Conceptual Navigation in Large Knowledge Graphs

Sébastien Ferré[⋆]

Abstract A growing part of Big Data is made of knowledge graphs. Major knowledge graphs such as Wikidata, DBpedia or the Google Knowledge Graph count millions of entities and billions of semantic links. A major challenge is to enable their exploration and querying by end-users. The SPARQL query language is powerful but provides no support for exploration by end-users. Question answering is user-friendly but is limited in expressivity and reliability. Navigation in concept lattices supports exploration but is limited in expressivity and scalability. In this paper, we introduce a new exploration and querying paradigm, Abstract Conceptual Navigation (ACN), that merges querying and navigation in order to reconcile expressivity, usability, and scalability. ACN is founded on Formal Concept Analysis (FCA) by defining the navigation space as a concept lattice. We then instantiate the ACN paradigm to knowledge graphs (Graph-ACN) by relying on Graph-FCA, an extension of FCA to knowledge graphs. We continue by detailing how Graph-ACN can be efficiently implemented on top of SPARQL endpoints, and how its expressivity can be increased in a modular way. Finally, we present a concrete implementation available online, Sparklis, and a few application cases on large knowledge graphs.

Sébastien Ferré
Univ Rennes, CNRS, IRISA, Campus de Beaulieu, 35042 Rennes, France
e-mail: ferre@irisa.fr

[⋆] This research is supported by ANR project PEGASE (ANR-16-CE23-0011-08).

1 Introduction

A growing part of Big Data is made of knowledge graphs. The World Wide Web Consortium[2] has defined a number of standards for representing them (RDF), reasoning about them (RDFS and OWL), and querying them (SPARQL) [18]. A knowledge graph is a collection of entities and values interlinked with semantic relationships. In RDF, every link is a *triple* (source entity, relation, target entity). Examples of entities are *France* and *Paris*, and examples of triples are *(France, capital, Paris)* and *(France, population, 66000000)*. The notion of knowledge graph has been popularized by the Google Knowledge Graph, and the several web search engines have agreed on the `schema.org` vocabulary to support semantic search [25], i.e. search in terms of entities and relationships rather than search in terms of pages and keywords. The two main open sources of knowledge graphs are Linked Open Data[3] (LOD) and microdata[4]. Both count more than 30 billions of triples each. Examples of large open datasets are DBpedia, Wikidata, or YAGO, each of which contains billions of triples.

A major challenge with knowledge graphs is to enable their exploration and querying by people who are interested in the data but are not necessarily proficient in the semantic technologies. A simple kind of exploration is to *browse* the knowledge graph, surfing from entity to entity by following links. However, this kind of exploration can only answer the simplest questions, like *"What is the capital of Paris?"* or *"What is the birth date of the president of the commission of the European Union?"*. We are interested in answering questions that involve sets of entities, and shared properties, like *"Who are the actors playing most often in films directed by Tim Burton since 2000?"*. The SPARQL query language [31] allows to answer a wide range of questions (high expressivity) but it is a formal language targeted at computer scientists. Even for SPARQL practitioners, writing SPARQL queries is a tedious task because of the need to know the data schema, and because of inevitable trial and errors. Question Answering (QA) approaches [19] are attractive because they rely on spontaneous natural language but they lack expressivity and reliability in practice, because of the challenge of natural language understanding. There exist yet other approaches to help with the exploration and querying of knowledge graphs (e.g., semantic faceted search [17, 2], query builders [20]) but they all tend to trade expressivity for usability.

Formal Concept Analysis (FCA) [14] automatically defines from data a navigational structure, called *concept lattice*, which has been used early for data exploration and analysis [15, 4, 12]. Each concept can be understood as the equivalence class of questions that have the same answers. The *extent* of the concept is that set of answers, and the *intent* of the concept

[2] http://www.w3.org/

[3] See https://lod-cloud.net/

[4] See http://webdatacommons.org/

is the most specifc question in the equivalence class. The intent represents what all answers have in common. The lattice structure partially orders the concepts according to inclusion between concept extents, with more general concepts (larger extents, smaller intents) at the top, and more specific concepts (smaller extents, larger intents) at the bottom. The concept lattice can be used for exploration and querying as follows. The user starts at the top concept, i.e. with the empty question, and with all entities as answers. She can then move in the lattice down and up according to the lattice structure. For example, she can move down to the concept of *films*, then down again to the concept of *films directed by Tim Burton*, and again to *films directed by Tim Burton since 2000*. From there, she can move down for each actor playing in those films; or she can move up to *films since 2000*. The major advantage of concept lattices is that they materialize a possibly infinite set of questions into a finite navigational structure that is automatically derived from data. However, their major drawback is that their size explodes with the amount of data, and the range of questions.

In this paper, we show how to reconcile expressivity, usability, and scalability in the exploration and querying of knowledge graphs. Compared to a previous paper [7], our approach has improved a lot in expressivity (from a subset of SPARQL 1.0 to most of SPARQL 1.1), in usability (introducing a natural language interface), and in scalability (from tens of thousands of triples to billions of triples). First, to address expressivity, we present an extension of FCA to knowledge graphs, Graph-FCA, where questions and concept intents are analogous to conjunctive SPARQL queries (Section 2). Second, we introduce *Abstract Conceptual Navigation* (ACN) as an exploration and querying paradigm that reconciles expressivity and usability by relying on the concept lattice to guide end-users in the building of complex queries, and we instantiate it to Graph-FCA (Section 3). Third, we address scalability on large knowledge graphs by leveraging the computation power of SPARQL endpoints (Section 4). From there, we demonstrate how our paradigm can rise in expressivity (e.g., logical operators, expressions, aggregations) in an incremental way without losing on usability and scalability (Section 5). We then present Sparklis, a concrete implementation of our approach available online, and we illustrate it on a few application cases (Section 6). Readers more interested by practical aspects can read this section first, before diving into the more technical sections. Finally, we conclude and draw a few perspectives (Section 7).

2 Graph-FCA: Extending FCA to Knowledge Graphs

Graph-FCA [8, 11] is an extension of FCA for multi-relational data, and in particular for knowledge graphs of the Semantic Web [18]. The specific nature of Graph-FCA is to extract n-ary concepts from a knowledge graph using k-ary relations. The extents of n-ary concepts are sets of n-ary tuples of graph nodes, and their intents are expressed as a graph pattern with n distinguished nodes, which are called the *projected nodes*. For instance, in a knowledge graph that represents family members with a "parent" binary relation, the "sibling" binary concept can be discovered, and described as *"a pair of persons having a parent in common"*.

Classical FCA corresponds to the case where all relations are *unary* ($k = 1$), i.e. when graph nodes are disconnected, and where all concepts are unary too ($n = 1$), i.e. when concept extents are sets of graph nodes. Graph-FCA differs from Graal [24] and applications of Pattern Structures to graphs [22] in that here objects are the nodes of one large knowledge graph, instead of having each object being described by a small graph. Graph-FCA shares theoretical foundations with the work of Kötters [21], where PGPs are called windowed structures. Another extension of FCA to multi-relational data is Relational Concept Analysis (RCA) [28]. Compared to Graph-FCA, RCA is limited to unary and binary relations and to unary concepts, and its concept intents are tree-shaped graph patterns. However, RCA supports various quantifiers (called *relational scaling operators*, while Graph-FCA is limited to existentials.

2.1 Graph Context

Whereas FCA defines a formal context as an incidence relation between objects and attributes, Graph-FCA defines a *graph context* as an incidence relation between *tuples of* objects and attributes. A *graph context* is a triple $K = (O, A, I)$, where O is a set of *objects*, A is a set of *attributes*, and $I \subseteq O^* \times A$ is an *incidence relation* between object tuples $\overline{o} = (o_1, \dots, o_k) \in O^k$, for any arity k, and attributes $a \in A$. $O^* = \bigcup_{k \in \mathbb{N}} O^k = O \cup (O \times O) \cup (O \times O \times O) \cup \dots$ denotes the set of all tuples of objects. A graph context is therefore a labeled multi-hyper-graph, where objects are the nodes, incidence elements are the hyper-edges, and attributes are hyper-edge labels. Note that attributes can be interpreted as k-ary predicates, and graph contexts as First Order Logic (FOL) models (without functions and constants). An hyper-edge $((o_1, \dots, o_k), a)$ can be seen as the FOL *atom* $a(o_1, \dots, o_k)$, and represents a knowledge *fact*. In the following, we indifferently use the terms *hyper-edge*, *atom*, and *fact*. Different kinds of knowledge graphs, such as Conceptual Graphs [30], RDF graphs, or RCA contexts, can be directly mapped to a graph context.

We illustrate Graph-FCA on an example taken from project KNOMANA[5] and used in the RCA literature. It is about plants that treat pests found in African countries, which possess such plants. Figure 1 shows a graphical rep-

[5] https://ur-aida.cirad.fr/nos-recherches/projets-et-expertises/knomana

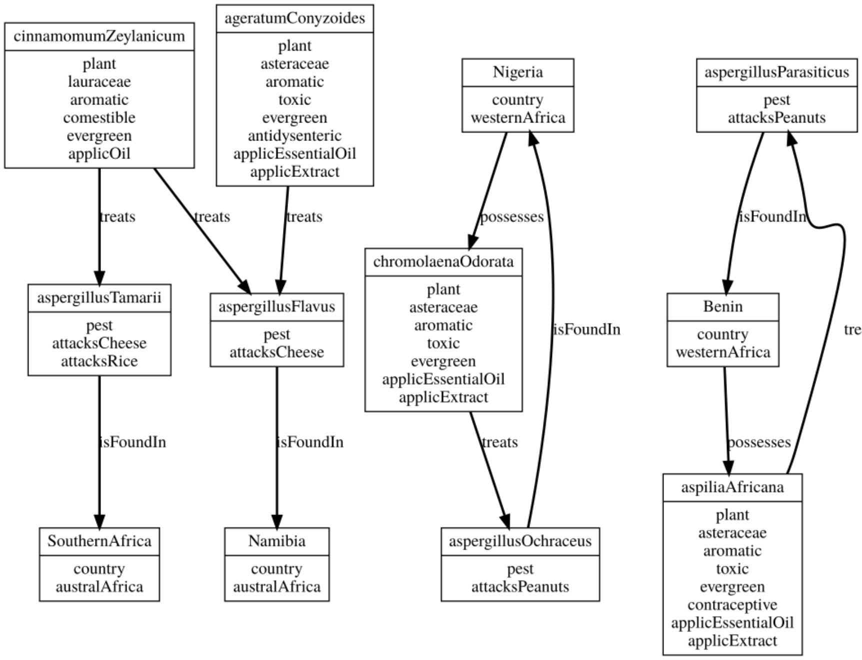


Fig. 1 Graph context about plants, pests, and countries. Rectangles are objects, with their labels in the top part, and unary attributes applying to them in the bottom part. Links are binary edges, with the binary attribute as label.

resentation of the graph context. The objects are plants (e.g. *aspiliaAfricana*), pests (e.g. *aspergillusParasiticus*), and countries (i.e. *Benin*). They are represented as rectangles. The attributes are either unary attributes, like classical FCA attributes (e.g., *plant*, *westernAfrica*, *aromatic*), or binary attributes (e.g., *treats*, *isFoundIn*, *possesses*). The former are represented in the bottom part of object rectangles, and the latter are represented as directed edges between objects. More generally, a unary edge $a(x)$ is represented as a label in the bottom part of the rectangle representing object x; a binary edge $a(x, y)$ is represented by an edge from x to y labeled by a; and other edges $a(x_1, \ldots, x_k)$, for $k > 2$, are represented as ellipses labeled by a, having an edge labeled i to each node x_i.

2.2 Graph Patterns

Whereas FCA is about finding closed sets of attributes, Graph-FCA is about finding closed graph patterns. A *graph pattern* is similar to a graph context but with variables instead of objects as nodes, in order to generalize over particular objects. A key aspect of Graph-FCA is that closure does not apply directly to graph patterns but to *Projected Graph Patterns* (PGP), i.e. graph patterns with one or several distinguished nodes. Those projected nodes define a projection on the occurences of the pattern, like a projection in relational algebra. For example, the PGP $(x, y) \leftarrow \mathit{parent}(x, z), \mathit{parent}(y, z)$ defines a graph pattern with two edges, $\mathit{parent}(x, z)$ and $\mathit{parent}(y, z)$, and with projection on variables x and y. It means that for every occurence of the pattern in the context, the valuation of (x, y) is an occurence of the PGP. It can be used as a definition of the "sibling" relationship, i.e. the fact that x and y are siblings if they have a common parent z.

PGPs are analogous to anonymous definitions of FOL predicates and to conjunctive SPARQL queries. They play the same role as sets of attributes in FCA, i.e. as concept intents. Set operations are extended from sets of attributes to PGPs. PGP inclusion $\subseteq_q$ is based on graph homomorphisms [16]. It is similar to the notion of *subsumption* on queries [5] or rules [26]. PGP intersection $\cap_q$ is defined as a form of graph alignment, where each pair of variables from the two patterns becomes a variable of the intersection pattern. It corresponds to the *categorical product* of graphs (see [16], p. 116), and to the least general generalization of Plotkin [27].

2.3 Graph Concepts

The Galois connection that is the basis for computing concepts is defined in Graph-FCA between n-ary PGPs $(\mathcal{Q}_n, \subseteq_q)$ and n-ary *object relations* $(\mathcal{R}_n, \subseteq)$, where an n-ary object relation is a set of n-ary object tuples ($\mathcal{R}_n = 2^{O^n}$). The connection from PGP $Q \in \mathcal{Q}_n$ to object relation $Q' \in \mathcal{R}_n$ is analogous to query evaluation, and the connection from object relation $R \in \mathcal{R}_n$ to PGP $R' \in \mathcal{Q}_n$ is analogous to relational learning [26]. In the definitions of Q' and R' below, the PGP $(\overline{o}, I)$ represents the description of an object tuple $\overline{o}$ by the whole incidence relation I seen from the relative position of $\overline{o}$.

$$\begin{array}{ll} Q' := \{\overline{o} \in O^n \mid Q \subseteq_q (\overline{o}, I)\}, & \text{for } Q \in \mathcal{Q}_n, \text{ for } n \in \mathbb{N} \\ R' := \bigcap_q \{(\overline{o}, I)\}_{\overline{o} \in R}, & \text{for } R \in \mathcal{R}_n, \text{ for } n \in \mathbb{N} \end{array}$$

From there, concepts can be defined in the usual way, and proved to be organized into lattices. A concept is a pair (Q, R) such that $Q' = R$ and $R' =_q Q$. The arity of the projected tuple of Q must be the same as the arity of object tuples in R. It determines the arity of the concept. Unary concepts are about sets of objects, while binary concepts are about relationships between objects, and so on. Note however that the intent of an unary concept can mix attributes of different arities. Unlike RCA, there is a concept lattice for each concept arity rather than for each object type.

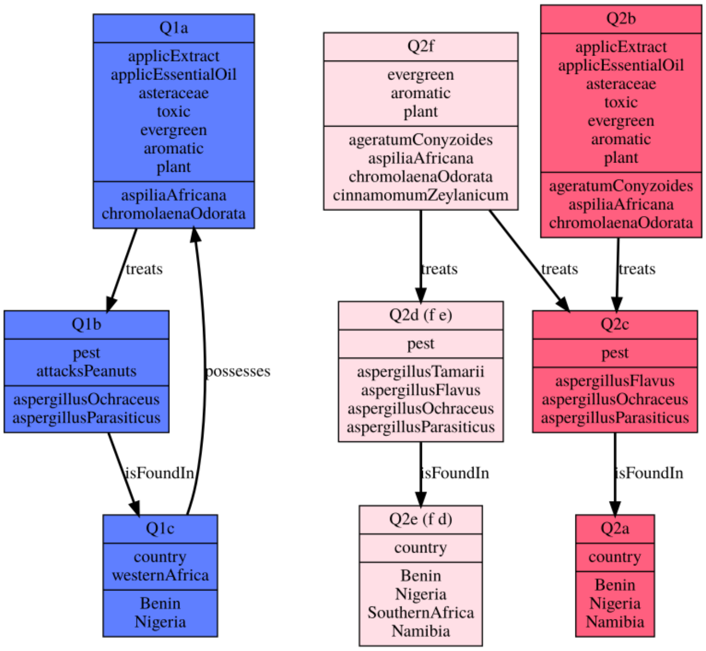


Fig. 2 Compact representation of graph concepts about plants, pests, and countries, with minimum support 2.

Figure 2 displays a compact representation of the graph concepts about plants, pests, and countries as a set of graph patterns. Each rectangle node x identifies a unary concept (e.g., Q1a) along with its extent (here, *aspiliaAfricana*, *chromolaenaOdorata*). The concept intent is the PGP $x \leftarrow P$, where P is the subgraph containing node x and all dark-colored nodes (called the *pattern core*). In the first pattern, all concepts belong to the core, while on the second pattern, only Q2a, Q2b, Q2c belong to the core. By reading the graph, we learn that Concept Q1a is the concept of *"plants that treat pests attacking peanuts, which are found in Western Africa countries that possess the plant, where the plant has a number of features such as being toxic, aromatic, applicable to essential oil, etc."* This concept therefore identifies the valuable situation where a country possesses plants to treat some pests they have. Concepts Q1b and Q1c have the same graph pattern as Q1a but a different projected node, on pests for Q1b and on countries for Q1c. N-ary concepts are obtained by picking several projected nodes. For example, (Q1a,Q1b,Q1c) is a ternary concept whose instances are the object triples $(aspiliaAfricana, aspergillusParasiticus, Benin)$ and $(chromolaenaOdarata, aspergillusOchraceus, Nigeria)$. It represents the cyclic relationship existing between plants, pests, and countries in Western Africa countries. Pattern Q2 shows that every plant (Q2f) treats some pest

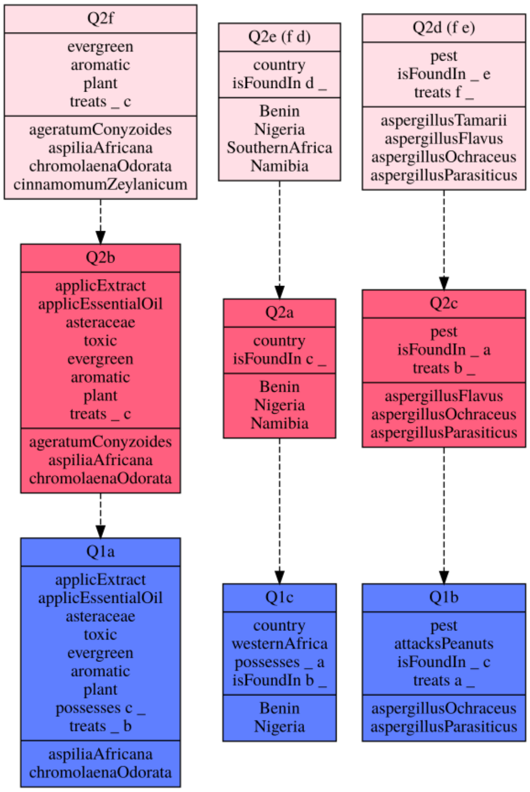


Fig. 3 Generalization ordering of unary graph concepts about plants, pests, and countries, with minimum support 2.

(Q2c), and every pest (Q2d) is found in some country (Q2e) but not all countries possess a plant treating a pest. The concept identifiers between bracket, e.g. Q2d(fe), indicate which nodes of the pattern belong to the concept intent, in addition to the core. For Q2d(fe), it means that *"all four pests are found in some country (Q2e), and are treated by some plant (Q2f) that also treats a pest (Q2c) that can be treated by a more specific kind of plant (Q2b), one that is toxic and can be used for essential oils and extracts."*

2.4 Graph Concept Lattice

The generalization ordering between concepts, hence (part of) the concept lattice, is shown in Figure 3. The relation from concepts to their children is represented by dashed arrows. The top concept that contains all objects is omitted for clarity. In this representation, the hyper-edges with arity at least

2 are shown inside rectangles with unary attributes. For example in Q1a, the label `possesses c _` says that there is a binary edge labeled *possesses* from concept Q1c to this concept Q1a; and label `treats _ b` says that there is a binary edge from Q1a to Q1b (edges are never across patterns). For example, concepts Q1a, Q1b, Q1c are respectively specializations, hence sub-concepts, of concepts Q2b, Q2c, Q2a. The latter are more general concepts respectively for plants, pests, and countries.

3 Conceptual Navigation in Graph-FCA Lattices

The idea of *conceptual navigation* is to see the concept lattice as a navigation structure, with concepts as navigation places, and the lattice structure as a set of navigation links between concepts. In a basic setting, the concept lattice is displayed graphically, and the user can visually navigate it. However, this only works for very small datasets generating at most a few dozen concepts. A more scalable approach consists in displaying only a local view centered on the current concept in the navigation process [13, 6, 1]. Typically, the local view consists of the extent and intent of the current concept, and navigation links to the neighbour concepts. The neighbour concepts are generally the parents and children of the current concept in the Hasse diagram of the concept lattice, but nothing prevents to have links to more distant concepts. Navigation links are generally labeled by the properties that have to be added or removed from the current intent in order to get the intent of the target concept.

The major advantage of local views over concept lattices is that the amount of information to be displayed does not grow exponentially with the size of data, like concept lattices, but rather linearly. This fosters expressivity because higher expressivity leads to many more concepts but not to bigger representations of individual concepts. Conceptual navigation with local views also satisfies usability because the user experience is similar to that of faceted search [29], which is commonly used in e-commerce websites. Indeed, users simply have to make a choice among navigation links at each step, and they get a clear view of their current state at all time.

3.1 Abstract Conceptual Navigation (ACN)

We first introduce and formalize a very generic framework for conceptual navigation based on local views, called *Abstract Conceptual Navigation* (ACN). We call it abstract by analogy with an abstract class in object-oriented programming, where class members are given a specification but not yet an implementation. The genericity concerns two aspects: (1) the nature of con-

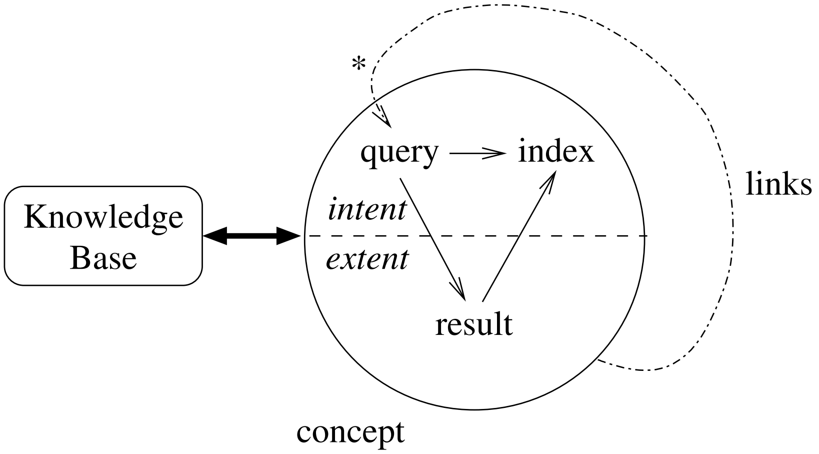


Fig. 4 Schema showing the different components of Abstract Conceptual Navigation (ACN), and their interactions.

cept intents and extents in order to foster expressivity, and (2) the contents of local views in order to foster usability. In the next section, we instantiate ACN to Graph-FCA concept lattices, where intents are PGPs (projected graph patterns), and extents are object relations. An instantiation on classical FCA would have sets of attributes as intents, and sets of objects as extents.

The main novelty of ACN w.r.t. conceptual navigation, as presented above, is the replacement of the concept intent by a *query* and an *index* in the local view (see Figure 4). The motivation is that concept intents are often at the same time overly specific, and not informative enough. This is because the concept intent is defined as the set of *all and only the* properties that are shared by all elements of the concept extent. The *query* is a subset of the intent that contains only intensional elements selected by the user during her navigation. The query is related to the intent in that both characterize the same extent. The query is useful to the user to let her know the current state of the navigation in a concise way. The *index* is typically a superset of the intent by including properties that are shared only by a subset of the extent, along with their frequencies in the extent. This provides the user with the distribution of the extent elements over the set of properties: e.g. *"80% films by Tim Burton have genre Fantastic"*. Those properties can then be used as navigation links to more specific concepts. Finally, the extensional component is replaced by *result* to allow for diverse representations of the extent, e.g. tables, maps or charts.

ACN is formally defined by the following components:

knowledge base (K). K is a *knowledge base* that contains the formal representation of facts, rules, ontological axioms, taxonomies, etc. Its content is constrained by the needs of the application, and the availability of data and domain knowledge. It is an implicit parameter of all ACN operations defined below.

query language (Q) and **initial query** (q_0). Q is a *query language*, i.e. a set of expressible queries. A *query* is a summary of the navigation history, and expresses the user information needs. The current query $q \in Q$ characterizes the current concept, and is part of its intent. Here, query should be understood in a broad sense, and may include closed and open questions, analytical questions (OLAP), sets of keywords, folder paths, updates, commands, ontological assertions, etc. The initial query q_0 is a fixed query, generally very simple, that determines the starting point of the navigation process.

result (R and $\mathit{result} \in Q \to R$). R is the set of all possible *results* (a.k.a., *query results*), and *result* is a function from queries to their result, given a knowledge base K. The query result is any piece of data representing the concept extent, returned to users from the query evaluation. It can be a set of values, an answer list, a table, etc. For some queries, e.g. updates, the knowledge base may be modified as a side effect.

index (S and $\mathit{index} \in Q \times R \to S$). S is the set of all possible indices over the extent, and *index* is a function from queries and their result to their index. An index is at the same time a *summary* over the extent, and a set of query refinements. The intent of the current concept is typically part of the index. Its role is to provide feedback and guidance.

links ($\mathit{links} \in Q \times R \times S \to 2^Q$). Function *links* defines a set of navigation links from the current concept to related concepts. Links can be derived from any component of the current concept: the query, the result, or the index.

The main benefit of ACN is to subsume three paradigms of information access: query languages, navigation structures, and interactive views.

- *ACN as a query language.* Q is the query language, and *result* defines query results for each query. Indices and links are void. For instance, SPARQL editors can be seen as partial ACN instances where the query language is SPARQL, and where results are tables of RDF nodes or RDF graphs.
- *ACN as a navigation structure.* Navigation places are specified by queries in Q, and navigation links are given by the ACN component *links*. Results and indices may be used to compute the links, and may be used to define the contents of navigation places. For instance, file system hierarchies can be seen as partial ACN instances where queries are directory paths, results are file lists, and navigation links lead to children directories and the parent directory.
- *ACN as an interactive view.* Interactive views follow the MVC architecture (model-view-controller). In ACN, the model is the knowledge base K. The view is the composition of a query, its result, and its index (an element of $Q \times R \times S$). The controller is made of the links, which are derived from the

view contents. Each link activation generates a new view by computing a new result (*result*) and a new index (*index*) from the target query. For instance, faceted search can be seen as an ACN instance where a query is a set of facet-values, a result is a selection of items, an index gives the frequency of facet-values among those items, and links allow the addition and removal of facet-values to the query.

ACN inherits *expressivity* from query languages, and *guidance* from navigation structures and interactive views, which achieves our main objective to reconcile *expressivity* and *usability*. However, it must be noted that the actual *expressivity* may be less than the *expressivity* of the query language in the case where navigation links are not rich enough to reach all valid queries. Ideally, links should be both *safe* and *complete*. *Safeness* means that no navigation path leads to dead-ends (e.g., empty results). *Completeness* means that every safe query can be reached through a finite navigation path. *Safeness* is important for the quality of *guidance* as it avoids users to "bump into the walls", and *completeness* is important to leverage the *expressivity* of the query language. Therefore, a critical issue in ACN is the definition of links, and hence the definition of results and indices because links are derived from them.

3.2 Graph-ACN: Instantiating ACN to Knowledge Graphs

We here instantiate ACN to Knowledge Graphs, which we call Graph-ACN, by defining each ACN component in Graph-FCA terms (see Section 2).

Knowledge base.

A knowledge base is formalized as a graph context $K = (O, A, I)$, where objects O are the KG entities, attributes A are the KG k-ary relations, and the incidence relation I is the set of relational facts. As a running example, we consider the graph context displayed in Figure 1.

Query.

A query is an unary PGP $Q = x \leftarrow P$, where P is a graph pattern that uses variable x unless it is empty, and that has a single connected component (*connected pattern*). The motivation for excluding disconnected patterns is that each connected component can be navigated to independently of others.

The projected variable x is called the *focus* of the query. It determines the contents of the index, and the behaviour of navigation links. Some navigation links defined below allow users to move it on other variables in the pattern.

The initial query is the empty query $q_0 := x_1 \leftarrow \emptyset$. In our running example, we consider the query

$$q := x_2 \leftarrow \mathit{plant}(x_1), \mathit{toxic}(x_1), \mathit{treats}(x_1, x_2),$$

which corresponds to concept Q2c in Figure 3. It selects *"everything treated by a toxic plant."*

Result.

A result is a table with variables in headers, and objects in cells. A result is therefore a pair $r = ((x_1, \dots, x_n), R)$ composed of a tuple of n variables, and an n-ary object relation $R \in \mathcal{R}_n$. A result is said *empty* if its object relation R is the empty set.

The function from queries to results is defined as

$$\mathit{result}(Q := x \leftarrow P) := (\mathit{vars}(P), (\mathit{vars}(P) \leftarrow P)'),$$

where $\mathit{vars}(P)$ is the row of variables occuring in P. This definition has the same effect as a `SELECT *` in SPARQL, and provides a richer result than the set of objects Q' that only contains objects at the focus. The result of the initial query q_0 is a one-column table that contains the list of all objects in O. The result of our running query is the following table. Here, there is a bijection between x_1-values and x_2-values but this need not be the case.

$r := \mathit{result}(q) =$

x_1	x_2
ageratumConyzoides	aspergillusFlavus
chromolaenaOdorata	aspergillusOchraceus
aspiliaAfricana	aspergillusParasiticus

Index.

An index is a set that contains two kinds of elements: variables and *attribute positions*. An attribute position is a pair (a, i) where $a \in A$ is an attribute of arity k belonging to the graph context, and $i \in [1, k]$ is an argument position of the attribute. In the above example about plants, attribute *treats* has two positions: position 1 denotes the plants that treat some pest, while position 2 denotes pests that are treated by some plant.

Given a query $q := x \leftarrow P$ and its result $r = \mathit{result}(q)$, the index $s = \mathit{index}(q, r)$ contains all and only the elements that are *relevant* at focus x. Here, "relevant" means that when the element e is inserted into the query q at focus (as defined right below), the new query has a non-empty result. The insertion function $\mathit{insert}(q, e)$ is defined as follows.

- **Inserting a variable** y modifies the query by replacing every occurences of x by y in q. For example, the insertion of x_1 in the running query q, i.e.

$$insert(q, x_1) = x_1 \leftarrow plant(x_1), toxic(x_1), treats(x_1, x_1),$$

generates a new query where occurences of x_2 is replaced by x_1. That query has obviously an empty result as no pest treats a pest.

- **Inserting an attribute position** (a, i) modifies the query by adding atom $a(y_1, \dots, y_k)$, where variable y_i is focus variable x and every other variable is a fresh variable. For example, the insertion of $(\mathit{isFoundIn}, 1)$ in the running query, i.e.

$$\begin{aligned}&insert(q, (\mathit{isFoundIn}, 1)) = \\ &\quad x_2 \leftarrow plant(x_1), toxic(x_1), treats(x_1, x_2), \mathit{isFoundIn}(x_2, x_3),\end{aligned}$$

generates a new query with an additional atom. That query has a non-empty result with an additional column x_3.

x_1	x_2	x_3
ageratumConyzoides	aspergillusFlavus	Namibia
chromolaenaOdorata	aspergillusOchraceus	Nigeria
aspiliaAfricana	aspergillusParasiticus	Benin

The insertion of attribute positions allows to make patterns grow, and the insertion of variables allows to form cycles in patterns. From there, the index can be defined as

$$\begin{aligned} index(q, r) := &\{y \mid y \in vars(P), y \neq x, result(insert(q, y)) \neq \emptyset\} \\ \cup\ &\{(a, i) \mid a \in A, i \in [1, k], result(insert(q, (a, i)) \neq \emptyset\}\end{aligned}$$

A frequency value can be associated to each index element e as the number of objects at focus after inserting it in the query.

$$\mathit{freq}(e) := |insert(q, e)'|$$

Frequencies of attribute positions provide a hint on their distribution over objects at focus. The index of our running query is given in the following table listing index elements along with their frequency.

index element	frequency
(*pest*,1)	3
(*isFoundIn*,1)	3
(*treats*,2)	3
(*attacksPeanut*,1)	2

Links.

There are two kinds of links: focus moves and insertion of index elements. A focus move simply changes the focus variable in the query. The insertion of an index element applies function $insert(q, e)$ defined above on any index element $e \in s$. The ACN function *links* is therefore defined as follows, where

$q = x \leftarrow P$.

$$\begin{aligned} \mathit{links}(q, r, s) = {} & \{y \leftarrow P \mid y \in \mathit{vars}(P), y \neq x\} \\ & \cup \{\mathit{insert}(q, e) \mid e \in s\} \end{aligned}$$

The number of links is therefore bounded by the number of variables in the current query plus the number of attribute positions. In our running examples, there are 5 links: 1 focus move on x_1, leading to concept Q2b; and 4 insertions for the above index elements, 3 staying on Q2c and the last one leading to Q1b.

Safeness.

We define a navigation state as safe in Graph-ACN when it has a non-empty result. Indeed, empty results are like dead-ends in the navigation process, and they are frustrating in a user experience. This is why it should be avoided as much as possible. We can prove safeness by showing that the initial navigation place is safe, and that every navigation link preserves safeness. In our instance, the initial place is obviously safe as the result contains all objects, i.e. all entities of the knowledge graph (otherwise, the KG is empty). Then, there are two kinds of navigation links to consider. Focus moves do not change the query pattern, and by definition of results, they do not change the result either. Therefore, if the source place of the link is safe, then so is the target place. Insertions of index elements also preserve safeness by definition of index elements because their insertion into the current query must lead to queries with non-empty results.

Completeness.

We consider Graph-ACN as complete if every connected graph pattern P that has a non-empty extension can be reached in a finite sequence of navigation links, starting from the initial place. We can prove completeness by building a navigation path from the initial query $q_0 = x \leftarrow \emptyset$ to the target query $q = x \leftarrow P$. This can be done by induction. If the pattern has 0 atoms, then it is the initial query, and no navigation link is required. Now, assuming that all patterns with n atoms are reachable, let us consider a pattern P_{n+1} with $n+1$ atoms. It is always possible to remove a pattern atom $a(x_1, \ldots, x_k)$ from P_{n+1}, and still have a connected pattern P_n with n atoms. Moreover, given that P_{n+1} has a non-empty result, so does P_n because of the Galois connection that exists between PGPs and object relations. By induction hypothesis, there is a navigation path to P_n. From there, P_{n+1} can be reached by moving the focus on a variable x_i that already belongs to P_n (it must exist as P_{n+1} is connected); then by inserting the attribute position (a, i); and finally, for every other position j of the new atom where $x_j \in \mathit{vars}(P_n)$, move the focus on that position, and insert variable x_j.

4 Scaling to Large RDF Graphs with SPARQL Endpoints

We here address the concrete implementation of Graph-ACN to knowledge graphs, using semantic web technologies [18]. The objective is to run the navigation process on top of SPARQL endpoints. SPARQL endpoints are web services that can answer complex SPARQL queries over large RDF graphs. Compared to a previous work [7], our approach has scaled from tens of thousands of triples to billions of triples (e.g., DBpedia). In the following we first describe the correspondence between Graph-FCA and both RDF and SPARQL. Then, we describe how to compute results and indices of navigation places from the queries, taking into account that SPARQL engines often return only partial results on large knowledge graphs.

4.1 From Graph-FCA to RDF and SPARQL

An RDF graph has three kinds of nodes: URIs (entity identifiers), literals (strings and typed values such as numbers and dates), and blank nodes (anonymous nodes). It has one kind of edge, called *triple* (s, p, o), whose components are respectively called *subject* (the source of the edge), *predicate* (the label of the edge), and *object* (the target of the edge). When a URI is used as a predicate, it is called a *property*, and denotes a binary relationship between RDF nodes. When a URI is used as an object with predicate `rdf:type`, it is called a *class*, and denotes a set of RDF nodes. An RDF graph is simply defined as a set of triples.

In order to apply Graph-ACN on RDF graphs, we need to map RDF graphs to graph contexts. This can be done by applying the following rules:

1. each RDF node n becomes a Graph-FCA object o_n;
2. each URI u and literal l also becomes a Graph-FCA unary attribute, in order to allow for the identification of individual entities and values. This is analogous to nominal scaling in FCA;
3. each RDF class c becomes a Graph-FCA unary attribute;
4. each RDF property p becomes a Graph-FCA binary attribute;
5. *triple* is a Graph-FCA ternary attribute, in order to allow for reified triples;
6. for each URI u, fact $u(o_u)$ represents the identity of the object as a URI;
7. for each literal l, fact $l(o_l)$ represents the value of the object as literal;
8. for each triple $(s, \texttt{rdf:type}, c)$, fact $c(o_s)$ represents the membership of the subject to the class;
9. for each triple (s, p, o), facts $p(o_s, o_o)$ and $\mathit{triple}(o_s, o_p, o_o)$ represent respectively the binary relationship between the subjet and the object, and the reified triple.

It can be observed that the arity of attributes is maximum 3. The unary attributes that are derived from URIs and literals are special in that there is single fact that uses them in the graph context. They are *singleton classes*.

Conceptual navigation in Graph-ACN produces PGPs that have to be translated to SPARQL queries in order to use SPARQL endpoints for the computation of results and indices. A SPARQL query has the form `SELECT` ?x_1 `...` ?x_n `WHERE` { GP }, where x_is are variable names, and GP is a SPARQL graph pattern. SPARQL has a rich language of graph patterns but we here only need *basic graph patterns*, which are concatenations of *triple patterns*; and equality filters `FILTER (?`x `=` n`)` between variable x and RDF node n. A triple pattern `s p o.` is like an RDF triple except that variables can be used as components in addition to RDF nodes. The function σ translating PGPs Q to SPARQL queries can be defined as follows:

1. $\sigma(Q) =$ '`SELECT ?`x_1 `...` `?`x_n `WHERE {` $\sigma(P)$ `}`' with $Q = (x_1, \ldots, x_n) \leftarrow P$;
2. $\sigma(P) =$ '$\sigma(atom_1)$ `. ... .` $\sigma(atom_m)$' with $P = \{atom_1, \ldots, atom_m\}$;
3. $\sigma(u(x)) =$ '`FILTER (?`x `= <`u`>)`' where u is a URI;
4. $\sigma(l(x)) =$ '`FILTER (?`x `= "`l`")`' where l is a literal;
5. $\sigma(c(x)) =$ '`?`x `rdf:type <`c`>`' where c is a class;
6. $\sigma(p(x, y)) =$ '`?`x `<`p`> ?`y' where p is a property;
7. $\sigma(triple(x, y, z)) =$ '`?`x `?`y `?`z'

The generated SPARQL query can be simplified by eliminating the equality filters. Each equality filter `FILTER (?`x `=` n`)` can be eliminated by replacing in the query all occurences of `?`x by n.

4.2 Computing the Result, Index, and Links

We here detail the computation of the result, index, and links, assuming that the query q of the current place is the PGP $Q := x \leftarrow P$.

Result.

The result of the current place is directly obtained by evaluating the SPARQL translation of the PGP $vars(P) \leftarrow P$. Indeed, the evaluation of a SPARQL query amounts to find all homomorphisms from the graph pattern to the knowledge graph, and it is therefore equivalent to the Galois connection from PGPs to object relations. The output of SPARQL query evaluation is a table with projected variables as headers, and one row per found homomorphism, binding variables to RDF nodes. This is exactly the expected structure for Graph-ACN results.

Index.

As defined above, the index is composed of variables and attribute positions whose insertion in q produces a new query with a non-empty result. For efficiency reasons, we want to avoid to compute the result, i.e. to evaluate a SPARQL query, for each variable, and for each attribute position. In other words, we want to decide on condition $\mathit{result}(\mathit{insert}(q, e)) \neq \emptyset$ without actually computing the result nor the insertion.

First, we show that the above condition can be decided for variables only by looking at the current result.

Lemma 1. *Let $r := \mathit{result}(q) = ((x_1, \ldots, x_n), R)$, let $x_i = x$ for some $i \in [1, n]$, and let $y = x_j$ for some $j \in [1, n], j \neq i$. The insertion of variable y does not lead to an empty result iff there is a row in the current result s.t. both variables x and y have the same value.*

$$\mathit{result}(\mathit{insert}(q, y)) \neq \emptyset \iff \exists (o_1, \ldots, o_n) \in R : o_i = o_j$$

The frequency of index element y can be computed as follows.

$$\mathit{freq}(y) = |\{o_i \mid (o_1, \ldots, o_n) \in R, o_i = o_j\}|$$

The variables in the index, and their frequencies can therefore be computed in one pass over the result.

Second, we show that the above condition can be decided for attribute positions only by looking at the adjacent edges of the focus objects, i.e. by looking at facts that contain an object that is the value of the focus variable in some row of the current result.

Lemma 2. *Let $r := \mathit{result}(q) = ((x_1, \ldots, x_n), R)$, let $x_i = x$ for some $i \in [1, n]$, and let (a, j) be an attribute position. The insertion of attribute position (a, j) does not lead to an empty result iff there is a row in the current result, and a fact in the graph context s.t. the value of x in the row is the same as the object at position j in the fact.*

$$\begin{aligned}&\mathit{result}(\mathit{insert}(q, (a, j))) \neq \emptyset \iff \\ &\quad \exists (o_1, \ldots, o_n) \in R : \exists a(w_1, \ldots, w_k) \in K : o_i = w_j\end{aligned}$$

The frequency of index element (a, j) can be computed as follows.

$$\mathit{freq}((a, j)) = |\{o_i \mid (o_1, \ldots, o_n) \in R, a(w_1, \ldots, w_k) \in K, o_i = w_j\}|$$

Assuming that the graph context is only available through the SPARQL endpoint, we need to come up with SPARQL queries that will retrieve the attribute positions. In spirit, we would like to evaluate the conjunctive query $(a, j) \leftarrow P \cup \{a(x_1, \ldots, x_{j-1},\ x,\ x_{j+1}, \ldots, x_k)\}$ in order to retrieve all valid attribute positions. Unfortunately, attributes and positions are not valid variables in Graph-FCA queries. However, when considering its translation to

SPARQL for the different kinds of attributes available in RDF, we obtain valid SPARQL queries as we show below.

For attributes that represent RDF nodes (1 position), the additional query atom translates as `FILTER (?x = ?n)`, and the SPARQL query simplifies to `SELECT ?x WHERE {` *GP* `}`. This is actually a projection on the focus variable of the query that computes the result. Those attribute positions can therefore be obtained simply by reading column x of the current result, without any request to the SPARQL endpoint.

For attributes that represent RDF classes (1 position), the SPARQL query is `SELECT ?x ?c WHERE {` *GP* `. ?x rdf:type ?c }`. It retrieves all types of focus objects. The inclusion of `?x` in the select clauses enables to compute frequencies as a post-processing, and also to use the relationships between focus objects and classes in rich user interfaces.

For attributes that represent RDF properties, there are two positions. The SPARQL query for position 1 is `SELECT ?x ?p WHERE {` *GP* `. ?x ?p [] }`, and for position 2 it is `SELECT ?x ?p WHERE {` *GP* `. [] ?p ?x }`, where `[]` denotes an anonymous variable.

Finally, for attribute *triple* that has 3 positions, there are three queries similar to the queries for properties, except that the additional triple patterns are respectively (`?x [] []`), (`[] ?x []`), and (`[] [] ?x`), and only `?x` is in the SELECT-clause.

The above SPARQL queries have the drawback that *GP* has to be evaluated 7 times, including the computation of the result. This is a serious issue for complex patterns. Fortunately, several optimizations are possible. First, all above queries only need the values of x in the pattern. The translated pattern *GP* can therefore be replaced by an enumeration of values for x, as already available in column x of the result: `VALUES ?x {` o_1 `...` o_m `}`, where $\{o_1, \ldots, o_m\}$ is the projection of result r on column x. This kind of SPARQL pattern is very efficiently evaluated. Second, the results for attribute *triple* at positions 1 and 3 can be deduced from the results about properties because the latter are generalizations of the former. For instance, if $(p, 1)$ is a valid attribute position, then so is $(\textit{triple}, 1)$. The related SPARQL queries can therefore be dropped. Third, the remaining queries can be factorized by using a UNION-pattern, as follows.

```
SELECT ?x ?c ?p1 ?p2
WHERE {
  VALUES ?x { o1 ... om }

  { ?x rdf:type ?c }
  UNION
  { ?x ?p1 [] }
  UNION
  { [] ?p2 ?x }
  UNION
  { [] ?x [] }
}
```

Each result of that query will bind at most one variable of `c`, `p1`, `p2` (none of them for attribute position $(\mathit{triple}, 2)$), so that there is no ambiguity in the interpretation of results.

In total, only two requests are done to the SPARQL endpoint per navigation place. The first one to compute the result and part of the index (URIs and literals), and the second one to compute the rest of the index (classes, properties, and *triple*).

Links.

The computation of links is immediate once the index is computed. A natural optimization here is to compute the target queries in a lazy way, delaying the actual computation (focus move or index element insertion) to when the user triggers a link.

4.3 Living with Partial Results

On large knowledge graphs and for some complex queries, the number of results may be impractical for exhaustive computation, transmission from the endpoint, and display to the end-user. In fact, SPARQL endpoints enforce limits in the number of results (e.g., 10,000 for DBpedia). It is therefore necessary to live with this constraint that entails that only partial results may be available. Using such a limit also allows to tune the responsiveness of the system: the lower the limit, the more responsive the system is.

What are the consequences on Graph-ACN navigation? The fact that the result is partial is not really an issue as it does not make sense to display thousands or millions of results to the end-user. This is similar to web search engines that only show the first 10 results by default, and never show thousands of results at once. Now, a partial result entails a partial index because attribute positions in the index are computed from objects found in the focus column of the result. For RDF nodes (URIs and literals), like for results, it does not make sense to display all of them when there are thousands of them. For classes and properties, their number is generally much smaller than for RDF nodes but in some cases it can still be too high for complete display (e.g., classes in YAGO).

The completeness of conceptual navigation is broken by partial indices. However, experience has shown us that the partial results act as a sample from which the most common classes and properties are retrieved, and the missing index elements are generally unfrequently used. The main limitation of partial indices in practice is when one wants to insert in the query a specific RDF node (e.g., selecting a specific film director among all of them), or occasionally a rare class or property. In order to overcome missing index elements and recover completeness, the solution we have adopted is to let the user enter keywords, and to refine the SPARQL queries with filters using those keywords. For instance, if the user enters keywords "Tim Burton" about focus objects, the SPARQL query computing the result will be `SELECT ?`x_1` ... ?`x_n` WHERE` $\{$ GP `FILTER (REGEX(?x, "Tim", i) && REGEX(?x, "Burton", i))` $\}$. It retrieves only results where the focus value contains words "Tim" and "Burton". When the keywords are about classes and properties, it is the SPARQL query retrieving them that is modified accordingly. A further refinement is to match the keywords on the RDFS labels of URIs rather than on the URIs themselves. We omit the details here.

5 Rising in Expressivity

SPARQL has a lot to offer beyond conjunctive queries. It features relational algebra operators (UNION, MINUS, OPTIONAL); scalar expressions that can be used in filters, variable bindings (BIND), and in the SELECT-clause; aggregations with GROUP BY; and yet other features. It is desirable to extend our conceptual navigation framework to cover such expressive features. The difficulty is that operators like UNION and MINUS do not apply on the whole query, nor on the focus variable but on a subset of query atoms. We therefore need a representation of queries and their focus that is more adequate to the insertion of those SPARQL features.

5.1 An Algebraic Form of Queries

We introduce an algebraic form of queries. A *pattern tree* is a pair $T = \langle x, D \rangle$, where x is a variable, and D is a *description* of x possibly containing a number of subtrees. A description D is one of the following forms, where $\bullet$ denotes the described node x:

1. $\top$: the void description;
2. $a(T_1, \ldots, T_{i-1}, \bullet, T_{i+1}, \ldots, T_k)$ with $T_j = \langle x_j, D_j \rangle$ for $j \in [1, k], j \neq i$: a description stating that the described variable is the i-th argument of the hyper-edge $a(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_k)$, and that each x_j is recursively described by D_j;
3. $\bullet = y$: a description stating that the described variable is equal to y;
4. $D_1 \wedge D_2$: a conjunction of two descriptions, stating that x satisfies both descriptions.

An *algebraic query* is a pattern tree with the focus localized on a part of the tree, either a subtree or a description. Every subtree has a distinct variable, and cycles are expressed with descriptions like $\bullet = y$.

For example, the algebraic query (with focus underlined)

$$\begin{aligned}&\langle x,\, \mathit{plant}(\bullet)\\ &\qquad \wedge\ \mathit{treats}(\bullet, \langle y, \mathit{isFoundIn}(\bullet, \langle z, \mathit{country}(\bullet)\rangle)\rangle)\\ &\qquad \wedge\ \mathit{possesses}(\underline{\langle w, \bullet = z\rangle}, \bullet)\rangle\end{aligned}$$

is equivalent to the PGP query

$$z \leftarrow \mathit{plant}(x), \mathit{treats}(x,y), \mathit{isFoundIn}(y,z), \mathit{country}(z), \mathit{possesses}(z,x).$$

The algebraic form can be easily obtained from the PGP form $x \leftarrow P$ by using a tree traversal of P, and by putting the focus on the subtree with variable x. For binary attributes, description $a(\bullet, T_2)$ corresponds to cross the binary relation forward, while $a(T_1, \bullet)$ corresponds to cross it backwards.

The computation of the result of algebraic queries requires to translate them to SPARQL, which is defined as follows:

1. $\sigma(T) = \sigma_x(D)$, with $T = \langle x, D\rangle$ (trees)
2. $\sigma_x(\top) = \texttt{''}$ (void description)
3. $\sigma_x(u(\bullet)) = \texttt{'FILTER (?}x\texttt{ = <}u\texttt{>)'}$ (URIs)
4. $\sigma_x(l(\bullet)) = \texttt{'FILTER (?}x\texttt{ = "}l\texttt{")'}$ (literals)
5. $\sigma_x(c(\bullet)) = \texttt{'?}x\texttt{ rdf:type <}c\texttt{>'}$ (classes)
6. $\sigma_x(p(\bullet, \langle y, D_y\rangle)) = \texttt{'?}x\texttt{ <}p\texttt{> ?}y\texttt{ . }\sigma_y(D_y)\texttt{'}$ (properties)
7. $\sigma_x(p(\langle y, D_y\rangle, \bullet)) = \texttt{'?}y\texttt{ <}p\texttt{> ?}x\texttt{ . }\sigma_y(D_y)\texttt{'}$ (properties)
8. $\sigma_x(\mathit{triple}(\bullet, \langle y, D_y\rangle, \langle z, D_z\rangle)) = \texttt{'?}x\texttt{ ?}y\texttt{ ?}z\texttt{ . }\sigma_y(D_y)\texttt{ . }\sigma_z(D_z)\texttt{'}$ (triples)
9. $\sigma_x(\mathit{triple}(\langle y, D_y\rangle, \bullet, \langle z, D_z\rangle)) = \texttt{'?}y\texttt{ ?}x\texttt{ ?}z\texttt{ . }\sigma_y(D_y)\texttt{ . }\sigma_z(D_z)\texttt{'}$ (triples)
10. $\sigma_x(\mathit{triple}(\langle y, D_y\rangle, \langle z, D_z\rangle, \bullet)) = \texttt{'?}y\texttt{ ?}z\texttt{ ?}x\texttt{ . }\sigma_y(D_y)\texttt{ . }\sigma_z(D_z)\texttt{'}$ (triples)
11. $\sigma_x(D_1 \wedge D_2) = \texttt{'}\sigma_x(D_1)\texttt{ . }\sigma_x(D_2)\texttt{'}$ (conjunction)

Finally, it remains to redefine the insertion of variables and attribute positions at the current focus. The insertion of a variable y introduces a new conjunct at the focus. If the focus is on a description D, it is replaced by $D\ \wedge\ \bullet = y$. If the focus is on a subtree $T = \langle x, D\rangle$, then the insertion applies on D. Similarly, the insertion of an attribute position (a, i) introduces at the focus the new conjunct $a(T_1, \ldots, T_k)$ where $T_i = \bullet$ and other T_j are fresh variables with void descriptions.

The index and navigation links are then computed in exactly the same way because they rely on the result, which is left unmodified.

5.2 Extensions of the Query Algebra

Each extension of the query algebra, aimed at covering some SPARQL feature, consists in introducing new algebraic query constructs. For each new construct, one needs to extend the SPARQL translation, and to define new links to introduce the new construct in a query. The index is generally left unmodified because most additional constructs do not contribute to the description of data. As a consequence, those extensions have almost no impact on usability and scalability. From the user point of view, there are only a few additional navigation links, which can be ignored if she is not interested in the feature. From the point of view of scalability, the query computing the index is the same, and the only impact is on the computation of results depending on which features are used. For example, aggregations are typically costly to evaluate.

We illustrate such an extension with SPARQL UNION, which expresses a disjunction between two patterns. First, we introduce a new description $D_1 \vee D_2$, a disjunction of descriptions. Second, we define its translation to SPARQL, using the UNION operator:

$$\sigma_x(D_1 \vee D_2) = \texttt{'\{ } \sigma_x(D_1) \texttt{ \} UNION \{ } \sigma_x(D_2) \texttt{ \}'}.$$

Third, we add a link from a query whose focus is on a description D to the same query where that description is replaced by $D \vee \top$, introducing the disjunction with an alternative description that is initially void. The focus can be moved on the void description, ready to be refined by insertion of an attribute position for instance.

The query algebra can be extended in a similar way with SPARQL MINUS, which expresses negation on a pattern. The new description is $D_1 \wedge \neg D_2$, whose translation to SPARQL is $\texttt{'}\sigma_x(D_1) \texttt{ MINUS \{ } \sigma_x(D_2) \texttt{ \}'}$. The additional link replaces description D at focus by $D \wedge \neg\top$.

Other description extensions express numeric inequalities (e.g., ≥ 10) or string matching (e.g. *contains* "foo", *matches* "[0-9]+"). Query trees can be extended with binding of variables by complex expressions (e.g., computing people's age as the difference between current date and their birthdate), filtering of results by complex expressions (e.g., filtering people whose age is at least 18), and by aggregations over results (e.g., computing the average age of people per country). Expressions constitute a third kind of part in queries, in addition to trees and descriptions, and have thus their own SPARQL translations and navigation links (see [10] for technical details).

An advantage of the algebraic form of queries, beyond rising in expressivity, is that it is more amenable to verbalization in natural language. Indeed, every tree can be verbalized as a noun phrase, while every description can be verbalized as a relative clause. For instance, the example query above can be verbalized as *"a plant that treats something found in a country, and that the country possesses."* This idea has been formalized with the `N<A>F` design pattern [9].

6 The Sparklis Tool and Application Cases

The Graph-ACN framework presented above is implemented in full in the Sparklis tool[6], though historically Graph-FCA was formalized later than the first implementation. After presenting the implementation, user interface and capabilities of Sparklis, we discuss a few application cases on large knowledge graphs.

6.1 Sparklis

Sparklis implements Graph-ACN as a web application that runs on top of SPARQL endpoints. It runs in the web browser as JavaScript (compiled from OCaml with tool `js_of_ocaml`[7]), and computes extensions and indices by sending SPARQL requests to the SPARQL endpoint.

The user interface of Sparklis is made of a view of the current navigation place, plus configuration widgets. Figure 5 shows a screenshot of Sparklis on a core subset of DBpedia. The main configuration widget is a text input at the top where the URL of the SPARQL endpoint has to be entered. Other configuration widgets are available through the “Configure” menu but defaults are generally fine.

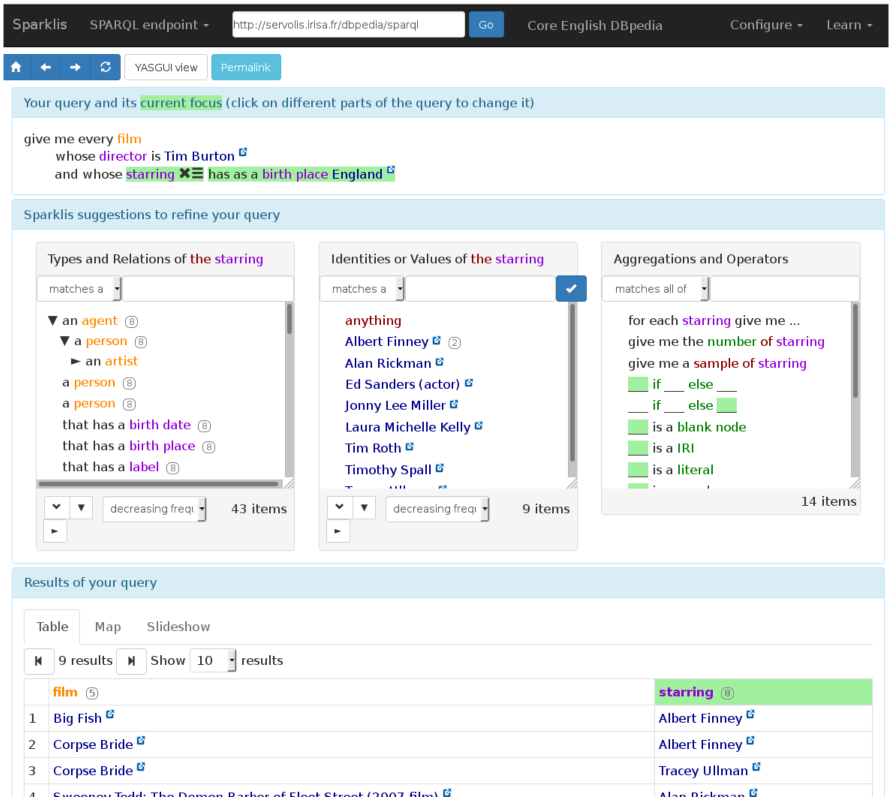
The current place view is made of three parts, in accordance with the ACN framework: the query (top), the index (middle), and the result (bottom). The query is fully verbalized in (controlled) natural language, using RDF classes and properties as nouns, and entities as proper nouns. The focus part is highlighted in light green, and can be moved simply by clicking on different parts of the query. At the end of the focus part, the cross allows to delete the focus part, and the hamburger-icon provides a few navigation links to insert operators like conjunction, disjunction, negation, or sorting of results. In the current version, Sparklis covers all SELECT-queries with the exclusion of queries that need disconnected graph patterns.

The index is splitted into three lists: attribute positions for RDF classes, properties, and *triple* (left), attribute positions for RDF nodes (center), and other navigation links for introducing various operators (right). Each index element can be inserted at the query focus simply by clicking it. By default, index elements are sorted by decreasing frequency but they can also be sorted in lexicographic order. Filter inputs at the top of each list allows to force the retrieval of index elements matching some constraint (matching some keywords or satisfying numerical inequalities).

The result is displayed as a table where headers are verbalizations of variables, and the focus column is highlighted in light green. Headers can be

[6] Available online at http://www.irisa.fr/LIS/ferre/sparklis/

[7] https://ocsigen.org/js_of_ocaml/3.1.0/manual/overview



Your query and its current focus (click on different parts of the query to change it)

give me every film
whose director is Tim Burton
and whose starring ✖☰ has as a birth place England

Sparklis suggestions to refine your query

Types and Relations of the starring

matches a

▼ an agent (8)
▼ a person (8)
► an artist
a person (8)
a person (8)
that has a birth date (8)
that has a birth place (8)
that has a label (8)

decreasing frequ 43 items

Identities or Values of the starring

matches a ✓

anything
Albert Finney (2)
Alan Rickman
Ed Sanders (actor)
Jonny Lee Miller
Laura Michelle Kelly
Tim Roth
Timothy Spall

decreasing frequ 9 items

Aggregations and Operators

matches all of

for each starring give me ...
give me the number of starring
give me a sample of starring
🟩 if ___ else ___
___ if ___ else 🟩
🟩 is a blank node
🟩 is a IRI
🟩 is a literal

14 items

Results of your query

Table Map Slideshow

9 results Show 10 results

	film (5)	starring (8)
1	Big Fish	Albert Finney
2	Corpse Bride	Albert Finney
3	Corpse Bride	Tracey Ullman
4	Sweeney Todd: The Demon Barber of Fleet Street (2007 film)	Alan Rickman

Fig. 5 Screenshot of Sparklis where the current query is “the films directed by Tim Burton and starring somebody born in England”, with focus on the actors.

clicked to move the focus, and the RDF nodes in the cells can be inserted in the query by clicking them. Depending on the result contents, Sparklis can offer additional views on the concept extent: a slideshow of all images occuring in the result, or a map of all geolocalized objects in the result.

The “Learn” menu provides a large number of example queries, and for some of them, a screencast showing how to build them step by step. The user interface and verbalization of queries are available in several languages: English, French, Spanish and Dutch.

6.2 Application Cases

The first version of Sparklis was put online in Spring 2014. Since then, it has been regularly used by thousands of unique users over hundreds of different knowledge graphs (SPARQL endpoints) covering many domains (encyclopaedias, bioinformatics, medicine, bibliography, administrations, etc.). At the time of writing, Sparklis earned consistently around 1000 hits per month over the last years. Those figures demonstrate that our approach is successful in supporting exploration and querying of real knowledge graphs.

In the following, we shortly present four application cases on large knowledge graphs. In each case, we describe the knowledge graph, i.e. its contents, its size, its peculiarities, and we list a few questions that can be built and answered in Sparklis. For some of the questions, we provide a short link to its navigation place in Sparklis.

Encyclopedic Graphs: DBpedia and Wikidata.

The most commonly used knowledge graphs are those that have encyclopedic contents. For example, DBpedia [23] is a KG version of Wikipedia, where triples are mostly extracted from the infoboxes included in Wikipedia pages. This implies that DBpedia covers all kinds of topics (e.g., people, organizations, creative works, species, places), and for this reason, it stands in the middle of Linked Open Data (LOD). It is one of the biggest open knowledge graph with more than 3 billions of triples. We list a few questions, taken from QALD-4 challenge, along with the Sparklis link.

1. *Which rivers flow into a German lake?* (`http://bit.ly/2NpuIlY`)
2. *Give me all films produced by Steven Spielberg with a budget of at least $80 million.* (`http://bit.ly/2WSlmlX`)
3. *How many languages are spoken in Colombia?* (`http://bit.ly/2WQyfNo`)
4. *Which poet wrote the most books?* (`http://bit.ly/32mWRP6`)

Another important encyclopedic KG is Wikidata[8]. Unlike DBpedia, it can be directly edited by people (like a wiki) in a structured way. Editors can create new entities (called *items*), and add *statements* about entities. Some of the contents is automatically imported from existing structured sources but a lot of its contents is also added manually. Wikidata does not follow the RDF model, although an RDF version exists. In particular, it is possible to make statements about statements. For instance, it is possible to say that Barack Obama is a president of the Unites States, and then say that this presidency started in 2008, and ended in 2016. This specificity and others required several adaptations in Sparklis. Here are a few questions that can be answered in Wikidata.

1. *Show me a map of French hospitals.* (`http://bit.ly/2CoTeNP`)
2. *What are the popular given names among the French people?* (`http://bit.ly/33swvfI`)
3. *Give me a slideshow of French prime ministers (since 1815) by decreasing total length of service.* (`http://bit.ly/2NtNNUp`)

[8] `https://www.wikidata.org/`

Bibliographic Data at Persée.

Persée[9] is a French organization that provides free access to more than 600,000 scientific publications, notably in the domain of humanities and social sciences. It maintains a SPARQL endpoint that gives access to their metadata[10], and they have officially adopted Sparklis as an exploration and querying tool in 2017. Their motivation was to empower their researchers by enabling them to build complex questions in a free way. The dataset contains about 42 millions of triples. Here are a few questions that they give as example on their web site (translated from French).

1. *Co-authors of Pierre Bourdieu, ordered by lastname and firstname*
2. *Authors with the largest number of co-authors*
3. *For each author, and for each co-author, how many articles in common*
4. *For a given author, the journal titles where he/she published, along with the number of articles and the publication dates*

Pharmacovigilance Data in PEGASE.

In the context of the PEGASE[11] research project on pharmacovigilance, we have designed a large knowledge graph about (anonymous) patients taking drugs and having adverse drug reactions (ADR) [3]. It includes patient data from FAERS (about 25 millions of triples for three months of records), and terminological data from SNOMED CT and MedDRA (about 3 millions of triples). The first specificity of this dataset is that there are statements about statements like in Wikidata: e.g., there are statements about how patients have taken drugs (how many, how often, etc.). The second specificity is that MedDRA and SNOMED CT are two large hierarchies that required extensions of Sparklis for querying and visualizing them. There is a maintained SPARQL endpoint but it is not publicly available because of license restrictions on MedDRA and SNOMED CT. Here are a few examples of questions that can be answered on the PEGASE knowledge graphs.

1. *Are there patients who experienced renal failure after taking Cloxacillin? patients being more than 60 years old?*
2. *Which MedDRA terms describe an inflammatory ADR in the brain?*

Government Data.

The QALD-6 challenge (Question Answering over Linked Data) introduced a new task (Task 3) on "Statistical question answering over RDF datacubes" [32]. The dataset contains about 4 million transactions on government spendings all over the world, organized into 50 datacubes, all modeled

[9] http://www.persee.fr/

[10] http://data.persee.fr/

[11] https://anr.fr/Project-ANR-16-CE23-0011

in RDF. The specificity of the dataset is that it contains a lot of numeric data, and that most questions on it consists in computing aggregations such as averages and totals. The KG contains about 16 millions of triples in total. Unfortunately, the SPARQL endpoint is not maintained so that it is no more possible to explore it in Sparklis. Sparklis could build 148 out of the 150 questions of the challenge. We list here a few questions as examples.

1. *How much was spent on public safety by the Town of Cary in 2010?*
2. *Which expenses had the highest total amount of proposed expenditures for the Maldives?*
3. *How many suppliers did the Newcastle city council use for education?*

7 Conclusion and Perspectives

We have introduced *Abstract Conceptual Navigation (ACN)*, a general paradigm based on concept lattices to reconcile expressivity, usability, and scalability in the exploration and querying of a knowledge base. Users are guided in a safe and complete way through the concept lattice, and the user navigation state is represented at all time as a readable query, which is updated according to the navigation links chosen by users. Only a local view centered on the current concept is shown, but ACN gives room to enriched representations of the extent (called *result*) and intent (called *index*) that can provide hints about the concept neighborhood.

We have instantiated ACN to knowledge graphs (Graph-ACN) with Graph-FCA as a formal ground. We have shown how to implement it on top of SPARQL endpoints in a scalable way. In Graph-ACN, queries are conjunctive queries to start with, and are then extended to advanced features available in SPARQL. Results are tables, like for SPARQL queries, thus going beyond the usual sets of objects in other FCA approaches. We have shown an effective implementation of Graph-ACN through the Sparklis tool, and effective applications on real and large knowledge graphs.

In terms of scalability, a perspective is to improve efficiency by designing specialized data structures and algorithms in the SPARQL endpoint to optimize the Graph-ACN-specific SPARQL queries. In terms of expressivity, beyond the `SELECT`-queries of SPARQL, there are also `CONSTRUCT`-queries that return graphs, and updates. Beyond SPARQL, we can imagine ACN instances for other query languages such as SQL, XQuery or Cypher. Another perspective for increasing expressivity is to enrich the index in order to support knowledge extraction tasks, e.g. with association rules or functional dependencies.

References

1. Alam, M., Buzmakov, A., Napoli, A.: Exploratory knowledge discovery over web of data. Discrete Applied Mathematics **249**, 2–17 (2018). DOI 10.1016/j.dam.2018.03.041
2. 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)
3. Bobed, C., Douze, L., Ferré, S., Marcilly, R.: Sparklis over PEGASE knowledge graph: a new tool for pharmacovigilance. In: A. Waagmeester, et al. (eds.) Int. Conf. Semantic Web Applications and Tools for Life Sciences (SWAT4LS), *CEUR Workshop Proceedings*, vol. 2275 (2018)
4. Carpineto, C., Romano, G.: A lattice conceptual clustering system and its application to browsing retrieval. Machine Learning **24**(2), 95–122 (1996)
5. Chekol, M.W., Euzenat, J., Genevès, P., Layaïda, N.: SPARQL query containment under SHI axioms. In: AAAI Conf. Artificial Intelligence (2012)
6. Ducrou, J., Eklund, P.: An intelligent user interface for browsing and search MPEG-7 images using concept lattices. Int. J. Foundations of Computer Science, World Scientific **19**(2), 359–381 (2008)
7. Ferré, S.: Conceptual navigation in RDF graphs with SPARQL-like queries. In: L. Kwuida, B. Sertkaya (eds.) Int. Conf. Formal Concept Analysis, LNCS 5986, pp. 193–208. Springer (2010)
8. Ferré, S.: A proposal for extending formal concept analysis to knowledge graphs. In: J. Baixeries, C. Sacarea, M. Ojeda-Aciego (eds.) Int. Conf. Formal Concept Analysis (ICFCA), LNCS 9113, pp. 271–286. Springer (2015)
9. Ferré, S.: Bridging the gap between formal languages and natural languages with zippers. In: H. Sack, et al. (eds.) Extended Semantic Web Conf. (ESWC), pp. 269–284. Springer (2016)
10. Ferré, S.: A SPARQL 1.1 query builder for the data analytics of vanilla RDF graphs. Research report, IRISA, team SemLIS (2018). URL https://hal.inria.fr/hal-01820469
11. Ferré, S., Cellier, P.: Graph-FCA: An extension of formal concept analysis to knowledge graphs. Discrete Applied Mathematics **273**, 81–102 (2019). DOI https://doi.org/10.1016/j.dam.2019.03.003. URL http://www.sciencedirect.com/science/article/pii/S0166218X19301532
12. Ferré, S., Ridoux, O.: A file system based on concept analysis. In: Y. Sagiv (ed.) Int. Conf. Rules and Objects in Databases, LNCS 1861, pp. 1033–1047. Springer (2000)
13. Ferré, S., Ridoux, O.: An introduction to logical information systems. Information Processing & Management **40**(3), 383–419 (2004)
14. Ganter, B., Wille, R.: Formal Concept Analysis — Mathematical Foundations. Springer (1999)
15. Godin, R., Missaoui, R., April, A.: Experimental comparison of navigation in a Galois lattice with conventional information retrieval methods. International Journal of Man-Machine Studies **38**(5), 747–767 (1993)
16. Hahn, G., Tardif, C.: Graph homomorphisms: structure and symmetry. In: Graph symmetry, pp. 107–166. Springer (1997)
17. Hildebrand, M., van Ossenbruggen, J., Hardman, L.: /facet: A browser for heterogeneous semantic web repositories. In: I.C. *et al* (ed.) Int. Semantic Web Conf., LNCS 4273, pp. 272–285. Springer (2006)
18. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2009)
19. Höffner, K., Walter, S., Marx, E., Lehmann, J., Ngomo, A.C.N., Usbeck, R.: Overcoming challenges of semantic question answering in the semantic web. Semantic Web Journal (2016)

20. 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)
21. Kötters, J.: Concept lattices of a relational structure. In: H. Pfeiffer, and others (eds.) Int. Conf. Conceptual Structures for STEM Research and Education, LNAI 7735, pp. 301–310. Springer (2013)
22. Kuznetsov, S.O., Samokhin, M.V.: Learning closed sets of labeled graphs for chemical applications. In: S. Kramer, B. Pfahringer (eds.) Int. Conf. Inductive Logic Programming, LNCS 3625, pp. 190–208. Springer (2005)
23. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web Journal (2013). Under review.
24. Liquiere, M., Sallantin, J.: Structural machine learning with galois lattice and graphs. In: Int. Conf. Machine Learning, pp. 305–313 (1998)
25. Mika, P.: On schema.org and why it matters for the web. IEEE Internet Computing **19**(4), 52–55 (2015)
26. Muggleton, S., Raedt, L.D.: Inductive logic programming: Theory and methods. Journal of Logic Programming **19,20**, 629–679 (1994)
27. Plotkin, G.: Automatic methods of inductive inference. Ph.D. thesis, Edinburgh University (1971)
28. Rouane-Hacene, M., Huchard, M., Napoli, A., Valtchev, P.: Relational concept analysis: mining concept lattices from multi-relational data. Annals of Mathematics and Artificial Intelligence **67**(1), 81–108 (2013)
29. Sacco, G.M., Tzitzikas, Y. (eds.): Dynamic taxonomies and faceted search. The information retrieval series. Springer (2009)
30. Sowa, J.: Conceptual structures. Information processing in man and machine. Addison-Wesley, Reading, US (1984)
31. SPARQL 1.1 query language (2012). URL http://www.w3.org/TR/sparql11-query/. W3C Recommendation
32. Unger, C., Ngomo, A.C.N., Cabrio, E.: 6th open challenge on question answering over linked data (QALD-6). In: H. Sack, et al. (eds.) Semantic Web Evaluation Challenge, pp. 171–177. Springer (2016)