Reconciling faceted search and query languages for the Semantic Web

Sébastien Ferré, Alice Hermann

Reconciling Faceted Search and Query Languages for the Semantic Web

S. Ferré

IRISA,
University Rennes 1,
35042 Rennes, France
Fax: +33 2 99 84 71 71
E-mail: ferre@irisa.fr
*Corresponding author

A. Hermann

IRISA,
INSA Rennes,
35043 Rennes, France
Fax: +33 2 99 84 71 71
E-mail: Alice.Hermann@irisa.fr

Abstract Faceted search and querying are two well-known paradigms to search the Semantic Web. Querying languages, such as SPARQL, offer expressive means for searching RDF datasets, but they are difficult to use. Query assistants help users to write well-formed queries, but they do not prevent empty results. Faceted search supports exploratory search, i.e., guided navigation that returns rich feedbacks to users, and prevents them to fall in dead-ends (empty results). However, faceted search systems do not offer the same expressiveness as query languages. We introduce *Query-based Faceted Search* (QFS), the combination of an expressive query language and faceted search, to reconcile the two paradigms. We formalize the navigation of faceted search as a navigation graph, where navigation places are queries, and navigation links are query transformations. We prove that this navigation graph is *safe* (no dead-end), and *complete* (every query that is not a dead-end can be reached by navigation). In this paper, the LISQL query language generalizes existing semantic faceted search systems, and covers most features of SPARQL. A prototype, Sewelis, has been implemented, and a usability evaluation demonstrated that QFS retains the ease-of-use of faceted search, and enables users to build complex queries with little training.

Keywords: semantic web; faceted search; query language; exploratory search; navigation; expressiveness.

Reference to this paper should be made as follows: Ferré, S. and Hermann, A. (2012) 'Combining Faceted Search and Query Languages for the Semantic Web', *Int. J. Metadata, Semantics, and Ontologies*, Vol. ?, Nos. ?/?, pp.???–???.

Biographical notes: S. Ferré is an assistant professor at the University of Rennes 1, France, and a member of the LIS research team in the IRISA laboratory. He holds a PhD in Computer Science from the University of Rennes 1, and has also been an assistant researcher at the University of Wales, Aberystwyth. He works on Logical Information Systems (LIS), and has published papers about formal concept analysis, logics for knowledge representation and reasoning, information retrieval and exploration, and data-mining. His application domains are personal information management, geographical information systems, and software engineering.

A. Hermann is a PhD student at INSA Rennes, and a member of the LIS research team in the IRISA laboratory. Her PhD subject is to adapt and extend Logical Information Systems (LIS) to the Semantic Web, both for data exploration and data creation.

1 Introduction

A key issue of the Semantic Web is to provide an easy and effective access to them, not only to specialists, but also to casual users. The challenge is not only to allow users to retrieve particular resources (e.g., flights), but to support them in the exploration of a knowledge base (e.g., which are the destinations? Which are the most frequent flights? With which companies and at which price?). We call the first mode *retrieval search*, and the second mode *exploratory search*, following Marchionini (2006). Exploratory search is often associated to *faceted search* (Hearst et al. 2002, Sacco & Tzitzikas 2009), but it is also at the core of Logical Information Systems (LIS) (Ferré & Ridoux 2000, Ferré 2009), and Dynamic Taxonomies (Sacco 2000). Exploratory search allows users to find information without *a priori* knowledge about either the data or its schema. Faceted search works by suggesting restrictions, i.e., selectors for subsets of the current selection of items. Restrictions are organized into facets, and only those that share items with the current selection are suggested. This has the advantage to provide guided navigation, and to prevent dead-ends, i.e., empty selections. Therefore, faceted search is *easy-to-use* and *safe*: *easy-to-use* because users only have to choose among the suggested restrictions, and *safe* because, whatever the choice made by users, the resulting selection is not empty. The selections that can be reached by navigation correspond to queries that are generally limited to conjunctions of restrictions, possibly with negation and disjunction on values. This is far from the expressiveness of query languages for the semantic web, such as SPARQL[1]. There are *semantic faceted search* systems that extend the expressiveness of reachable queries, but still to a small fragment of SPARQL (e.g., SlashFacet (Hildebrand et al. 2006), BrowseRDF (Oren et al. 2006), SOR (Lu et al. 2007), gFacet (Heim et al. 2010), VisiNav (Harth 2010)). For instance, none of them allows for cycles in graph patterns, unions of complex graph patterns, or negations of complex graph patterns.

Languages for querying the semantic web, such as SPARQL (Angles & Gutierrez 2008), OWL-QL (Fikes et al. 2004), or SPARQL-DL (Sirin & Parsia 2007), are quite expressive but are difficult to use, even for specialists. Users are asked to fill an empty field (problem of the writer's block), and nothing prevents them to write a query that has no answer (dead-end). Even if users have a perfect knowledge of the syntax and semantics of the query language, they may be ignorant about the data schema, i.e., the *ontology*. If they also master the ontology or if they use a graphical query editor (e.g., SemanticCrystal (Kaufmann & Bernstein 2010), the SCRIBO Graphical Editor[2]) or an auto-completion system (e.g., Ginseng (Kaufmann & Bernstein 2010)) or keyword query translation (e.g., Hermes (Tran et al. 2009)), the query will be syntactically correct and semantically consistent w.r.t. the ontology but it can still produce no answer.

The contribution of this paper, *Query-based Faceted Search* (QFS), is to define a semantic search that is (1) easy to use, (2) safe, and (3) expressive. Ease-of-use and safeness are retained from existing faceted search systems by keeping their general principles, as well as the visual aspect of their interface. Expressiveness is obtained by representing the current selection by a *query* rather than by a set of items, and by representing navigation links by *query transformations* rather than by set operations (e.g., intersection, crossing). In this way, the expressiveness of faceted search is determined by the expressiveness of the query language, rather than by the combinatorics of user interface controls. In this paper, the query language, named LISQL, generalizes existing semantic faceted search systems, and covers most features of SPARQL. The use of queries for representing selections in faceted search has other benefits than navigation expressiveness. The current query is an intensional description of the current selection that complements its extensional description (list of items). It informs users in a precise and concise way about their exact position in the navigation space. It can easily be copied and pasted, stored and retrieved later. Finally, it allows expert users to modify the query by hand at any stage of the navigation process, without loosing the ability to proceed by navigation.

The paper is organized as follows. Section 2 gives preliminaries about the Semantic Web and Faceted Search. Section 3 discusses the limits of set-based faceted search by formalizing the navigation from selection to selection. Section 4 introduces LISQL queries and their transformations. In Section 5, navigation with QFS is formalized and proved to be *safe* and *complete* w.r.t. LISQL, as well as *efficient*. Section 6 provides details about our prototype implementation Sewelis. Section 7 reports about a user study that demonstrates the usability of our approach. Our approach is also compared in Section 8 to other works in faceted search and query languages for the Semantic Web. Finally, Section 9 concludes.

2 Preliminaries

2.1 Semantic Web

The Semantic Web (SW) is founded on several representation languages, such as RDF, RDFS, and OWL, which provide increasing inference capabilities (Hitzler et al. 2009). The two basic units of these languages are *resources* and *triples*. A resource can be either a URI (Uniform Resource Identifier), a literal (e.g., a string, a number, a date), or a *blank node*, i.e., an anonymous resource. A URI is the absolute name of a *resource*, i.e., an entity, and plays the same role as a URL w.r.t. web pages. Like URLs, a URI can be a long and cumbersome string (e.g., `http://www.w3.org/1999/02/22rdf-syntaxns#type`), so that it is often denoted by a qualified name (e.g., `rdf:type`). We assume pairwise disjoint infinite sets of URIs (U), blank nodes (B), and

literals (L). The set of *resources* is then defined as $R = U \cup B \cup L$.

A triple (s, p, o) is made of 3 resources, and can be read as a simple sentence, where s is the subject, p is the verb (called the predicate), and o is the object. For instance, the triple `(ex:Bob,rdf:type,ex:man)` says that "Bob has type man", or simply "Bob is a man". Here, the resource `ex:man` is used as a class, and `rdf:type` is used as a property, i.e., a binary relation. The triple `(ex:Bob,ex:friend,ex:Alice)` says that "Bob has friend Alice", where `ex:friend` is another property. The triple `(ex:man,rdfs:subClassOf,ex:person)` says that "man is subsumed by person", or simply "every man is a person". The set of all triples of a knowledge base forms a RDF graph.

Definition 2.1: An *RDF graph* is defined as a set of *triples* $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$, where s is the *subject*, p is the *predicate*, and o is the *object*.

A *vocabulary* is a set of resources having a meaning defined by convention. RDF(S) is a vocabulary used to represent the membership to a class (`rdf:type`), subsumption between classes (`rdfs:subClassOf`), subsumption between properties (`rdfs:subPropertyOf`), the domain (`rdfs:domain`) and range (`rdfs:range`) of properties, the meta-class of classes (`rdfs:Class`), the meta-class of properties (`rdf:Property`), etc. OWL introduces additional vocabulary to represent complex classes and properties: e.g., restrictions on properties, intersection of classes, inverse properties. The variant OWL-DL is the counterpart of Description Logics (DL) (Baader et al. 2003), where resources are individuals, classes are concepts, and properties are roles. A RDF graph that uses the OWL vocabulary to define classes and properties is generally called an *ontology*. Each vocabulary comes with a semantics, and the richer the vocabulary is, the more expressive and the more complex inference is.

Vocabulary for genealogy. For illustration purposes, we consider RDF graphs about genealogical data. To this purpose, we introduce a custom vocabulary for genealogy. The URIs of this domain are associated to a namespace (`gen:`). This prefix is omitted if there is no ambiguity. Resources can be *persons*, *events*, *places* or literals such as names or dates. Persons belong either to the class of *men* or to the class of *women*, may have a *firstname*, a *lastname*, a *sex*, a *father*, a *mother*, a *spouse*, a *birth*, and a *death*. A birth or a death is an event that may have a *date* and a *place*. Places can be described as *parts* of larger places. The classes of *men* and *women* are declared as subclasses of the class of *persons*. Properties *father* and *mother* are declared as subproperties of property *parent*. The transitive closure of property *parent* is obtained by defining property *ancestor* as transitive, and a super-property of *parent*. For illustration purposes, we use genealogical datasets converted from GED files[3]. In particular, we use a small dataset about the ascendancy of George Washington, which we also used in our user evaluation, reported in Section 7. This dataset has about 400 resources, including 79 persons, and about 4000 triples.

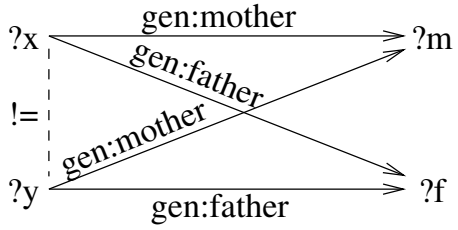

Figure 1 A graphical representation of a graph pattern.

Query languages provide on semantic web knowledge bases the same service as SQL on relational databases. They generally assume that implicit triples have been inferred and added to the base. The most well-known query language, SPARQL, reuses the `SELECT FROM WHERE` syntax of SQL queries, using graph patterns in the WHERE clause. For instance, pairs of siblings can be retrieved by the following query:

```
SELECT DISTINCT ?x ?y FROM <mygen.rdf>
WHERE { ?x gen:mother ?m. ?x gen:father ?f.
        ?y gen:mother ?m. ?y gen:father ?f.
        FILTER ?x != ?y }
```

Figure 1 shows a graphical representation of the graph pattern of this query, where arrows represent triples oriented from the subject to the object. The query reads "two persons `?x` and `?y` are siblings if they share a same mother and a same father, and are different". The FILTER condition is necessary because nothing prevents two variables to bind to a same resource. SPARQL provides a number of relational algebra operators to combine basic graph patterns: join, set union (`UNION`), left join (`OPTIONAL`), named graphs (`GRAPH`), and filtering by various constraints (`FILTER`). SPARQL 1.1 extends its expressiveness with set difference (`MINUS`), negation (`NOT EXISTS`), subqueries, aggregations, and expressions.

2.2 Faceted Search

Faceted Search (Hearst et al. 2002, Sacco & Tzitzikas 2009) covers a family of user interfaces for browsing a collection of items. It is becoming a *de facto* standard in e-commerce websites, and its scope of application is wide (see Chap. 9 in (Sacco & Tzitzikas 2009)). It is suitable for *retrieval search*, i.e., the quick retrieval of an item already known to the user. It is also suitable for *exploratory search* (Marchionini 2006), i.e., the discovery of the objects that best suits the needs of the user, who has no prior knowledge of the item collection. An example of the later is when users want to buy a new camera. They do not know which models exist and what their features are, but they have constraints and preferences such as

low cost, high resolution, or brand. Faceted search systems guide users through the item collection, and give them the feeling to have considered all the possibilities. At each navigation step, users only have to make a choice among a set of alternatives that are suggested by the system.

The data model underlying faceted search is simple. Each item is described along a set of *facets*, or dimensions. Each facet has a range of values. Therefore, each item is described by a set of pairs facet-value, which we call *features*. Conversely, each feature f has a set of items, noted $items(f)$. A facet is not necessarily defined on all items. At any navigation step, the current *selection* is defined as a set of items S. The initial selection S_0 is generally the whole item collection. From the current selection S, a set of *restrictions* are computed and displayed to the user. A restriction is a feature f that matches at least one item of the current selection, i.e., the intersection between S and the set of items $items(f)$ is not empty ($S \cap items(f) \neq \emptyset$). Each restriction is generally accompanied by the number of items it matches ($\|S \cap items(f)\|$). Restrictions are organized by facets, and for the sake of conciseness, most facets are initially collapsed, and expanded on demand. On the one hand, restrictions provide a summary of the current selection. On the other hand, each restriction is a selector for a subset of the selection. The summary plays a crucial role in exploratory search because for each facet it shows only and all of the relevant values for the current selection. This allows for the informed choice of a restriction: e.g., the lowest price or the highest resolution that is available given previous selected restrictions.

Exploration in faceted search is based on set operations between selections and restrictions. At each navigation step, a new selection is derived by applying a set operation between the current selection S and a restriction f chosen by the user. Typically, the set operation is intersection, i.e. $S := S \cap items(f)$. Extensions of faceted search may allow for the exclusion of a restriction ($S := S \setminus items(f)$), or the union with a restriction ($S := S \cup items(f)$). After a navigation step, a new set of restrictions is displayed to reflect the new selection. The list of chosen restrictions is generally displayed, and any of them can be removed by users, leading to a larger selection. This is useful to relax a constraint, for example in order to get more items and restrictions. The list of chosen restrictions can be seen as a *query*, which, in general, is limited to a conjunction of features, while restricted forms of negation and disjunction are sometimes available.

Dynamic Taxonomies (DT) (Sacco 2000, 2006, Sacco & Tzitzikas 2009) are a model of faceted search, where a multidimensional taxonomy is used instead of facets and values. In fact, facets and values form a two-level taxonomy, with facets at the first level, and values at the second level. Using taxonomies of arbitrary depth allows for features at different granularity levels. For instance, a facet of date can be used at the levels of days, weeks, months, years, etc. Weeks and months can be combined because taxonomies need not be trees but can be directed acyclic graphs. Features are called *concepts*, and the generalization ordering between features is called *subsumption*. Taxonomies are multidimensional, in that several features, even under a same facet, can be attached to a same item. This is useful with a facet of topics as a same item can match several topics. The term "dynamic taxonomy" stands for the fact that the summary is now a subset of the taxonomy, which dynamically adapts to the selection.

Logical Information Systems (LIS) (Ferré & Ridoux 2000, 2004, Ferré 2009) are another model of faceted search that has been developed in our team since 1999, on the basis of Formal Concept Analysis (Ganter & Wille 1999) and logic-based information retrieval (van Rijsbergen 1986). For what concerns us here, logical information systems can be defined as an extension of dynamic taxonomies, where features are the formulas of an ad-hoc logic, and subsumption is defined by logical inference rather than explicitly (Ferré & Ridoux 2007). Using logics enhances the expressiveness of features and queries, as well as the design and engineering of complex taxonomies (see Chapter 8 in (Sacco & Tzitzikas 2009)). In LIS, the selection is defined as the set of answers, called *extension*, of the query, and changes of the selection are done through changes to the query. In addition to navigation, LIS provide direct querying for expert users, and query-by-examples to find items similar to a given set of examples.

3 Limits of Set-based Faceted Search for the Semantic Web

The notions of faceted search can be transposed to the Semantic Web. Items and values are resources (URIs or literals), and facets are properties. The association between an item and a facet-value is a triple, where the subject is the item, the predicate is the facet, and the object is the value. Because of the relational nature of semantic data, new kinds of features and set operations have been introduced in *semantic faceted search* (e.g., /facet (Hildebrand et al. 2006), BrowseRDF (Oren et al. 2006), SOR (Lu et al. 2007), gFacet (Heim et al. 2010), VisiNav (Harth 2010)). In addition to facet-value pairs, a feature can be the name of a resource, a class as a type, the domain of a property, or the range of a property (e.g., BrowseRDF). Table 1 defines the syntax and semantics (set of items) of the various kinds of features, where r denotes any RDF resource (URI or literal), c denotes a RDFS class, p denotes a RDF property, and S_0 denotes the set of all items (possibly all resources of a RDF dataset). In semantic expressions, we use the following definitions of set-based operations involving a property p and a RDF graph G:

$$p(., S) := \{i \in S_0 \mid \exists j \in S : (i, p, j) \in G\}$$
$$p(S, .) := \{j \in S_0 \mid \exists i \in S : (i, p, j) \in G\}$$

feature	syntax f	semantics $items(f)$	examples
name	r	$\{r\}$	`<JohnSmith>`, `"John"`, `2011`
type	`a` c	rdf:type(., $\{c\}$)	`a person`
facet-value	p `:` r	$p(., \{r\})$	`year : 2011`
inverse facet-value	p `of` r	$p(\{r\}, .)$	`mother of <JohnSmith>`
domain	p `: ?`	$p(., S_0)$	`year : ?`
range	p `of ?`	$p(S_0, .)$	`mother of ?`

Table 1 Syntax, semantics, and examples of the various kinds of features.

Those operations can be used in addition to intersection, union, and exclusion (e.g., /facet, SOR, gFacet, VisiNav). The operation $p(S, .)$ is crossing forward p from the selection S, while the operation $p(., S)$ is crossing backwards. For example, starting from a set S of persons, `gen:lastname`$(S, .)$ returns their lastnames, while starting from a set of lastnames, `gen:lastname`$(., S)$ returns the set of persons having one of those lastnames. Table 2 defines the various kinds of operations that can be used to navigate from one selection to another. Crossings apply to domain and range restrictions, while other operations apply to arbitrary restrictions. In order to better expose the limits of set-based faceted search, we introduce in the table a syntax for those operations, where S denotes the current selection.

operation	syntax	semantics
reset	`?`	S_0
intersection	S `and` f	$S \cap items(f)$
exclusion	S `and not` f	$S \setminus items(f)$
union	S `or` f	$S \cup items(f)$
crossing backwards	p `:` S	$p(., S)$
crossing forwards	p `of` S	$p(S, .)$

Table 2 Syntax and semantics of the various kinds of set-based operations.

The syntactic form of selections in Table 2 implicitly defines the language of queries (q) that can be reached by set-based faceted search:

$$q \rightarrow \texttt{?} \mid q \texttt{ and } f \mid q \texttt{ and not } f \mid q \texttt{ or } f \mid p \texttt{ : } q \mid p \texttt{ of } q$$

This grammar already defines a rich language of accessible queries, but it has strong limits in terms of flexibility and expressiveness. This can be seen at first sight by the fact that the right-hand side of intersection, difference, and union are restricted to features, instead of arbitrary queries. This linearly recursive definition of queries comes from the linear navigation of set-based faceted search. The consequence of this linearity is that not all combinations of intersection, union, and crossings are reachable, which is counter-intuitive and limiting for end users. For example, the following kinds of selections are not reachable, where the R_i represent the set of items of some features:

- unions of complex selections, e.g., $(R_1 \cap R_2) \cup (R_3 \cap R_4)$;
- or intersections of crossings from complex selections, e.g., $p_1(., R_1 \cap R_2) \cap p_2(., R_3 \cap R_4)$.

Note that a selection $S_1 \cap p(., S_2)$ cannot in general be obtained by first navigating to S_1, then crossing forwards p, navigating to S_2, and finally crossing backwards p, because it is not equivalent to $p(., p(S_1, .) \cap S_2)$ unless p is inverse functional.

Existing approaches to semantic faceted search often have additional limitations, which are sometimes hidden behind a lack of formalization. In some systems (e.g., BrowseRDF, gFacet), a same facet (a property) cannot be used several times, which is fine for functional properties but not for relations such as "child": p `: (`f_1 `and` f_2`)` is reachable but not `(`p `:` f_1`) and (`p `:` f_2`)`. In other systems (e.g., /facet), a property whose domain and range are the same cannot be used as a facet, which includes all family and friend relationships for instance.

4 Expressive Queries and their Transformations

The contribution of our approach, *Query-based Faceted Search* (QFS), is to significantly improve the expressiveness of faceted search, while retaining its properties of safeness (no dead-end), and ease-of-use. The key idea is to define navigation links at the syntactic level as query transformations, rather than at the semantic level as set operations. Indeed, the syntactic expression of a query retains more information than its semantics (a set of items) because a query has a single set of items, but a set of items can be the semantics of many different queries. The navigation from selection to selection, as well as the computation of restrictions related to the current selection, are retained because a set of items of the current query can be computed at any time.

In Section 4.1, we first define the syntax and semantics of LISQL (LIS Query Language). LISQL generalizes in a natural way the query language behind set-based faceted search (see Section 3), by allowing for the free combination of features, intersection, difference, union, and crossings. We then define in Section 4.2 a set of query transformations so that every LISQL query can be reached in a finite sequence of such transformations. This is in contrast with previous contributions in faceted search that introduce new selection transformations, and

leave the query language implicit. We think that making the language of reachable queries explicit is important for reasoning on and comparing different faceted search systems. In Section 4.3, we give a translation from LISQL to SPARQL, the reference query language of the Semantic Web. This provides both a way to compute the answers of queries with existing tools, and a way to evaluate the level of expressiveness achieved by LISQL.

4.1 The LIS Query Language (LISQL)

query	syntax q	semantics $items(q)$
name	r	$\{r\}$
type	`a` c	rdf:type(., $\{c\}$)
all	`?`	S_0
crossing backwards	p `:` q_1	$p(., S_1)$
crossing forwards	p `of` q_1	$p(S_1, .)$
complement	`not` q_1	$S_0 \setminus S_1$
intersection	q_1 `and` q_2	$S_1 \cap S_2$
union	q_1 `or` q_2	$S_1 \cup S_2$

Table 3 Syntax and semantics of LISQL queries.

LISQL is obtained by merging the syntactic categories of features and queries in the grammar of Section 3, so that every query can be used in place of a feature.

Definition 4.1: The syntax and semantics of the LISQL constructs is defined in Table 3, where r is a resource, c is a class, p is a property, S_0 is the set of all items, and q_1, q_2 are LISQL queries s.t. $S_1 = items(q_1)$ and $S_2 = items(q_2)$.

The definition of LISQL allows for the arbitrary combination of intersection, union, complement, and crossings. In order to further improve the expressiveness of LISQL from tree patterns to graph patterns, we add variables (e.g., `?X`) as an additional construct. Variables serve as co-references between distant parts of the query, and allows for the expression of cycles. For example, the query that selects people who are an employee of their own father can be expressed as `a person and father : ?X and employee of ?X`, or alternately as `a person and ?X and employee of father of ?X`. The semantics of queries with variables is given with the translation to SPARQL in Section 4.3, because it cannot be defined inductively, like in Table 3.

Syntactic constructs are given in increasing priority order (see Table 3), and brackets or indentation are used in concrete syntax for disambiguation. The most general query `?` is a neutral element for intersection, and an absorbing element for union. In the following, we use the example query q_{ex} = `a person and birth : (year : (1601 or 1649) and place : (?X and part of England)) and father : birth : place : not ?X`, which uses all constructs of LISQL, and selects the set of "persons born in 1601 or 1649 at some place in England, and whose father is born at another place". The same LISQL query with indentation instead of brackets displays as follows (the `and` connectors are omitted).

```
a person
birth :
   year :
      1601 or 1649
   place :
      ?X
      part of England
father : birth : place : not ?X
```

This notation better renders the structure of the query, and is therefore used in our prototype Sewelis (see Section 6).

4.2 Query Transformations

We have generalized the query language by allowing complex queries in place of features: e.g., q_1 `and` q_2 instead of q `and` f. However, because the number of suggested restrictions in faceted search must be finite, it is not possible to suggest arbitrarily complex queries as restrictions. More precisely, the *vocabulary* of features must be finite. In QFS, we retain the same set of features as in Section 3 (i.e., names, types, pairs facet-value, domain, and range), which is a finite subset of LISQL for any given dataset.

The key notion we introduce to reconcile this finite vocabulary and the reachability of arbitrary LISQL queries is the notion of *focus* in a query. This notion allows our approach to escape the linearity of set-based navigation, and therefore to reach queries with arbitrary syntax trees.

Definition 4.2: A *focus* of a LISQL query q is a node of the syntax tree of q, or equivalently, a subquery of q. The set of foci of q is noted $\Phi(q)$; the *root focus* corresponds to the root of the syntax tree, and represents the whole query. The subquery at focus $\phi \in \Phi(q)$ is noted $q[\phi]$.

In the following, when it is necessary to refer to a focus in a query, the corresponding subquery is underlined with the focus name as a subscript, like in `mother of` $\underline{?}_\phi$. Foci are used in QFS to specify on which subquery a query transformation should be applied. For example, the query `(`f_1 `and` f_2`) or (`f_3 `and` f_4`)` can be reached from the query `(`f_1 `and` f_2`) or` f_3 by applying intersection with restriction f_4 to the subquery f_3, instead of to the whole query. Similarly, the query p_1 `:` `(`f_1 `and` f_2`) and` p_2 `:` `(`f_3 `and` f_4`)` can be reached by applying the intersection with restriction f_4 to the subquery f_3. This removes the problem of unreachable selections in set-based faceted search presented in Section 3.

Definition 4.3: A *query transformation* transforms a query q into the query $q[\phi := q_1]$ by replacing the subquery at focus $\phi \in \Phi(q)$ by another query q_1. One note $q[t_1]\dots[t_n]$ if several transformations are successively applied. The also define the following abbreviations for common query transformations:

$$\begin{array}{ll} [\phi \texttt{ and } q_1] & = [\phi := q[\phi] \texttt{ and } q_1] \\ [\phi \texttt{ and not } q_1] & = [\phi := q[\phi] \texttt{ and not } q_1] \\ [\phi \texttt{ or } q_1] & = [\phi := q[\phi] \texttt{ or } q_1] \end{array}$$

We show in the following equations how the intersection $q[\phi \texttt{ and } \delta]$ with any LISQL query δ that is not a feature can be recursively decomposed into a finite sequence of intersections with features, and exclusions or unions with the most general query `?`.

δ	$q[\phi \texttt{ and } \delta]$
`?`	q
p `:` q_1	$q[\phi \texttt{ and } p \texttt{ : } \underline{?}_{\phi_1}][\phi_1 \texttt{ and } q_1]$
p `of` q_1	$q[\phi \texttt{ and } p \texttt{ of } \underline{?}_{\phi_1}][\phi_1 \texttt{ and } q_1]$
`not` q_1	$q[\phi \texttt{ and not } \underline{?}_{\phi_1}][\phi_1 \texttt{ and } q_1]$
q_1 `and` q_2	$q[\phi \texttt{ and } \underline{q_1}_{\phi_1}][\phi_1 \texttt{ and } q_2]$
q_1 `or` q_2	$q[\phi \texttt{ and } \underline{q_1}_{\phi_1}][\phi_1 \texttt{ or } \underline{?}_{\phi_2}][\phi_2 \texttt{ and } q_2]$

For example, the complex query q_{ex} = `a person and birth : (year : (1601 or 1649) and place : (?X and part of England)) and father : birth : place : not ?X` can be reached through the navigation path:

$\underline{?}_{\phi_0}$
$[\phi_0$ `and a person`$]$
$[\phi_0$ `and birth :` $\underline{?}_{\phi_1}]$
$[\phi_1$ `and year :` $\underline{\texttt{1601}}_{\phi_2}]$
$[\phi_2$ `or` $\underline{?}_{\phi_3}]$
$[\phi_3$ `and 1649`$]$
$[\phi_1$ `and place :` $\underline{?}_{\phi_4}]$
$[\phi_4$ `and ?X`$]$
$[\phi_4$ `and part of England`$]$
$[\phi_0$ `and father :` $\underline{?}_{\phi_5}]$
$[\phi_5$ `and birth :` $\underline{?}_{\phi_6}]$
$[\phi_6$ `and place :` $\underline{?}_{\phi_7}]$
$[\phi_7$ `and not` $\underline{?}_{\phi_8}]$
$[\phi_8$ `and ?X`$]$.

The classical facet-value features (p `:` r and p `of` r) appear to be redundant for navigation as their intersection can be decomposed, but they are still useful for visualization in a faceted search interface.

Sequences of query transformations are analogous to the use of graphical query editors, but the key difference is that a valid query, answers, and restrictions will be returned at each navigation step, providing feedback, understanding-at-a-glance, no dead-end, and all benefits of exploratory search. Despite the syntax-based definition of navigation steps, they have a clear semantic counterpart. Intersection is the same as in standard faceted search, only making it available on the different entities involved in the current query. In the above example, intersection is alternately applied to the person, his birth, his birth's place, his father, etc. The set of relevant restrictions is obviously different at different foci. The union transformation introduces an alternative to some subquery (e.g., an alternative birth's year). The exclusion transformation introduces a set of exceptions to the subquery (e.g., excluding some father's birth's place). In Section 5, we precisely define which query transformations are suggested at each navigation step, and we prove that the resulting navigation graph is safe (no dead-end), and complete (every "safe" query is reachable).

4.3 Translation to and Comparison with SPARQL

We propose a (naive) translation of LISQL queries to SPARQL queries. It involves the introduction of variables that are implicit in LISQL queries. This translation provides an alternative way to compute LISQL query answers, in addition to Table 3. As this translation applies to LISQL queries with co-reference variables, it becomes possible to compute their set of items.

Definition 4.4: The *SPARQL translation* of a LISQL query q is $sparql(q)$ = `SELECT DISTINCT ?`x `WHERE {` $S_0(x)$ $GP(x,q)$ `}`, where the graph pattern $S_0(x)$ binds x to an element of the set of all items S_0, and the function GP inductively defines the graph pattern of q with variable x representing the root focus.

$$\begin{array}{ll} GP(x, \texttt{?}v) & = S_0(v) \texttt{ FILTER (?}x \texttt{ = ?}v\texttt{)} \\ GP(x, r) & = \texttt{FILTER (?}x \texttt{ = } r\texttt{)} \\ GP(x, \texttt{a } c) & = \texttt{?}x \texttt{ a } c\texttt{.} \\ GP(x, p \texttt{ : } q_1) & = \texttt{?}x\ p\ \texttt{?}y\texttt{.}\ GP(y, q_1) \\ & \quad \text{where } y \text{ is a fresh variable} \\ GP(x, p \texttt{ of } q_1) & = \texttt{?}y\ p\ \texttt{?}x\texttt{.}\ GP(y, q_1) \\ & \quad \text{where } y \text{ is a fresh variable} \\ GP(x, \texttt{?}) & = \texttt{\{ \}} \\ GP(x, \texttt{not } q_1) & = \texttt{NOT EXISTS \{ } GP(x, q_1) \texttt{ \}} \\ GP(x, q_1 \texttt{ and } q_2) & = GP(x, q_1)\ GP(x, q_2) \\ GP(x, q_1 \texttt{ or } q_2) & = \texttt{\{ } GP(x, q_1) \texttt{ \} UNION \{ } GP(x, q_2) \texttt{ \}} \end{array}$$

The graph pattern $S_0(x)$ may depend on the application. By default, it is defined as (`?`x `a rdfs:Resource.`), and allows to select all kinds of resources, including classes, properties, and literals. The use of $S_0(x)$ in graph patterns ensures that variables are bound in filters and negations.

We now discuss the translations of LISQL queries compared to SPARQL in general. They have only one variable in the SELECT clause because of the nature of faceted search, i.e., navigation from set to set. From SPARQL 1.0, LISQL misses the optional graph pattern, and the named graph pattern. Optional graph patterns are mostly useful when there are several variables in the SELECT clause. LISQL has the `NOT EXISTS` construct of SPARQL 1.1. If we look at the graph patterns generated for intersection and union, the two subpatterns necessarily share at least one variable, x. This is a restriction compared to SPARQL, but one that makes little

difference in practice as disconnected graph patterns are hardly useful in practice.

The translation $sparql(q_{ex})$ of the above example query $q_{ex} =$ `a person and birth : (year : (1601 or 1649) and place : (?X and part of England)) and father : birth : place : not ?X` is the following.

```
SELECT DISTINCT ?x
WHERE {
   ?x1 a gen:person.
   ?x1 gen:birth ?x2.
      ?x2 gen:year ?x3.
         { FILTER (?x3 = 1601)}
         UNION { FILTER (?x3 = 1649) }
      ?x2 gen:place ?x4.
         ?X a rdfs:Resource.
            FILTER (?x4 = ?X)
         gen:England gen:part ?x4.
   ?x1 gen:father ?x5.
      ?x5 gen:birth ?x6.
         ?x6 gen:place ?x7.
            NOT EXISTS {
               ?X a rdfs:Resource.
                  FILTER (?x7 = ?X)
            }
```

This query can be manually improved as follows:

```
SELECT DISTINCT ?f
WHERE {
   ?p a gen:person.
   ?p gen:birth ?b.
      ?b gen:year ?y.
         FILTER (?y=1601 || ?y=1649).
      ?b gen:place ?X.
         gen:England gen:part ?X.
   ?p gen:father ?f.
      ?f gen:birth ?bf.
         ?bf gen:place ?pf.
            FILTER (?pf != ?X) }
```

This example shows that LISQL is more concise, and makes a minimal use of variables. It also replaces a number of logical and algebraic symbols (curly brackets, dot, UNION, FILTER, `=`, `!=`, `&&`, `||`, and `!`) by keywords for the 3 Boolean operators (`and`, `or`, `not`) plus brackets or indentation. The LISQL syntax follows the usual syntax for expressions (infix operators and brackets/indentation to fix priorities), and we think that this makes it easier to read and learn.

5 A Safe and Complete Navigation Graph

In this section, we formally define the navigation space over a RDF dataset as a graph, where vertices are navigation places, and edges are navigation links. A navigation place is made of a query q and a focus ϕ of this query. The focus determines the selection of items to be displayed, and the set of restrictions for that selection. A navigation link is defined by a query transformation and, possibly, a focus change. We prove the safeness of navigation graphs in Section 5.1, and their completeness in Section 5.2. Finally, we discuss the efficiency of our approach relative to set-based faceted-search in Section 5.3. Before defining the navigation graph itself, we first define the set of items and the set of restrictions for some query q and some focus $\phi \in \Phi(q)$. The set of items is defined as the set of items of the query $\mathit{flip}(q, \phi)$, which is the reformulation of q from the point of view of the focus ϕ. For example, the reformulation, called the *flip*, of the query `a woman and mother of` $\underline{\texttt{name : "John"}}_{\phi}$ is the query `name : "John" and mother : a woman`.

Definition 5.1: The *flip* of a query q at a focus $\phi \in \Phi(q)$ is defined as $\mathit{flip}(q, \phi) = \mathit{flip}'(q, \phi, \texttt{?})$. The function flip' inductively deconstructs the query q until reaching the focus ϕ, and uses the additional (third) argument k as an accumulator for the resulting flipped query. In the following equations, the subquery that contains focus ϕ is underline.

$$
\begin{aligned}
\mathit{flip}'(p \texttt{ : } \underline{q_1}, \phi, k) &= \mathit{flip}'(q_1, \phi, p \texttt{ of } k)\\
\mathit{flip}'(p \texttt{ of } \underline{q_1}, \phi, k) &= \mathit{flip}'(q_1, \phi, p \texttt{ : } k)\\
\mathit{flip}'(\underline{q_1} \texttt{ and } q_2, \phi, k) &= \mathit{flip}'(q_1, \phi, k \texttt{ and } q_2)\\
\mathit{flip}'(q_1 \texttt{ and } \underline{q_2}, \phi, k) &= \mathit{flip}'(q_2, \phi, k \texttt{ and } q_1)\\
\mathit{flip}'(\underline{q_1} \texttt{ or } q_2, \phi, k) &= \mathit{flip}'(q_1, \phi, k)\\
\mathit{flip}'(q_1 \texttt{ or } \underline{q_2}, \phi, k) &= \mathit{flip}'(q_2, \phi, k)\\
\mathit{flip}'(\texttt{not } \underline{q_1}, \phi, k) &= \mathit{flip}'(q_1, \phi, k)\\
\mathit{flip}'(\underline{q_\phi}, \phi, k) &= q \texttt{ and } k
\end{aligned}
$$

When the focus is in the scope of an union, only the alternative that contains the focus is used in the flipped query. This is necessary to have the correct set of restrictions at that focus, and this is also useful to access the different subselections that compose an union. For example, in the query `a man and (`$\underline{\texttt{firstname : "John"}}_{\phi}$ `or lastname : "John")`, the focus ϕ allows to know the set of men whose firstname is John without removing the second alternative in the current query. When the focus is in the scope of a complement, this complement is ignored in the flipped query. This is useful to access the subselection to be excluded. For example, in the query `a man and not` $\underline{\texttt{father : ?}}_{\phi}$, the focus ϕ allows to know the set of men who have a father, i.e., those who are to be excluded from the selection of men.

Definition 5.2: The *items* of a query q at focus ϕ is defined as the items of the *flip* of q at focus ϕ, i.e., $\mathit{items}(q, \phi) = \mathit{items}(\mathit{flip}(q, \phi))$.

This enables the definition of the set of restrictions at each focus in the normal way. The navigation graph can then be formally defined.

Definition 5.3: The *restrictions* of a query q at focus ϕ is defined as the features that share items with the query q at focus ϕ:

$$restr(q, \phi) = \{f \mid items(q, \phi) \cap items(f) \neq \emptyset\}.$$

Definition 5.4: Let D be a RDF dataset. The *navigation graph* $G_D = (P, L)$ of D has its set of navigation places (vertices) defined by

$$P = \{(q, \phi) \mid q \in LISQL, \phi \in \Phi(q)\},$$

and its set of navigation links (edges) defined by Table 4 for every place $p = (q, \phi)$. The notation $p' = p[l]$ denotes the navigation place obtained by traversing the navigation link l from the navigation place p. One can note $p[l_1] \dots [l_n]$ when several links are traversed.

The number of navigation places is infinite because there are infinitely many LISQL queries, but the number of outgoing navigation links is finite at each navigation place because the vocabulary of features is finite, and the number of foci and variables in a query is finite. By default, the initial navigation place is $p_0 = (\underline{?}_\phi, \phi)$.

Before stating and proving safeness and completeness of the navigation graph, we state a few useful lemmas. The first lemma relates the flip of transformed queries to the transformation of flipped queries.

Lemma 5.1: *For every query q, focus $\phi \in \Phi(q)$, and query q', we verify:*

1. $flip(q[\phi \texttt{ and } \underline{q'}_{\phi'}], \phi') = flip(q, \phi) \texttt{ and } q'$,
2. $flip(q[\phi \texttt{ and not } \underline{q'}_{\phi'}], \phi') = flip(q, \phi) \texttt{ and } q'$, *(see remarks after Definition 5.1)*
3. $flip(q[\phi \texttt{ or } \underline{q'}_{\phi'}], \phi') = flip(q[\phi := \texttt{?}], \phi) \texttt{ and } q'$.

The second lemma states that intersection navigation links behave as in standard faceted search: intersection with a feature f leads to a navigation place, whose set of items is the intersection between the previous selection and the set of items matching that feature f.

Lemma 5.2: *For every query q, focus $\phi \in \Phi(q)$, and feature f, the following equality holds:*

$$items((q, \phi)[\texttt{and } f]) = items(q, \phi) \cap items(f).$$

Proof: $items((q, \phi)[\texttt{and } f])$
$= items(q[\phi \texttt{ and } \underline{f}_{\phi'}], \phi')$ (Definition 5.4)
$= items(flip(q[\phi \texttt{ and } \underline{f}_{\phi'}], \phi'))$ (Definition 5.2)
$= items(flip(q, \phi) \texttt{ and } f)$ (Lemma 5.1)
$= items(flip(q, \phi)) \cap items(f)$ (Definition 4.1)
$= items(q, \phi) \cap items(f)$ (Definition 5.2) □

The third lemma states that deletion navigation links can only make the set of items larger, and therefore cannot lead to dead-ends.

Lemma 5.3: *For every query q, focus $\phi \in \Phi(q)$, the following equality holds:*

$$items((q, \phi)[\texttt{delete}]) \supseteq items(q, \phi).$$

5.1 Safeness

Safeness is an important property to be retained from faceted search. A navigation graph is safe if no navigation path leads to a dead-end, unless it starts with a dead-end. Safeness prevents frustation in user experience. We prove that navigation graphs are safe, apart from the use of focus change (see discussion below).

Theorem 1: *Let D be a RDF dataset. The navigation graph G_D is* safe *except for some focus changes, i.e., for every path of navigation links without focus change from (q, ϕ) to (q', ϕ'), $items(q, \phi) \neq \emptyset$ implies $items(q', \phi') \neq \emptyset$.*

Proof: It suffices to prove that every navigation link l that is not a focus change is safe, i.e., that $items((q, \phi)[l]) \neq \emptyset$ assuming that $items(q, \phi) \neq \emptyset$. The proof is based on Definitions 5.2, 5.3, 5.4 and Lemmas 5.1, 5.2.

intersection: $items((q, \phi)[\texttt{and } f])$
$= items(q, \phi) \cap items(f)$ (Lemma 5.2)
$\neq \emptyset$ because $f \in restr(q, \phi)$ (Definition 5.3).

exclusion: $items((q, \phi)[\texttt{and not ?}])$
$= items(q[\phi \texttt{ and not } \underline{?}_{\phi'}], \phi')$ (Definition 5.4)
$= items(flip(q[\phi \texttt{ and not } \underline{?}_{\phi'}], \phi'))$ (Definition 5.2)
$= items(flip(q, \phi) \texttt{ and ?}) = items(flip(q, \phi))$
(Lemma 5.1)
$= items(q, \phi) \neq \emptyset$.

union: $items((q, \phi)[\texttt{or ?}])$
$= items(q[\phi \texttt{ or } \underline{?}_{\phi'}], \phi')$ (Definition 5.4)
$= items(q[\phi := q[\phi] \texttt{ or } \underline{?}_{\phi'}], \phi')$ (Definition 4.3)
$= items(q[\phi := \underline{?}_\phi], \phi)$ (? is absorbing for union)
$= items((q, \phi)[\texttt{delete}])$ (Definition 5.4)
$\supseteq items(q, \phi) \neq \emptyset$ (Lemma 5.3)
$\neq \emptyset$.

name: $items((q, \phi)[\texttt{name ?}v])$
$= items(q[\phi \texttt{ and } \underline{?v}_{\phi'}], \phi')$ (Definition 5.4)
$= items(flip(q[\phi \texttt{ and } \underline{?v}_{\phi'}], \phi'))$ (Definition 5.2)
$= items(flip(q, \phi) \texttt{ and ?}v)$ (Lemma 5.1)
$= items(flip(q, \phi))$ (because v is a fresh variable)
$= items(q, \phi) \neq \emptyset$.

reference: $items((q, \phi)[\texttt{ref ?}v]) \neq \emptyset$ by Definition 5.4.

delete: $items((q, \phi)[\texttt{delete}])$
$\supseteq items(q, \phi)$ (Lemma 5.3)
$\neq \emptyset$. □

We justify to allow for unsafe focus changes by considering the following navigation scenario. The current query has the form $q = \underline{f_1} \texttt{ or } \underline{f_2}_\phi$, i.e., the union of two restrictions. The feature f_3 is a restriction of q such that $items(f_2) \cap items(f_3) = \emptyset$, i.e., only items of f_1 match f_3. The intersection with f_3 leads to the query $q' = (f_1 \texttt{ or } f_2) \texttt{ and } \underline{f_3}_{\phi'}$, and a focus change on f_2

navigation link	notation (l)	target ($(q', \phi') = (q, \phi)[l]$)	conditions
focus change	`focus` ϕ'	(q, ϕ')	for every focus $\phi' \in \Phi(q)$
intersection	`and` f	$(q[\phi$ `and` $\underline{f}_{\phi'}], \phi')$	for every $f \in restr(q, \phi)$
exclusion	`and not ?`	$(q[\phi$ `and not` $\underline{?}_{\phi'}], \phi')$	
union	`or ?`	$(q[\phi$ `or` $\underline{?}_{\phi'}], \phi')$	
name	`name ?`v	$(q[\phi$ `and` $\underline{?v}_{\phi'}], \phi')$	for some fresh variable v
reference	`ref ?`v	$(q[\phi$ `and` $\underline{?v}_{\phi'}], \phi')$	for every $v \in vars(q)$ s.t. $items(q', \phi') \neq \emptyset$
deletion	`delete`	$(q[\phi := \texttt{?}], \phi)$	

Table 4 Definition of the set of navigation links linking a navigation place (q, ϕ) to the target navigation place (q', ϕ').

leads to an empty selection. We could prevent intersection with f_3 but this would be counter-intuitive because it is a valid restriction for (q, ϕ). We could simplify the query q' by removing the second alternative f_2 ($q' = f_1$ `and` f_3), or forbid the focus change, but we think users should have full control on the query they have built. Finally, allowing for the unsafe focus change is a simple way to inform users that no item of f_2 matches the new restriction feature f_3.

5.2 Completeness

Completeness is a key contribution of our approach, and is based on the explicit definition of a query language. A navigation graph is complete if there is a navigation path to every query that is not a dead-end, starting from the initial navigation place p_0. Completeness is important because it completely removes the need to manually edit queries, which makes guided navigation as expressive as the query language, here LISQL. This suggests that, in order to improve the expressiveness of semantic faceted search, one should first extend the query language, then extend the navigation graph with additional navigation links (i.e., query transformations), and finally prove safeness and completeness.

To be precise, completeness is proved for *safe* queries, i.e., queries without an unsafe focus change.

Definition 5.5: A query q is said to be *safe under* $\phi \in \Phi(q)$, which we note $safe(q, \phi)$, if for every focus $\phi' \in \Phi(q)$ that is equal to or under ϕ in the syntax tree of q, we have $items(q, \phi') \neq \emptyset$. Query q is said *safe*, which we note $safe(q)$, if it is safe under its root focus. By extension, we say that a navigation place (q, ϕ) is safe if $safe(q, \phi)$ holds.

Before stating and proving the main theorem on completeness, we need a few lemmas on the conservation of the safeness of queries when they are simplified. The proofs, not given here, are based on the translation of queries to SPARQL (Definition 4.4).

Lemma 5.4: *For every query q, and focus $\phi \in \Phi(q)$, if $safe(q, \phi)$, then the following propositions hold:*

1. $safe(q[\phi := \texttt{?}], \phi)$,
2. $q[\phi] = (q_1$ `and` $q_2) \Rightarrow safe(q[\phi := q_1], \phi)$,
3. $q[\phi] = (q_1$ `or` $q_2) \Rightarrow safe(q[\phi := q_1], \phi)$,
4. $q[\phi] = (p$ `:` $q_1) \Rightarrow safe(q[\phi := p$ `: ?`$], \phi)$,
5. $q[\phi] = (p$ `of` $q_1) \Rightarrow safe(q[\phi := p$ `of ?`$], \phi)$.

Theorem 2: *Let D be a RDF dataset. The navigation graph G_D is* complete *except for some queries having an unsafe focus change, i.e., for every safe query q, there is a navigation path from the initial navigation place p_0 to the navigation place $(\underline{q}_\phi, \phi)$.*

Proof: Suppose we have a navigation link [`and` q] for every safe query q, with the same definition as intersection with a restriction [`and` f] (Table 4). Then it would be possible to navigate (in one step) from the initial place p_0 to the place (`?` `and` $\underline{q}_\phi, \phi)$, which is equivalent to $(\underline{q}_\phi, \phi)$.

Therefore, we can prove completeness by showing how the hypothetical navigation link [`and` q] can be decomposed into a valid and finite navigation path, for every safe query q. The following table defines such a finite decomposition by induction on the structure of q.

q	[`and` $\underline{q}_\phi$]
r	[`and` r]
`a` c	[`and a` c]
`?`v	[`name ?`v] (if v is new)
	[`ref ?`v] (otherwise)
`?`	ϵ
p `:` q_1	[`and` p `: ?`$_{\phi_1}$][`focus` ϕ_1][`and` q_1][`focus` ϕ]
p `of` q_1	[`and` p `of ?`$_{\phi_1}$][`focus` ϕ_1][`and` q_1][`focus` ϕ]
`not` q_1	[`and not ?`$_{\phi_1}$][`and` q_1][`focus` ϕ]
q_1 `or` q_2	[`and` $\underline{q_1}_{\phi_1}$][`or ?`$_{\phi_2}$][`and` q_2][`focus` ϕ]
q_1 `and` q_2	[`and` q_1][`and` q_2][`focus` ϕ]

It remains to prove that every navigation step of the decomposition is a valid navigation link, according to Table 4. The first step is to prove that every intermediate navigation place is safe. This can be done by recurrence, where the recurrence hypothesis says that for every subquery, the initial and final places are safe. This is true for the whole query q (the base case), because q is assumed safe under ϕ, and p_0 is trivially safe. For subqueries that are decomposed into several navigation steps, the intermediate vertices can be proved safe by applying Lemma 5.4.

Finally, every atomic navigation link can be proved valid. Focus change, exclusion, union, and deletion are always valid navigation links. For intersection and reference, we can use the fact that every intermediate place is safe, and therefore that every intermediate place has a non-empty set of items. This is enough for reference. For intersection, we can use Lemma 5.2 to prove that the feature f is a restriction. □

This proof also provides an algorithm for finding a path from the initial navigation place to the target query q. It exhibits a linear complexity: the path has a length that is linear in the size of the target query.

The restriction of completeness to safe queries is not a problem in practice because every query q such that $items(q) \neq \emptyset$ (the query has answers) and not $safe(q)$ (the query has no answer at some focus) can be simplified into a safe yet equivalent query. It suffices to delete from the query empty alternatives, and empty exclusions. An empty exclusion is a subquery (`not` $\underline{q_1}_{\phi_1}$) s.t. $items(q, \phi_1) = \emptyset$; and an empty alternative is either q_1 or q_2 in a subquery ($\underline{q_1}_{\phi_1}$ `or` $\underline{q_2}_{\phi_2}$) s.t. $items(q, \phi_1) = \emptyset$ or $items(q, \phi_2) = \emptyset$. The simplified query is equivalent to the original query in that it has the same set of items at every remaining focus, and it is safe.

5.3 Efficiency

Each navigation step from a navigation place (q, ϕ) requires the computation of the set of items $items(q, \phi)$, the set of restrictions $restr(q, \phi)$, and the set of navigation links as specified in Definition 5.4. In many cases, the set of items can be obtained efficiently from the previous set of items, and the last navigation link. If the last navigation link was an intersection, Lemma 5.2 shows that the set of items is the result of the intersection that is performed during the computation of restrictions, like in standard faceted search. For an exclusion or a naming, the set of items is unchanged. For a reference, the set of items was already computed at the previous step. Otherwise, for an union or a focus change, the set of items is computed with a LISQL query engine, possibly reusing existing query engines for the Semantic Web (see Section 4.3).

Computing the set of restrictions is equivalent to set-based faceted search, i.e., amounts to compute set intersections between the set of items of the current navigation place and the precomputed set of items of features. The same datastructures and algorithms can therefore be used. As features are LISQL queries, their set of items can be computed like for queries, possibly with optimizations given that features are simple queries. Finally, determining the set of navigation links requires little additional computation. A navigation link is available for each focus of the query (focus change), and each restriction (intersection). Three navigation links for exclusion, union, and naming are always available. Only for reference navigation links it is necessary, for each variable in the query, to compute the set of items of the target navigation place, in order to check it is not empty. This additional cost is limited as the number of variables in a LISQL query is very small in practice, and is bounded by the number of foci of the query.

6 User Interface and Interaction

Query-based Faceted Search has been implemented as a prototype, Sewelis[4]. Figure 2 shows a screenshot of Sewelis. From top to bottom, and from left to right, it is composed of a menu bar (M), a toolbar (T), a query box (Q), query transformations (QT), a suggestion area (S) that is mainly composed of a facet hierarchy (F), a set of value boxes (V), and an answer list (A). A query engine can be derived from Sewelis by retaining only the components Q and A. A standard faceted search system can be derived by retaining only the components A, F, and V.

The query box displays the current LISQL query q, where the current focus ϕ is rendered by highlighting the subquery $q[\phi]$. In Figure 2, the whole query is highlighted because the current focus is the root focus. For a better readability of queries, we use indentation instead of brackets, URIs are represented by their labels if available or abbreviated as qualified names if possible, a concrete syntax is used for some datatypes (e.g., `42` instead of `"42"^^xsd:integer`), and resources are colored according to their kind (orange for classes, purple for properties, blue for other URIs, and green for literals).

The suggestion area (S) contains the set of restrictions for the current set of items. In fact, that set of items is a subset of the restrictions, and is displayed in the answer list (A). The classic restrictions, pairs facet-value, are found in value boxes (V), one for each applicable property (p `: ?`) and inverse property (p `of ?`). If that property is transitive (an instance of the class `owl:TransitiveProperty`), then the values are organized hierarchically accordingly. For instance, the value box of `ancestor : ?` displays a descendancy chart, while the value box of `ancestor of ?` displays an ancestry chart. Similarly, the value box of `part of ?` displays a taxonomy of locations. Other kinds of restrictions are placed in the facet box (F). This includes co-reference variables (e.g., `?X`), classes as types (e.g., `a man`), and properties as domains (e.g., `parent : ?`) and ranges (e.g., `child of ?`). Classes and properties are hierarchically organized according to the `rdfs:subClassOf` and `rdfs:subPropertyOf` transitive properties. For instance, in Figure 2, the class `a man` is a subclass of `a person`; and the properties `father : ?` and `mother : ?` are subproperties of `parent : ?`, which is itself a subproperty of `ancestor : ?`. In order to increase user feedback about suggestions, each restriction is prefixed by the number of answers that match that restriction. If a restriction matches all answers, that number is highlighted like the focus. If two restrictions match the same subset of answers, this is indicated by highlighting the two numbers with a same color. Finally, a dim font is used for restrictions that are included in the current query, hence emphasizing the other restrictions as "new". For example, in Figure 2, the restriction `a man` (bold font) is a way to make the query more specific, while `a person` (dim font) is already in the query (at the current focus). The buttons "More" and "Less" are out the

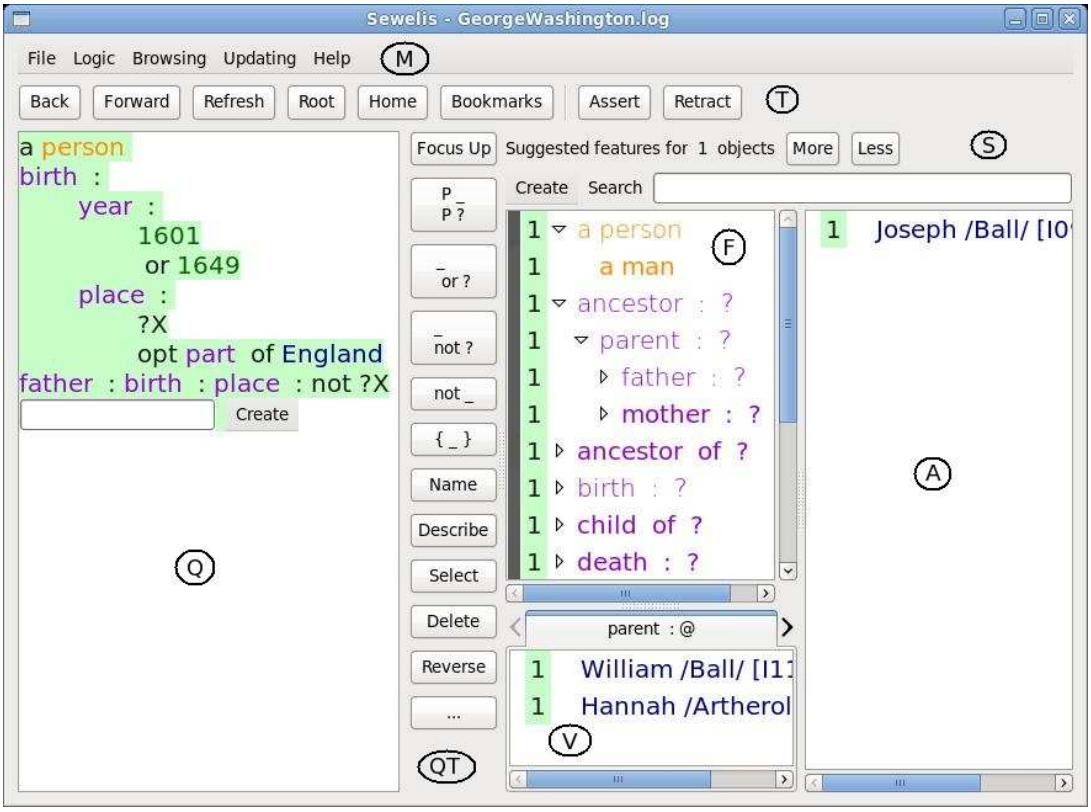

Figure 2 A screenshot of the user interface of Sewelis. It shows the selection of "persons born in 1601 or 1649 somewhere in England, and whose father was born at another place".

scope of this paper, and are mostly used when editing the RDF graph (Hermann et al. 2011).

Navigation links, i.e. the application of query transformations, are available on most components (T,Q,QT,S). Focus changes can be triggered by clicking on the subquery of interest in Q, by keystrokes (CTRL + arrows), or by pushing the button "Focus Up" in QT. Intersection with a feature and reference to a variable can be performed by double-clicking a restriction in the suggestion area (S). Other navigation links are available as buttons in the component QT (Query Transformations): "_ or ?" for union, "_ and not ?" for exclusion, "Name" for naming, and "Delete" for deletion. Other useful transformations are available in QT and T areas: "Back" and "Forward" for navigating in the history of navigation places, "Root" for jumping to the initial navigation place that contains all resources of the dataset, "Home" for jumping to the user-defined home query, "P _ and P ?" for adding a value to the same property (e.g., for describing a second child), "not _" for (un)applying negation on the current subquery, "{ _ }" for (un)quoting the current subquery as a literal, "Describe" for replacing a resource by its full description in the query, "Select" for selecting the subquery and removing the rest of the current query, "Reverse" for reformulating the query from the current focus. Other buttons ("Assert", "Retract", "...") are used for edition.

The entry field at the top of the suggestion area (S) and below the focus in the query box (Q) enables to find and select a restriction by auto-completion. This is useful when the number of restrictions is high, and the user has a clue on the text of the restriction. Matching is performed on the syntax of restrictions as rendered in the user interface, which is based on labels (using `rdfs:label`). For instance, the URI of George Washington can be retrieved by entering any of "Georges Wash", "geo wa", or even "Wash ge". The list of possible completion is updated after every keystroke. The "Create" menu near entry fields gives access to domain-specific dialogs for choosing dates, times, and filenames.

In Sewelis, data can be loaded either by importing RDF files in various formats (RDF/XML, N-Triples, Turtle), or by dereferencing URIs according to the Linked Data[5] principles. The former is available in the "File" menu, and the latter is available on every restriction that includes a URI through the contextual menu. The same contextual menu also provides means to improve the presentation of restrictions by defining labels and namespaces. Sewelis supports RDFS inference, as well as some OWL inference on properties (`owl:TransitiveProperty`, `owl:SymmetricProperty`, `owl:inverse`).

7 Usability Evaluation

This section reports on the evaluation of QFS in terms of usability[6]. We have measured the ability of users to answer questions of various complexities, as well as their response times. Results are strongly positive and demonstrate that QFS offers expressiveness and ease-of-use at the same time.

Dataset. The datasets were chosen so that subjects had some familiarity with the concepts, but not with the individuals. We found genealogical datasets about former US presidents, and converted them from GED to RDF. We used the genealogy of Benjamin Franklin for the training, and the genealogy of George Washington for the test. The latter describes 79 persons by their birth and/or death events, which are themselves described by their year and place, by their firstname, lastname, and sex, and by their relationships (father, mother, child, spouse) to other persons. Places are linked by a transitive *part-of* relationship, allowing for the display of place hierarchies in Sewelis.

Methodology. The subjects consisted of 20 graduate students in computer science. They had prior knowledge of relational databases but neither of Sewelis, nor of faceted search, nor of Semantic Web, nor of US presidents. None was familiar with the dataset used in the evaluation. The evaluation was conducted in three phases. First, the subjects learned how to use Sewelis through a 20min tutorial, and had 10 more minutes for free use and questions. Second, subjects were asked to answer a set of questions, using Sewelis. We recorded their answers, the queries they built, and the time they spent on each question. Finally, we got feedback from subjects through a SUS questionnaire (System Usability Scale (Brooke 1996)) and open questions. The test was composed of 18 questions, with smoothly increasing difficulty. Table 5 groups the questions in 7 categories: the first 2 categories are covered by standard faceted search, while the 5 other categories are not in general. The first category, *Visualization*, did not require the creation of a query. The exploration of the suggested restrictions was sufficient: e.g., "How many men are there?". In the second category, *Selection*, we asked to count or list items that have a particular feature: e.g., "How many women are named Mary?". In the third category, *Path*, subjects had to follow a path of properties: e.g., "Which man is married with a woman born in 1708?". The fourth category, *Disjunction*, required to use unions: e.g., "Which women have for mother Jane Butler or Mary Ball?". The fifth category, *Negation*, required to use exclusions: e.g., "How many women have a mother whose death's place is not Warner Hall?". The sixth category, *Inverse*, required the backward crossing of a property: e.g., "Who was born in the same place as Robert Washington?". In the seventh category, *Cycle*, required the use of co-references: e.g., "How many persons have the same firstname as one of their parent?".

Results. Figure 3 shows the number of correct queries and answers, the average time spent on each question and the number of participants who had a correct query for at least one question of each category. For example, in category "Visualization", the first two questions had 20 correct answers and queries; the third question had 10 correct answers and 13 correct queries; all the 20 participants had a correct query for at least one question of the category; the average response times were respectively 43, 21, and 55 seconds. The difference between the number of correct queries and correct answers is explained by the fact that some subjects forgot to set the focus on the whole query after building the query, which we know from the navigation trace of subjects.

All subjects but one had correct answers to more than half of the questions. Half of the subjects had the correct answers to at least 15 questions out of 18. Two subjects answered correctly to 17 questions, their unique error was on a disjunction question for one and on a negation question for the other. All subjects had the correct query for at least 11 questions. Among all questions, the worst success rate is 50 percent. The subjects spent an average time of 40 minutes on the test, the quickest one spent 21 minutes and the slowest one 58 minutes.

The first 2 categories corresponding to standard faceted search, visualization and selection, had a high success rate (between 94 and 100) except for the third question. The most likely explanation for the latter is that the previous question was so simple (`a man`) that subjects forgot to reset the query between questions 2 and 3 (we know this from the navigation traces). All questions of the first two categories were answered in less than 1 minute and 43 seconds on average. Those results indicate that the more complex user interface of QFS does not entail a loss of usability compared to standard faceted search for the same tasks.

For other categories, all subjects but two managed to answer correctly at least one question of each category. Within each category, we observed that response times decreased, except for the *Cycle* category. At the same time, for *Path*, *Disjunction* and *Inverse*, the number of correct answers and queries increased. Those results suggest a quick learning process of the subjects. The decrease in category *Negation* is explained by a design flaw in the interface. For category *Cycle*, we conjecture some lassitude at the end of the test. Nevertheless, all but two subjects answered correctly to at least one of *Cycle* questions. The peak of response time in category *Inverse* is explained by the lack of inverse property examples in the tutorial. It is noticeable that subjects, nevertheless, managed to solve the *Inverse* questions with a reasonable success rate, and a decreasing response time.

SUS Questionnaire. Table 6 shows the answers to the SUS questions, which are quite positive. The first noticeable thing is that, despite the relative complexity of the user interface, subjects do not find the system *unnecessarily complex* nor *cumbersome to use*. We think this is because the principles of QFS are very regular, i.e., they

Category	Question (# navig. links)
Visualization	1 How many persons are there? (0) 2 How many men are there? (0) 3 How many persons have a birth's place in the base? (0)
Selection	4 How many women are named Mary? (4) 5 Who was born at Stone Edge? (4) 6 Which man was born in 1659? (5) 7 Who is married with Edward Dymoke? (3)
Path	9 Which man has his father married with Alice Cooke? (5) 11 Which man is married with a woman born in 1708? (7)
Disjunction	8 Which women have for mother Jane Butler or Mary Ball? (6) 12 Which men are married with a woman whose birth's place is Cuckfields or Stone Edge? (9)
Negation	10 How many men were born in the 1600 or 1700 years, and not in Norfolk? (12) 13 How many women have a mother whose death's place is not Warner Hall? (7)
Inverse	14 Who was born in the same place as Robert Washington? (6) 15 Who died during the year when Augustine Warner was born? (6)
Cycle	16 Which persons died in the same area where they were born? (9) 17 How many persons have the same firstname as one of their parent? (8) 18 Which persons were born the same year as their spouse? (10)

Table 5 Questions of the test, by category, and the minimum number of navigation links to answer them.

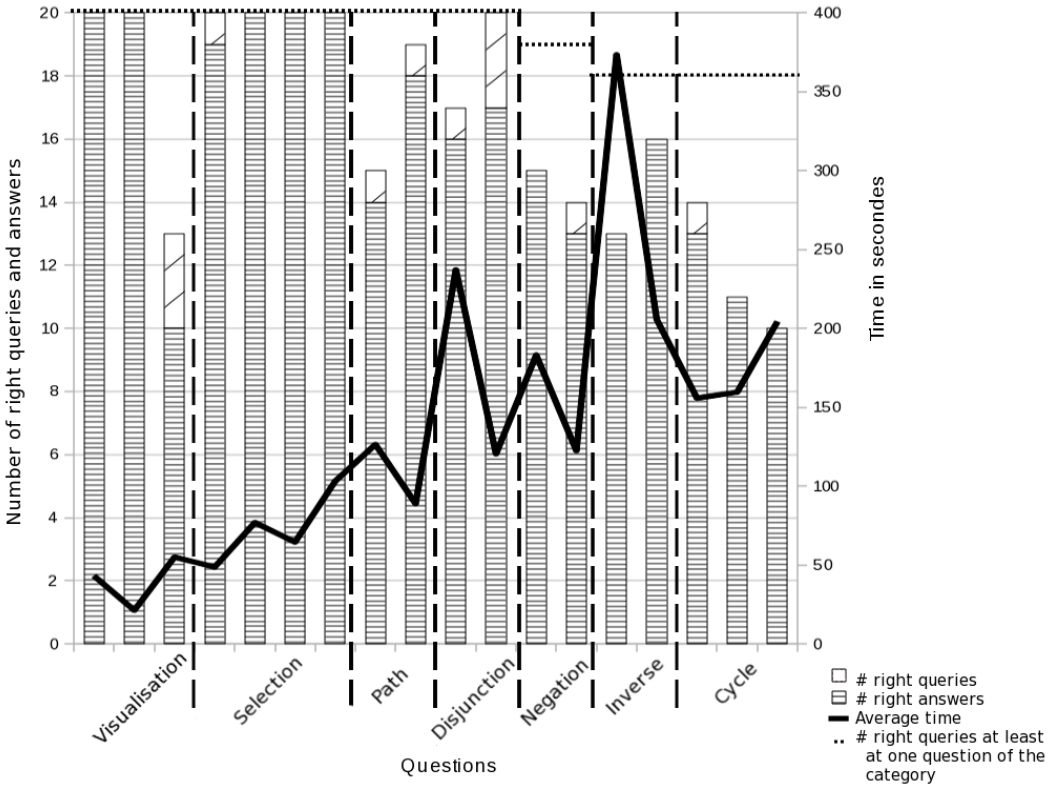

Figure 3 Average time and number of correct queries and answers for each question

follow few rules with no exception. The second noticeable thing, which may be a consequence of the first, is that subjects *felt confident using the system* and found no *inconsistency*. Finally, even if it is necessary for subjects to learn how to use the system, they *thought that the system was easy to use*, and that they *would learn to use it very quickly*. The results of the test demonstrate that they are right, even for features that were not presented in the tutorial (the Inverse category).

SUS Question	(polarity)	Score (on a 0-4 scale)	
I think that I would like to use this system frequently	+	2.8	Agree
I found the system unnecessarily complex	−	0.8	Strongly disagree
I thought the system was easy to use	+	2.6	Agree
I think that I would need the support of a technical person to use this system	−	1.5	Disagree
I found the various functions in this system were well integrated	+	2.9	Agree
I thought there was too much inconsistency in this system	−	0.6	Strongly disagree
I would imagine that most people would learn to use this system very quickly	+	2.5	Agree
I found the system very cumbersome to use	−	1.0	Disagree
I felt very confident using the system	+	2.8	Agree
I needed to learn a lot of things before I could get going with this system	−	1.7	Neutral

Table 6 Results of SUS questions.

8 Related Work

We discuss other approaches for applying and extending faceted search to the Semantic Web. We also compare the expressiveness of LISQL with two expressive query languages of the Semantic Web: SPARQL and SPARQL-DL.

8.1 Faceted Search for the Semantic Web

As faceted search is becoming widespread, a number of proposals have been made to apply it on the Semantic Web (SW). They all have in common to assume that data is represented in a SW format, either RDF(S) or OWL. Most of them, such as Ontogator (Mäkelä et al. 2006), mSpace[7], and Longwell[8], do not claim for a contribution in term of expressiveness, and contribute either to the design of better interfaces and visualizations, or to methods for the rapid or user-centric configuration of faceted views (Suominen et al. 2007). Therefore, their contributions are somewhat orthogonal to ours, and could certainly complement them. Other approaches, such as SlashFacet (Hildebrand et al. 2006) and BrowseRDF (Oren et al. 2006), extend faceted search towards a more expressive navigation.

The most essential ingredient for an expressive and flexible semantic search in RDF graphs is *focus change*. It allows to change the perspective without changing the underlying graph pattern. To the best of our knowledge, no faceted search system offers this in a general way. SlashFacet has the *crossing* operation that selects the images of the items in the current selection through a property. Crossing includes a focus change, but crossing back a property is not equivalent to a focus change, because it introduces an additional restriction: starting from q and crossing p `:` `?` and then p `of` `?` leads to p `:` p `of` q instead of q `and` p `:` `?` (they are not equivalent). Other systems allow to focus on different types of items, but this focus cannot be changed in the course of a search. For example, in a dataset about publications, a choice has to be made between authors and documents.

It is generally considered that the query should be hidden from the interface. In fact, in most faceted search systems, the query *is* displayed as the list of the restriction values users have already selected in the course of their search. This is important so that users do not feel lost, and can easily reverse previous selections. On our case, the query is also important to specify focus changes. Of course, displaying the query in SPARQL would ruin those benefits: the display of the query is part of the design of the user interface. Now, when the expressiveness is raised to SPARQL with graph patterns, disjunction, and negation, it becomes necessary to introduce syntax. While, in Sewelis, the query is simply rendered as a sentence following some grammar, nothing prevents to render syntax through graphical widgets (e.g., lists for conjunction, trees for restrictions, tab panels for disjunction). In our approach, LISQL is used to render the query in a way that fits query-based faceted search (see Section 4.1).

Disjunction and negation are either absent or strongly limited in existing approaches. Disjunction is restricted to build sets of values or sets of items, e.g., in SlashFacet. Negation is restricted to restriction values, and also applies to unqualified restrictions (e.g., `not father : ?`) in BrowseRDF. No other system allows to form cycles as we do with co-references.

The value boxes of SlashFacet can handle only one taxonomy of values, whereas we can use any transitive property that link the values together. For instance, when values are persons, we can use either `ancestor :` (descendancy chart), or `ancestor of` (ancestry chart).

8.2 Query Languages for the Semantic Web

We compare our query language LISQL to SPARQL, as the reference query language for the Semantic Web, and to SPARQL-DL (Sirin & Parsia 2007) for the syntactic similarity of complex classes with LISQL queries.

8.2.1 Comparison with SPARQL

Haase et al. (2004) define a set of 14 use cases for comparing the expressiveness of RDF query languages. We use them to evaluate and compare the expressiveness of SPARQL and LISQL. First, a significant difference is that LISQL has mono-dimensional queries, i.e., LISQL

queries are translated to SPARQL queries having a single variable after SELECT. This constraint comes from the nature of faceted search, not from LISQL itself as several foci could be selected to have several variables after SELECT. The facet hierarchy, the value boxes, and a highlighting mechanism compensate for this constraint. Assume users want to know who is the mother of each male Washington. They first navigate to the query `a man and lastname : Washington`. Then, they expand the facet `mother : ?` in the facet hierarchy, which opens a value box that lists the mothers of male Washingtons, and for each mother, tells how many children she has among them. The associations between male Washingtons and their mothers are accessible by a dynamic highlighting mechanism. When selecting a male Washington (in the extension box), his mother is highlighted in the value box. Symmetrically, when a mother is selected in the value box, her children are highlighted in the extension box.

The use cases that SPARQL and LISQL have in common are path expressions (e.g., "the name of the author of some publication X"), union, partial support for collections and containers, support for literals, and entailment through class and property hierarchies. Compared to SPARQL, LISQL has not the OPTIONAL construct because it is useless in one-dimensional queries. However, it covers the difference use case with the complement construct (`not`), and recursion through transitive properties. The difference use case is covered in extensions of SPARQL with the operator MINUS of Angles & Gutierrez (2008), or the operator NOT EXISTS of SPARQL 1.1. The recursion use case is covered in nSPARQL (Pérez et al. 2008), an extension of SPARQL with *nested regular expressions*. The reification use case is covered by SPARQL: e.g., "the person who has classified the publication X". As defined in Section 4.1, LISQL does not cover it, but its implementation in Sewelis does. The LISQL query for the previous example is (`a publication and ?X and topic [classifier : ?] : ?`), where the subquery into square brackets after `topic` put a constraint on the reified triple whose predicate is `topic`. This query can be navigated to, in the same way as other queries.

In total, SPARQL scores 9.5/14, LISQL scores 8/14, as defined in Section 4.1, and scores 10/14, as implemented in Sewelis. In fact, SPARQL and LISQL have a similar expressiveness, and most differences can be removed by extending either language: adding difference and recursion to SPARQL; adding multiple foci and optional pattern to LISQL.

8.2.2 Comparison with SPARQL-DL

Syntactically, LISQL queries are similar to complex classes as defined in OWL-DL. This suggests that SPARQL-DL (Sirin & Parsia 2007) could be used instead of SPARQL to translate from the LISQL syntax. However, this is not possible because SPARQL-DL is restricted to conjunctive queries, and variables cannot occur in complex classes. On one hand, a LISQL query that contains unions and complements but no variables (hence no cycles) and the root focus, can be translated to a SPARQL-DL query in the form `Type(?x,`q`)`, where q is a complex class that has the same abstract syntax as the LISQL query. For example, the LISQL query `a man and birth : (year : (1601 or 1649) and place : not part of England)` can be translated to

```
Type(?x, and(
   man,
   some(birth, and(
     some(year, or({1601},{1649})),
     some(place, not(some(partOf, {England}))
))))).
```

On the other hand, a LISQL query that contains variables but neither union nor complement, can be translated in a similar way to SPARQL-DL, using in fact the common subset between SPARQL and SPARQL-DL. For example, the LISQL query `a man and father : ?X and mother : spouse of ?X` can be translated to

```
Type(?z,man),
PropertyValue(?z,father,?x),
PropertyValue(?z,mother,?y),
PropertyValue(?x,spouse,?y).
```

The two kinds of translations cannot be reconciled in the general case, in particular when variables occur in the scope of unions or complements.

In fact, SPARQL-DL and LISQL work at different levels, and might complement each other by benefiting from a comparable syntax. SPARQL-DL, like OWL-DL, works at the *intentional* level, whereas LISQL and SPARQL work at the *extensional* level. The intentional level is associated to open world assumption, and ontological reasoning. The extensional level is associated to closed world assumption, and query answering over a unique and finite interpretation, namely a RDF graph.

9 Conclusion

We have introduced *Query-based Faceted Search* (QFS) as a search paradigm for Semantic Web knowledge bases, in particular RDF graphs. It combines the expressiveness of the SPARQL query language, and the benefits of exploratory search and faceted search. Exploratory search is formalized as a navigation graph, where navigation places are queries, and navigation links are query transformations. The navigation graph is proved to be safe, because whatever the path of navigation links, the current set of items is never empty. It is also proved complete w.r.t. the query language, because for every safe query, there is a navigation path that leads to it. Finally, it is as efficient as standard faceted search w.r.t. the computation of facets and restrictions. The completeness proof

is the key result here because it draws an equivalence between expressive querying and exploratory search, therefore totally freeing users from editing queries, even the most complex ones.

The user interface of QFS includes the user interface of other faceted search systems, and can be used as such. It adds a query box to tell users where they are in their search, and to allow them to change the focus or to remove query parts. It also adds a few controls for applying some query transformations such as insertion/deletion of unions, complements, and co-references. Query transformations determines a new query language, LISQL, that is similar to SPARQL in terms of expressiveness, and with a more compact syntax. Beside the list of selected items, the user interface has a hierarchy of facets organizing classes and properties by subsumption, and value boxes that can be displayed as flat lists or as various taxonomies automatically derived from the dataset.

QFS has been implemented as a prototype, Sewelis. Its usability has been demonstrated through a user study, where, after a short training, all subjects were able to answer simple questions, and most of them were able to answer complex questions involving disjunction, negation, or cycles. This means semantic faceted search retains the ease-of-use of other faceted search systems, while offering the expressiveness of query languages such as SPARQL.

We think that QFS is not tied to LISQL, and could be adapted to other query languages. Indeed, the definition of a navigation graph only requires the definition of the answers of a query, and the definition of query transformations. The hard part is then to prove that the resulting navigation graph is safe and complete, which we have successfully done here for LISQL.

As future work, our main objective is to fully match the expressiveness of SPARQL 1.1 by extending QFS to the few missing features: multi-dimensional queries and the OPTIONAL construct, aggregations and expressions, and built-in predicates. Other objectives are to integrate Sewelis with existing SW tools (e.g., Solr/Lucence-index for fast literal-indexing), and to perform more user evaluation in order to improve its user interface.

Acknowledgments. We would like to thank the 20 students, from the University of Rennes 1 and the INSA engineering school, for their volunteer participation to the usability evaluation. We also thank the reviewers for their useful remarks.

References

Angles, R. & Gutierrez, C. (2008), The expressive power of SPARQL, *in* A. P. S. *et al*, ed., 'Int. Semantic Web Conf.', LNCS 5318, Springer, pp. 114–129.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D. & Patel-Schneider, P. F., eds (2003), *The Description Logic Handbook: Theory, Implementation, and Applications*, Cambridge University Press.

Brooke, J. (1996), SUS: A quick and dirty usability scale, *in* P. Jordan, B. Thomas, B. Weerdmeester & A. McClelland, eds, 'Usability evaluation in industry', London: Taylor and Francis, pp. 189–194.

Ferré, S. (2009), 'Camelis: a logical information system to organize and browse a collection of documents', *Int. J. General Systems*.

Ferré, S. & Ridoux, O. (2000), A file system based on concept analysis, *in* Y. Sagiv, ed., 'Int. Conf. Rules and Objects in Databases', LNCS 1861, Springer, pp. 1033–1047.

Ferré, S. & Ridoux, O. (2004), 'An introduction to logical information systems', *Information Processing & Management* **40**(3), 383–419.

Ferré, S. & Ridoux, O. (2007), Logical information systems: from taxonomies to logics, *in* 'Int. Work. Dynamic Taxonomies and Faceted Search (FIND)', IEEE Computer Society, pp. 212–216.

Fikes, R., Hayes, P. J. & Horrocks, I. (2004), 'OWL-QL - a language for deductive query answering on the semantic web', *J. Web Semantic* **2**(1), 19–29.

Ganter, B. & Wille, R. (1999), *Formal Concept Analysis — Mathematical Foundations*, Springer.

Haase, P., Broekstra, J., Eberhart, A. & Volz, R. (2004), A comparison of RDF query languages, *in* S. M. *et al.*, ed., 'Int. Semantic Web Conf.', LNCS 3298, Springer, pp. 502–517.

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

Hearst, M., Elliott, A., English, J., Sinha, R., Swearingen, K. & Yee, K.-P. (2002), 'Finding the flow in web site search', *Communications of the ACM* **45**(9), 42–49.

Heim, P., Ertl, T. & Ziegler, J. (2010), Facet graphs: Complex semantic querying made easy, *in* L. A. et al., ed., 'Extended Semantic Web Conference', LNCS 6088, Springer, pp. 288–302.

Hermann, A., Ferré, S. & Ducassé, M. (2011), Guided creation and update of objects in rdf(s) bases, *in* M. A. Musen & Ó. Corcho, eds, 'Int. Conf. Knowledge Capture (K-CAP)', ACM, pp. 189–190.

Hildebrand, M., van Ossenbruggen, J. & Hardman, L. (2006), /facet: A browser for heterogeneous semantic web repositories, *in* I. C. *et al*, ed., 'Int. Semantic Web Conf.', LNCS 4273, Springer, pp. 272–285.

Hitzler, P., Krötzsch, M. & Rudolph, S. (2009), *Foundations of Semantic Web Technologies*, Chapman & Hall/CRC.

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

Lu, J., Ma, L., Zhang, L., Brunner, J., Wang, C., Pan, Y. & Yu, Y. (2007), SOR: A practical system for ontology storage, reasoning and search (demo), *in* 'Int. Conf. Very Large Databases (VLDB)', VLDB Endowment, ACM, pp. 1402–1405.

Mäkelä, E., Hyvönen, E. & Saarela, S. (2006), Ontogator - a semantic view-based search engine service for web applications, *in* I. F. C. *et al.*, ed., 'Int. Semantic Web Conf.', LNCS 4273, Springer, pp. 847–860.

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

Oren, E., Delbru, R. & Decker, S. (2006), Extending faceted navigation to RDF data, *in* I. C. *et al*, ed., 'Int. Semantic Web Conf.', LNCS 4273, Springer, pp. 559–572.

Pérez, J., Arenas, M. & Gutierrez, C. (2008), nSPARQL: A navigational language for RDF, *in* A. P. S. *et al*, ed., 'Int. Semantic Web Conf.', LNCS 5318, Springer, pp. 66–81.

Sacco, G. M. (2000), 'Dynamic taxonomies: A model for large information bases', *IEEE Transactions Knowledge and Data Engineering* **12**(3), 468–479.

Sacco, G. M. (2006), Some research results in dynamic taxonomy and faceted search systems, *in* 'Faceted Search Work. at ACM SIGIR 2006', ACM.

Sacco, G. M. & Tzitzikas, Y., eds (2009), *Dynamic taxonomies and faceted search*, The information retrieval series, Springer.

Sirin, E. & Parsia, B. (2007), SPARQL-DL: SPARQL query for OWL-DL, *in* C. Golbreich, A. Kalyanpur & B. Parsia, eds, 'Work. OWL Experiences and Directions (OWLED)', Vol. 258, CEUR-WS.

Suominen, O., Viljanen, K. & Hyvönen, E. (2007), User-centric faceted search for semantic portals, *in* E. Franconi, M. Kifer & W. May, eds, 'Eu. Semantic Web Conf.', LNCS 4519, Springer, pp. 356–370.

Tran, T., Wang, H. & Haase, P. (2009), 'Hermes: Data web search on a pay-as-you-go integration infrastructure', *Web semantics: Science, Services and Agents on the World Wide Web* **7**, 189–203.

van Rijsbergen, C. J. (1986), A new theoretical framework for information retrieval, *in* 'Int. ACM SIGIR Conference on Research and Development in Information Retrieval', ACM, pp. 194–200.

Note

[1]SPARQL `http://www.w3.org/TR/rdf-sparql-query/`
[2]The SCRIBO graphical editor `http://www.scribo.ws/-xwiki/bin/view/Blog/SparqlGraphicalEditor`
[3]GED files `http://jay.askren.net/Projects/SemWeb/`
[4]Sewelis: see `http://www.irisa.fr/LIS/softwares/-sewelis/` for a presentation, screencasts, a Linux executable, and sample data.
[5]Linked Data `http://linkeddata.org/`
[6]Usability evaluation: details can be found on `http://-www.irisa.fr/LIS/alice.hermann/camelis2.html`
[7]mSpace `http://mspace.fm/`
[8]Longwell `http://simile.mit.edu/wiki/Longwell`