the scenario of Table 1.

`the` *p* `of` *NP*. The choice of the pattern can depend on whether the object of the property is better verbalized as a *NP* or a *VP*. An example of that is visible in Table 1, when comparing steps 5 and 6: `that has a birthDate` becomes `whose birthDate is after 1800` after the insertion of the filter. To render the correct precedences of Boolean connectors, indentation is used in the display of the verbalized query (see Figure 1). That makes the query more readable, and avoids the use of brackets. Finally, syntax coloring is used to differentiate the different kinds of query elements: class names (orange), property names (purple), entity names (blue), values (green), modifiers (red).

SPARQL translation is based on Montague grammars [3], which were invented to bridge the gap between NL and formal languages. Those are founded on lambda calculus, and are fully compositional in the sense that the meaning of a sentence is the direct result of the composition of the meaning of its parts. Here, the "meaning" of a sentence or any of its parts is represented with SPARQL patterns. For example, a VP is translated to a function from entities to graph patterns, while a NP is translated to a function from graph patterns to graph patterns. Note that, unlike translating from SPARQL to NL, translating from ASTs to SPARQL is deterministic, and can be performed very efficiently. Figure 2 shows the SPARQL translation of the final query in the above navigation scenario, as produced by Sparklis. Table 2 maps elements of Sparklis queries to SPARQL features, and hence provides an overview of the expressivity of Sparklis compared to SPARQL. Remember that QFS interaction loop allows to build arbitrary combinations of those query elements, provided that those combinations make sense in the dataset, i.e. return results. For example, it is possible to use two aggregations in a same query, to perform ordering on an aggregated value, to follow arbitrary long property paths, or to define Boolean combinations of filters on a same variable. The main missing features compared to SPARQL 1.1 are: arbitrary expressions, only simple filters are available; subqueries, which are for example necessary to express nested aggregations; iterated property paths (operators `+` and `*`); named graphs (`GRAPH`) and federated search (`SERVICE`); `CONSTRUCT` and `DESCRIBE` queries; updates. Technical details about query/focus internal representation and SPARQL translation can be found in a research report [7].

5 Scalable QFS over SPARQL Endpoints

In this section, we improve previous work on QFS by entirely defining the computation of results and suggestions on top of SPARQL endpoints, rather than with in-memory RDF stores. We also take care to make it scale to the largest endpoints by limiting the number of results and suggestions. To preserve guidance completeness, results and suggestions beyond that limit can be found with an intelligent auto-completion mechanism.

NL verbalization, SPARQL translation, and the application of query transformations are computationally cheap, and are all done entirely on the client side. The computation of results from the SPARQL query uses one HTTP request to the SPARQL endpoint at each step, and therefore costs the same as when using a classical query editor in an incremental way. Therefore, the only significant additional cost of QFS, compared to direct querying in SPARQL, is the computation of suggestions, i.e. the three lists in the middle of Figure 1. Indeed, the three lists must be computed at each navigation step.

5.1 Computation of Suggestion Lists

The first suggestion list contains possible entities/values at the focus. For example, in the query `Give me a Writer that has a nationality` (step 3), nationalities of a writer are possible entities. Those entities/values are exactly those in the focus column of the result table, and can therefore be computed efficiently on client side. The third list contains applicable modifiers. There are only a dozen modifiers, and their applicability can be decided efficiently by looking at the query/focus. The second list contains concepts that possibly apply to entities/values in the first list. Here, a client-side computation is incompatible with portability, because it would require a client-side knowledge of the dataset, such as an ontology or indices. Moreover, ontology-based suggestions would be less precise because they would provide general rules (e.g., "books generally have authors"), rather than concrete facts (e.g., "only 10 of those 200 books have a defined author"). In heterogeneous datasets, like DBpedia, ontology-based guidance would often lead to empty results, and hence user frustration. Computing suggested concepts therefore require to query the SPARQL endpoint, and is actually the main issue for the scalability of QFS.

Assuming SPARQL variable `?f` is bound to a possible entity or value at focus, the possible classes can be obtained as the bindings of variable `?c` in the triple pattern `?f a ?c`. Similarly, possible properties can be obtained with the triple pattern `?f ?p []`, and inverse properties with the pattern `[] ?p ?f`. The question is how to use those triple patterns into SPARQL queries so as to efficiently get lists of suggestions. There are several ways to do it. For the binding of `?f`, the SPARQL translation of the query can be reused, or the entities/values of the focus column can be used into a large UNION pattern. Relative to the three kinds of suggestions (classes, properties, inverse properties), three queries can be used, or a single query using an UNION pattern. We made extensive experiments [7] to compare the efficiency of the different options, and came to the conclusion that the best option is generally to use three queries, one for each kind of suggestion, and to avoid the recomputation of the main query by using an UNION pattern over all focus entities/values: `f1`, `f2`, ...

```
Q1: SELECT ?c WHERE { {f1 a ?c} UNION {f2 a ?c} UNION ... }
Q2: SELECT ?p WHERE { {f1 ?p []} UNION {f2 ?p []} UNION ... }
Q3: SELECT ?p WHERE { {[] ?p f1} UNION {[] ?p f2} UNION ... }
```

In principle, each triple pattern is efficiently evaluated by RDF stores using classical indices, because it contains one resource (URI or literal), and one unbound variable.

It remains to define the computation of suggestions at the initial step, when the query is still empty, and therefore no focus entity/value is available. That computation is crucial to provide guidance from the beginning. In a first stage, we use the following efficient SPARQL queries to retrieve classes and properties:

```
Q1: SELECT DISTINCT ?c WHERE { ?c a rdfs:Class }
Q2: SELECT DISTINCT ?p WHERE { ?p a rdf:Property }
```

In SPARQL endpoints whose RDF graph does not contain the schema itself, the above queries return empty results. We then resort to the following queries which are less efficient, but have the advantage to reflect actual data.

```
Q1': SELECT DISTINCT ?c WHERE { [] a ?c }
Q2': SELECT DISTINCT ?p WHERE { [] ?p [] }
```

Although not mentioned for the sake of generality, all above SPARQL queries come in practice with a `LIMIT` clause to better control response times. We discuss in the next section the impact on the *completeness* of the guidance, and how it is addressed in Sparklis.

5.2 Intelligent Auto-Completion

There are two important properties for QFS guidance: 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 QFS. However, in practice, scalability requires to put limits on the number of query results and suggestions, and SPARQL endpoints also enforce such limits. Therefore, partial results and suggestions are unavoidable with large datasets. A previous work [7] has shown that partial results have generally a small impact on frequent concepts, but a high impact on infrequent concepts and entities/values. That impact is significant at the beginning of a search when result sets are large, and tends to disappear when the query becomes specific.

Our objective is to reconcile scalability in the computation of suggestions, and completeness in guidance. The solution that we have found and implemented in Sparklis is based on *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 [15,8]. 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 two ways. First, the filter condition depends on the user-selected filter operator. If that operator is `matches all`, then the suggestion must contain all keywords, in any order and insensitive to the case. If that operator is `between`, then the suggestion must be between the two given values, using numerical comparisons. Second, Sparklis auto-completion uses a cascade of three stages to ensure completeness. 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 (depending on the filter operator). 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. Given the increasing cost of stages 2 and 3, they are triggered only when the user has entered a full keyword (trailing space), so as not to do it at every keystroke.

6 Evaluations

We present three evaluations to assess the portability, the expressivity and scalability, and the usability of QFS, based on its implementation as a Web application: Sparklis. The experimental data of those evaluations are available at `http://www.irisa.fr/LIS/ferre/pub/iswc2014/`.

6.1 Sparklis: QFS as a Web Application

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. 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. For code safety and development speed, the JS code is compiled from a high-level language (OCaml using `js_of_ocaml`[7]). The source code counts about 4000 lines of code, and the minimized JS code weights about 260k.

6.2 Portability to SPARQL Endpoints

We first assess the portability of Sparklis on permanent SPARQL endpoints found on the Web. The web site *SPARQL endpoints status*[8] maintains a list of SPARQL endpoints, along with dynamic information about their availability, performance, and expressivity. We took a sample of 57 active endpoints, and tried Sparklis on each of them. We report on three aspects: success/failure of the connection, performance, and usability. Out of the 57 endpoints, 3 connections failed because of the endpoint server (e.g., HTTP 500 error, POST requests not accepted), 24 were successful, and surprisingly 30 connections failed because of the *same-origin policy*. That policy, enforced by Web browsers, forbids scripts to send HTTP requests to other origins (e.g., `http://dbpedia.org/`) than the script origin (`http://www.irisa.fr/` for Sparklis). It is crucial for the security of many Web applications, but it is not relevant for SPARQL endpoints, which act as public web services. Fortunately, there is a simple solution, but it is the responsability of each endpoint administrator to apply it. It suffices to add the line `Access-Control-Allow-Origin: *` in the HTTP response headers.

Out of the 24 successful connections, 22 were responsive enough to allow for fluid exploration of the dataset. The initial step typically took between 1 and 3 seconds, and 7 seconds in the worst case. In terms of contents, 10 endpoints contained only facts about concepts, ontologies, and datasets; 10 endpoints contained concrete facts about various topics (e.g., Austrian skiers, Chile administration, Nobel prizes); and 4 endpoints appeared empty. The 4 latter endpoints appeared empty because of a bug in some RKBEXPLORER-based

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

[8] `http://sparqles.okfn.org/`

endpoints related to `UNION`: an union of empty results is not empty, and contains unbounded bindings. On a few endpoints, our random explorations have led to interesting results. For example, on `http://data.ox.ac.uk/sparql/`, we got the list of Oxford colleges along with their logo, and a picture. On `http://data.nobelprize.org/sparql`, we found the laureates of the Peace Nobel prize in decreasing year, people with several prizes (e.g., Marie Curie had 2), and the shocking gender imbalance (44 women vs 803 men). For 6 endpoints, URI local names of entities are opaque codes (e.g., numbers) that hinder the readability of suggestions and results. It does not prevent to access and use the real names of entities, but it requires additional navigation steps.

6.3 Expressivity and Scalability over DBpedia QALD Questions

We here assess the practical expressivity and scalability of Sparklis using questions from the QALD-3 challenge[9]. QALD (Question Answering over Linked Data) primarily targets Natural Language Interfaces (NLI) where natural questions are answered in one interaction step. We instead use QALD questions as the expression of information needs that are satisfied through multi-step interaction (QFS). For each of the 100 training questions over DBpedia, we evaluated the minimum interaction time to answer the question, when possible, starting from the empty query (`Give me something`). Because we here evaluate the scalability and efficiency of Sparklis, and not the usability (see next section), we strived to minimize the exploration and thinking user time by using the gold standard SPARQL queries provided by QALD as a guide in the selection of suggestions. Therefore, the measured times represent the optimal interaction time for a trained and focused user. In real use, interaction times will increase according to unfamiliarity with Sparklis and the dataset, and to lack of focus in search (exploratory search).

Expressivity. With Sparklis, we could answer 90/100 questions. Out of the 10 failed questions, 9 correspond to missing information in DBpedia (aka. OUT OF SCOPE questions). The other failed question is because a YAGO class could not be found due to timeouts from the endpoint. Unlike DBpedia classes that are quickly found among suggestions, YAGO classes are numerous and difficult to get. A few gold standard queries could not be constructed, but the answer was still easy to get. For example, the question asking *whether Natalie Portman was born in the US* cannot be reached because she was born in Israel, but as Sparklis immediately shows the actual birth place, the answer was obviously *No*.

Scalability. Figure 3 shows the distribution of the wall-clock interaction time for the 90 successful questions[10]. Half of the questions can be answered in less than 27s (median time). The most complex QALD questions can be answered in less than 2min. We found those results quite satisfactory given the billions of triples of DBpedia. The simplest questions generally involve an entity and

[9] `http://www.sc.cit-ec.uni-bielefeld.de/qald/`

[10] Note that the responsiveness of DBpedia endpoint may vary over time.

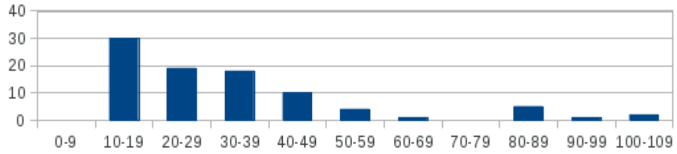


Fig. 3. Histogram of wall-clock interaction times for 90/100 QALD-3 questions.

a property (e.g., *Give me the homepage of Forbes*), or a class, a property, and a value (e.g., *Give me all books written by Danielle Steel*). The more complex questions combine unions, string matching, numerical comparisons, counts, or ordering. For example, question (*Which telecommunications organizations are located in Belgium?*) has 3 unions and 2 string matching because there are different ways to express *telecommunications* and *located in Belgium* due to data heterogeneity, and it requires 14 navigation steps.

For comparison, we shortly describe the performance of the best NLI participant to the QALD-3 challenge: CASIA [10]. CASIA produced correct answers for 29 questions, and partially correct answers for 8 questions. The average computation time over the 100 questions is 83s. With regard to faceted search, we found no system capable of exploring DBpedia and responsive enough.

6.4 First Usability Experiments

We have so far conducted three usability experiments of QFS. In the first experiment [6], we asked 20 graduate students to answer 18 questions about genealogical data using Sewelis, a desktop version of QFS. The dataset was small, but the questions already involved complex graph patterns, disjunction, and negation, and the subjects had no previous knowledge about SW technologies. The results showed that, after a short training, all subjects were able to answer simple questions, and most of them were able to answer complex questions. The average time per question ranged from half a minutes to six minutes. The main observed difficulty was in understanding the notion of focus. The SUS questionnaire showed that subjects did not find the system *unnecessarily complex*, and that they *would learn to use it very quickly*.

In the second experiment [7], the developer of the former version of Sparklis, Scalewelis, took part in the QALD3 challenge on DBpedia questions. He was of course expert in the use of Scalewelis, but he was unfamiliar with the DBpedia dataset. He tried to answer the 100 test questions, and submitted them. Because Scalewelis was less expressive than Sparklis, he could answer "only" 70 questions. Out of them, 32 were correct and 1 was partially correct. Most errors were because there are often several representations of the same meaning in DBpedia, which produce different answers: e.g., "being an actor" is represented either by the pattern `?x a dbo:Actor` or by `[] dbo:starring ?x`. Also, most properties come in two versions: one in DBpedia ontology, and one among DBpedia properties. Scalewelis was ranked third out of six participants, with recall

(33%) and precision (33%) very similar to best NLI approaches. This experiment demonstrated that QFS is a promising approach for question answering, albeit it is based on interaction rather than on NL understanding.

In the third experiment, which is still ongoing as an online survey we ask anonymous Web users to try and answer 12 questions over DBpedia, after viewing a tutorial video[11]. Only 6 people have filled the survey yet, but they have different profiles, and results are already instructive. On average, they have built correct queries for 7.8 questions. For non-IT people (2/6), the average goes only slightly down to 7. For people having some knowledge of SPARQL (2/6), it goes up to 10 questions. An expert user, who can write SPARQL queries, had 9 correct answers, made 2 errors related to disjunction, and skipped a single query (which was skipped by all users). (S)he found Sparklis much easier to use than SPARQL, and declared: "I like use this system and I find it is easy to use unlike other semantic web search engines". An advanced user, who can do some programming but never heard about SPARQL, had 8 correct answers, made 2 errors by using string matching instead of numerical comparison, and skipped 2 queries. (S)he found it difficult to find the right concepts (classes and properties), but declared that "there are no inconsistencies in the suggestions", and that "the system is really usable". A regular user, who know neither SPARQL nor programming, had still 6 correct answers (involving comparisons, negation, and ordering), 3 incorrect answers, and 4 skipped queries. (S)he expressed difficulties with focus position and interaction logic, but declared that "for some questions, it would take me hours to do the same in Excel, while here a few clicks are enough!". No user managed to answer the question *Which U.S. states do not possess gold minerals?* because it requires to find one among the many YAGO classes (see Section 6.3). Most expert and advanced users (4/5) managed to answer the complex question *Give me all bridges crossing the Saint Lawrence river, ordered by decreasing length, and with an optional depiction.* whose answer is an ordered three-dimensional table (bridge, length, and depiction). The SUS questionnaire gives results consistent with our first experiment, and answers mostly differ on the expected amount of learning depending on user background.

7 Conclusion

We have improved Query-based Faceted Search (QFS) with NL verbalization for readability, and with SPARQL endpoint-based computation of results and suggestions for scalability and portability. This makes it an appealing approach for semantic search as it reconciles the expressivity of formal languages, the guidance of query builders and faceted search, the readability of natural language interfaces, the scalability of the most powerful RDF stores and SPARQL engines, and the portability to many SPARQL endpoints thanks to a strong conformance to W3C standards. Our evaluations have shown that our QFS implementation, Sparklis, can be used effectively on various endpoints, without configuration, can

[11] Survey form and video are online at `http://tinyurl.com/kxozx9r`

answer most QALD questions and more, and was evaluated positively in first usability experiments.

Our priorities for future work concern expressivity, readability, and visualization. We aim to cover all SPARQL features while avoiding to make user interaction more complex. Note that union, negation, aggregation, and ordering all simply add modifiers as suggestions, and we expect the same for other SPARQL features. User interaction in QFS is *FS + query + focus* so that usability is mostly a matter of readability and visualization. We plan to improve readability by using ontology lexicons [20] in NL verbalization, when available, and visualization by producing graphical views (e.g., diagrams, charts, maps) from tables of results [21,14].

Acknowledgement We are grateful to Joris Guyonvarc'h for his master work that contributed to this work by designing, implementing, and experimenting a first version of QFS over SPARQL endpoints. Interesting technical details not present in this paper for space reasons can be found in a technical report [7]. We also thank the reviewers for their thorough analysis, and encouraging comments.

References

1. Arenas, M., Grau, B., Kharlamov, E., Š. Marciuška, Zheleznyakov, D., Jimenez-Ruiz, E.: SemFacet: Semantic faceted search over YAGO. In: World Wide Web Conf. Companion. pp. 123–126. WWW Steering Committee (2014)
2. Codd, E., Codd, S., Salley, C.: Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate. Codd & Date, Inc, San Jose (1993)
3. Dowty, D.R., Wall, R.E., Peters, S.: Introduction to Montague Semantics. D. Reidel Publishing Company (1981)
4. Ferré, S.: SQUALL: a controlled natural language for querying and updating RDF graphs. In: Kuhn, T., Fuchs, N. (eds.) Controlled Natural Languages. pp. 11–25. LNCS 7427, Springer (2012)
5. Ferré, S., Hermann, A.: Semantic search: Reconciling expressive querying and exploratory search. In: Aroyo, L., Welty, C. (eds.) Int. Semantic Web Conf. pp. 177–192. LNCS 7031, Springer (2011)
6. Ferré, S., Hermann, A.: Reconciling faceted search and query languages for the Semantic Web. Int. J. Metadata, Semantics and Ontologies 7(1), 37–54 (2012)
7. Guyonvarch, J., Ferre, S., Ducassé, M.: Scalable Query-based Faceted Search on top of SPARQL Endpoints for Guided and Expressive Semantic Search. Research report PI-2009, IRISA (2013), `http://hal.inria.fr/hal-00868460`
8. Haller, H.: QuiKey – an efficient semantic command line. In: Knowledge Engineering and Management by the Masses (EKAW). pp. 473–482. Springer (2010)
9. Harth, A.: VisiNav: A system for visual search and navigation on web data. J. Web Semantics 8(4), 348–354 (2010)
10. He, S., Liu, S., Chen, Y., Zhou, G., Liu, K., Zhao, J.: CASIA@QALD-3: A question answering system over linked data. In: et al., C.U. (ed.) Work. Multilingual Question Answering over Linked Data (QALD-3) (2013), `http://www.clef2013.org`
11. Hearst, M., Elliott, A., English, J., Sinha, R., Swearingen, K., Yee, K.P.: Finding the flow in web site search. Communications of the ACM 45(9), 42–49 (2002)

12. Heim, P., Ertl, T., Ziegler, J.: Facet graphs: Complex semantic querying made easy. In: et al., L.A. (ed.) Extended Semantic Web Conference. pp. 288–302. LNCS 6088, Springer (2010)
13. Hildebrand, M., van Ossenbruggen, J., Hardman, L.: /facet: A browser for heterogeneous semantic web repositories. In: *et al*, I.C. (ed.) Int. Semantic Web Conf. pp. 272–285. LNCS 4273, Springer (2006)
14. Hoefler, P., Granitzer, M., Sabol, V., Lindstaedt, S.: Linked data query wizard: A tabular interface for the semantic web. In: The Semantic Web: ESWC 2013 Satellite Events, pp. 173–177. Springer (2013)
15. Hyvönen, E., Mäkelä, E.: Semantic autocompletion. In: The Semantic Web (ASWC), pp. 739–751. Springer (2006)
16. Kaufmann, E., Bernstein, A.: Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. J. Web Semantics 8(4), 377–393 (2010)
17. Lopez, V., Fernández, M., Motta, E., Stieler, N.: PowerAqua: Supporting users in querying and exploring the semantic web. Semantic Web 3(3), 249–265 (2012)
18. Lopez, V., Uren, V.S., Sabou, M., Motta, E.: Is question answering fit for the semantic web?: A survey. Semantic Web 2(2), 125–155 (2011)
19. Mäkelä, E., Hyvönen, E., Saarela, S.: Ontogator - a semantic view-based search engine service for web applications. In: *et al.*, I.F.C. (ed.) Int. Semantic Web Conf. pp. 847–860. LNCS 4273, Springer (2006)
20. McCrae, J., Spohr, D., Cimiano, P.: Linking lexical resources and ontologies on the semantic web with lemon. In: Extended Semantic Web Conference (ESWC). pp. 245–259. LNCS 6643, Springer (2011)
21. Melo, C., Mikheev, A., Le Grand, B., Aufaure, M.A.: Cubix: A visual analytics tool for conceptual and semantic data. In: Int. Conf. Data Mining Workshops. pp. 894–897. IEEE computer society (2012)
22. Ngomo, A.C.N., Bühmann, L., Unger, C., Lehmann, J., Gerber, D.: Sorry, I don't speak SPARQL: translating SPARQL queries into natural language. In: WWW. pp. 977–988 (2013)
23. Oren, E., Delbru, R., Decker, S.: Extending faceted navigation to RDF data. In: *et al*, I.C. (ed.) Int. Semantic Web Conf. pp. 559–572. LNCS 4273, Springer (2006)
24. Sacco, G.M., Tzitzikas, Y. (eds.): Dynamic taxonomies and faceted search. The information retrieval series, Springer (2009)
25. Van Kleek, M., Moore, B., Karger, D., André, P., Schraefel, M.: Atomate it! end-user context-sensitive automation using heterogeneous information sources on the web. In: Int. Conf. World Wide Web. pp. 951–960. ACM (2010)