A Wizard for Composing SPARQL Queries in the GF Framework for Ontology-Based Data Access

Sergio Alejandro Gómez[1,2] and Pablo Rubén Fillottrani[1,2]

[1]Laboratorio de I+D en Ingeniería de Software y Sistemas de Información (LISSI)
Departmento de Ciencias e Ingenería en Computación
Universidad Nacional del Sur
San Andrés 800, (8000) Bahía Blanca, Argentina
Email: {sag,prf}@cs.uns.edu.ar
[2]Comisión de Investigaciones Científicas de la Provincia de Buenos Aires

Abstract. Ontology-Based Data Access is a methodology concerned with bridging the gap between legacy data sources and semantic web technologies by providing protocols and tools for translating old data into ontologies. Querying modern ontologies represented as networks of objects interlinked by relations and properties and stored in OWL/RDF text files requires writing SPARQL queries, an activity requiring technical proficiency that is not usually in the hands of lay users. We extend our prototype of OBDA called GF to include the functionality of executing arbitrary SPARQL queries posed against OWL/RDF ontologies obtained by OBDA from H2 relational databases as well as Excel and CSV spreadsheets. To help naive users with less technical programming skills perform queries on such ontologies, we introduce a wizard for visually expressing a subset of SPARQL queries in a Query-By-Example approach.

Keywords. Ontologies, Ontology-Based Data Access, SPARQL, Knowledge Representation

1 Introduction

The Semantic Web (SW) is a version of the web where data resources have a precise meaning given in terms of conceptualizations known as ontologies that allow software agents to reason about such meaning automatically [1]. Ontology-Based Data Access (OBDA) is a discipline concerned with bridging the gap between legacy data sources and SW technologies by providing protocols and tools for translating old data into ontologies [2]. Querying ontologies provides many benefits for querying relational data as it allows the usage of open-world semantics in contrast to closed-world semantics and also allows to make explicit implicit conclusions hidden in the non-trivial subclass and composition relations that describe the underlying application domain modeled by the queried ontologies.

One of the advantages of OBDA is that old, legacy data can be then combined with new ontological data. Legacy data include tabular data as relational

database, Excel spreadsheets and CSV text files. Modern ontological data in contrast is represented as networks of objects interlinked by relations and properties and stored as OWL/RDF text files distributed in the SW. Querying modern ontologies requires writing SPARQL queries [3], an activity that requires technical proficiency that quite normally is not in the hands of lay users.

In this work, we extend a prototype of OBDA called GF [4] that we have been developing in the last years to include the functionality of executing arbitrary SPARQL queries posed against OWL/RDF ontologies obtained by OBDA from H2 relational databases as well as Excel and CSV spreadsheets. Also to help naive users with less technical programming skills to perform queries on such ontologies, we introduce a wizard for expressing a subset of SPARQL queries visually based on a Query-By-Example (QBE) approach [5]. Our solution provides a concrete way of writing SPARQL queries over legacy data without requiring the user to know explicitly SPARQL syntax. For reproducibility of the reported results, an executable file along with the files of the examples presented in this paper and its results can be checked online at `http://cs.uns.edu.ar/~sag/gf-v4.3`.

The rest of the work is structured as follows. In Sect. 2, we review the subset of SPARQL queries that our wizard can generate. In Sect. 3, we present the wizard to build the queries discussed previously. In Sect. 4, we review related work. Finally, in Sect. 5, we conclude and foresee future work.

2 Queries in SPARQL

SPARQL is the standard query language and protocol for Linked Open Data and RDF databases that can efficiently extract information hidden in non-uniform data and stored in various formats and sources, such as the web or RDF triple-stores. The distributed nature of SW data, unlike relational databases, helps users to write queries based on what they want to know instead of how the data is organized. In contrast to the SQL query language for relational databases, SPARQL queries are not constrained to working within one database – federated queries can access multiple data stores (or endpoints) because SPARQL is also an HTTP-based transport protocol, where any endpoint can be accessed via a standardized transport layer. RDF results can be returned in several data-interchange formats and RDF entities, classes, and properties are identified by IRIs such as `<http://example.org/Person/name>`, which are difficult to remember even knowing SPARQL and the underlying structure of the data source.

As mentioned in the introduction, we propose a wizard for visually composing SPARQL queries posed against a data source expressed as an OWL/RDF ontology. We now present the subset of queries that we solve with our implementation. We present a running example with which we present some prototypical queries and in Sect. 3 we show how these queries can be solved by using the wizard that we defined. We present a relational database schema for which the GF system produces an ontology automatically. Then we show some SPARQL queries posed against the ontology. We will see that writing those queries from scratch present an important challenge even for experienced users and that the proposed wizard

can help in easing such task by allowing the composition of queries by a Query-By-Example methodology (i.e., visually and abstracting from some of the inner details of the query structure and the queried dataset).

Example 1. In Fig. 1, we define the schema of a very simple relational database and show how its translation to an OWL Description Logic (DL)[1] ontology should be and then propose some iconic SPARQL queries. There are two tables: *Person* and *Phone*. A person has a unique identifier, a name, a weight in kilograms, a sex that is false if the person is female and true if the person is male, also a person has a birth date. A phone has a unique identifier, a number, a price, and its owner. There is an implicit one-to-many relation from Person to Phone, meaning that a person can have 0, 1, or more phones and a phone can belong to 0 or at most 1 person.

Notice that in this work, we have added extra functionality to the direct mapping specification programmed in previous versions of GF (see [4] and references therein for details) in order to simulate the natural joins between tables and be able to retrieve that characteristic from SPARQL. Thus, the person now knows his phones and vice versa.

Person (*personID*, *name*, *weight*, *sex*, *birthDate*)
Phone (*phoneID*, *number*, *price*, *owner*)

Person

personID	*name*	*weight*	*sex*	*birthDate*
1	John	120.0	true	2001-01-01
2	Paul	110.0	true	2002-01-01
3	Mary	60.0	false	2001-01-01

Phone

phoneID	*number*	*price*	*owner*
1	555-1234	200.00	1
2	555-1235	220.00	1
3	555-1236	230.00	2

Fig. 1. Relational schema and instance for tables *Person* and *Phone*

Example 2 (Continues Ex. 1). In Fig. 2, the UML design of the classes Person and Phone can be seen. In Fig. 3, the instances of classes Person and Phone are shown. There are three people, two males named John and Paul, and one female of name Mary. John has two phones (viz., 1 and 2), Paul has only one (viz., 3) but Mary has none. The class HeavyYoungMan is defined as a subclass of Person according to the SQL filter: `select "personID" from "Person" where "sex"=true and "birthDate">='2001-01-01' and "weight">=100.0`.

We now explore several paradigmatic query cases in SPARQL. The choice of the particular syntax of some queries is due to that they are presented in the exact way that they are composed by our tool employing the visual specification that we present in Sect. 3. We solve a very specific subset of queries and categorize its cases as: queries over a single class, queries over a simple hierarchy of classes, and queries over an association.

[1] In this context, we see a DL ontology as a mathematical conceptualization of an equivalent OWL/RDF file, which is understood as the serialization of such ontology. We refer the reader to [6].

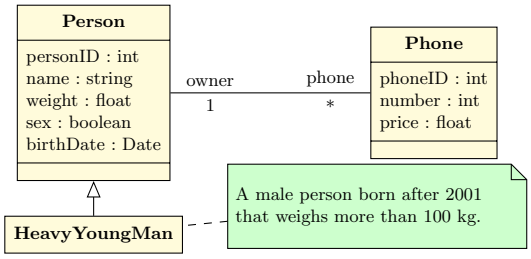


Fig. 2. UML class diagram for people and their phones obtained via OBDA from Fig. 1

```
Person(p1).                personID(p1,1)          name(p1, john).        weight(p1,120.0).      sex(p1,true).
birthDate(p1,2001-01-01).  phone(p1,t1)            phone(p1,t2).          HeavyYoungMan(p1).
Person(p2).                personID(p2,2).         name(p2, paul).        weight(p2,110.0).      sex(p2,true).
birthDate(p2,2002-01-01).  phone(p2,t3).           HeavyYoungMan(p2).
Person(p3).                personID(p3,2).         name(p3, mary).        weight(p3,60.0).       sex(p3,false).
birthDate(p3,2001-01-01).
Phone(t1).                 number(t1, 555-1234).   price(t1, 200.0).      owner(t1, p1).
Phone(t2).                 number(t2, 555-1235).   price(t2, 220.0).      owner(t1, p1).
Phone(t3).                 number(t3, 555-1236).   price(t3, 230.0).      owner(t1, p2).
```

Fig. 3. Assertional knowledge about people and their phones for UML diagram in Fig. 2 obtained from the relational instance in Fig. 1

Example 3 (Continues Ex. 2). We start with a *selection query having several conditions over a single class*: Select the portion of the data that comprise all the females that were born in 2001 that weigh less than 70 kilos, and her name optionally starts with an *M*, contains an *r*, and ends with a *y*. When relevant in all queries we ask the query processor to show at most 10 results starting with the first result. The text of the SPARQL query can be seen in Listing 1.1. The result of the query is computed in tabular form:

id	*name*	*isMale*	*bd*	*weight*
3	Mary	false	2001-01-01	60.0

Example 4 (Continues Ex. 2). We now show a *totalization query*: select the average weight of the men. The source code for the query is shown in Listing 1.2. The result of the query is:

averageWeight
115.0

Example 5 (Continues Ex. 2). We now show a query that works by *grouping similar data according to the value of a field*: Categorize people by sex and compute the average and maximum weight, least birthdate, person count, and sum of weights. The code of the query can be read in Listing 1.3. The result of the query is:

isMale	*averageWeight*	*maximumWeight*	*leastBirthDate*	*personCount*	*weightSum*
false	60.0	60.0	2001-01-01T00:00:00	1	60.0
true	115.0	120.0	2001-01-01T00:00:00	2	230.0

Example 6 (Continues Ex. 2). We now show a *query over a simple hierarchy of classes*, in particular showing the case of inheritance of attributes in subclassing: Find the name of all the young heavy weighted men. The source code is in Listing 1.4. The result of the query reads as:

personID	*name*
1	John
2	Paul

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?id ?name ?isMale ?bd ?weight
WHERE
{
        ?x rdf:type <http://example.org/Person> .
        ?x <http://example.org/Person/personID> ?id .
        ?x <http://example.org/Person/name> ?name .
        ?x <http://example.org/Person/sex> ?isMale .
        ?x <http://example.org/Person/birthDate> ?bd .
        ?x <http://example.org/Person/birthDate> ?bd .
        ?x <http://example.org/Person/weight> ?weight .
        ?x <http://example.org/Person/name> ?name .
        ?x <http://example.org/Person/name> ?name .
        FILTER ( strstarts(str(?name), 'M') && ?isMale = false && ?bd >= '2001-01-01T00:00:00'^^xsd:dateTime
        && ?bd <= '2001-12-31T00:00:00'^^xsd:dateTime && ?weight < 70
        && regex(str(?name), 'r', "i") && strends(str(?name), 'y') )
}
ORDER BY DESC(?name)
LIMIT 10
OFFSET 0
```

Listing 1.1. SPARQL query for all the females that were born in 2001 that weigh less than 70 kilos, and her name optionally starts with an M, contains an r and ends with a y

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT (AVG(?weight) AS ?averageWeight)
WHERE
{
        ?x rdf:type <http://example.org/Person> .
        ?x <http://example.org/Person/weight> ?weight .
        ?x <http://example.org/Person/sex> ?isMale .
        FILTER ( ?isMale = true )
}
```

Listing 1.2. SPARQL query for the average weight of the men

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?isMale (AVG(?weight) AS ?averageWeight) (MAX(?weight) AS ?maximumWeight) (MIN(?bd) AS ?leastBirthDate)
(COUNT(?id) AS ?personCount) (SUM(?weight) AS ?weightSum)
WHERE
{
        ?x rdf:type <http://example.org/Person> .
        ?x <http://example.org/Person/sex> ?isMale .
        ?x <http://example.org/Person/weight> ?weight .
        ?x <http://example.org/Person/weight> ?weight .
        ?x <http://example.org/Person/birthDate> ?bd .
        ?x <http://example.org/Person/personID> ?id .
        ?x <http://example.org/Person/weight> ?weight .
}
GROUP BY ?isMale
```

Listing 1.3. SPARQL query for categorizing people by sex and computing the average and maximum weight, least birthdate, person count, and sum of weights

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?personID ?name
WHERE
{
        ?x rdf:type <http://example.org/HeavyYoungMan> .
        ?x <http://example.org/Person/personID> ?personID .
        ?x <http://example.org/Person/name> ?name .
}
LIMIT 10
OFFSET 0
```

Listing 1.4. SPARQL query for finding the name of all the young heavy weighted men

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?personID ?name ?p ?phoneID ?phoneNumber ?phonePrice
WHERE
{
        ?x rdf:type <http://example.org/HeavyYoungMan> .
        ?x <http://example.org/Person/personID> ?personID .
        ?x <http://example.org/Person/name> ?name .
        ?x <http://example.org/Person/ref-phone> ?p .
        ?p rdf:type <http://example.org/Phone> .
        ?p <http://example.org/Phone/phoneID> ?phoneID .
        ?p <http://example.org/Phone/number> ?phoneNumber .
        ?p <http://example.org/Phone/price> ?phonePrice .
        FILTER (strstarts(str(?name), 'John') && regex(str(?phoneNumber), '555', "i") && ?phonePrice >= 100)
}
LIMIT 10
OFFSET 0
```

Listing 1.5. SPARQL query for finding all the heavy men named John that have a phone containing 555 in its number and with a price of at least 200 dollars

Example 7 (Continues Ex. 2). We now show a *query over an association*: Find all the heavy men named John that have a phone containing 555 in its number and with a price of at least 200 dollars. The source code of the query is presented in Listing 1.5 and its result is:

personID	*name*	*p*	*phoneID*	*phoneNumber*	*phonePrice*
1	John	http://example.org/Phone/phoneID=1	1	555-1234	200.0
1	John	http://example.org/Phone/phoneID=2	2	555-1235	220.0

Example 8 (Continues Ex. 2). As our last example, we present a *totalization query over a composition*: Find the average price of phones whose owner weighs between 110 and 120 kg. The source code of the query is in Listing 1.6. The result of the query is:

averagePrice
216.66666666666666

3 A Wizard for Writing SPARQL Queries

Now we present a wizard for writing the queries presented previously in a visual way. We based our approach on the Query-By-Example (QBE) paradigm where queries are specified by giving symbolic examples of the information to be retrieved. As in most QBE solutions, our program uses the usual form called QBE grid to indicate the subject, predicate, and object of the triples involved in the

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT (AVG(?phonePrice) AS ?averagePrice)
WHERE
{
        ?phone rdf:type <http://example.org/Phone> .
        ?phone <http://example.org/Phone/price> ?phonePrice .
        ?phone <http://example.org/Phone/ref-owner> ?phoneOwner .
        ?phoneOwner rdf:type <http://example.org/Person> .
        ?phoneOwner <http://example.org/Person/weight> ?ownersWeight .
        ?phoneOwner <http://example.org/Person/weight> ?ownersWeight .
        FILTER (?ownersWeight >= 110 && ?ownersWeight <= 120)
}
```

Listing 1.6. SPARQL query to find the average price of phones whose owner weighs between 110 and 120 kg

query, the conditions they have to satisfy if a totalization or grouping is involved and aliases for results. The names of properties and concepts are presented synthetically to avoid the information overload associated with full IRIs. As in all QBE environments, there is a parser that can convert the user's actions into statements expressed in a manipulation language, in this case, SPARQL. Behind the scenes, it is this statement that is executed. A suitably comprehensive front-end can minimize the burden on the user to remember the finer details of SPARQL, and it is easier and more productive for end-users (and even programmers) to select concepts and properties by selecting them rather than typing in their names.

We now address a brief description of the wizard. The ontology to be queried has to be loaded into the system. The limitations of the current status of the system include that only one ontology can be queried at a time. The ontology that is queried cannot reference other ontologies except the one that defines the basic datatypes. Our implementation addresses the visual specification employing a form, then generates automatically the source code of the equivalent SPARQL query and this query is evaluated against the ontology using the RDF4J library (see `https://rdf4j.org/`) and then generates a web page showing the result of the query (see accompanying online documentation).

For space reasons, we will only discuss how the queries of Sect. 2 are expressed in our tool. In Fig. 4, we can see how the SPARQL query presented in Lst. 1.1 is visually codified. The user has to name the subject of the triples (viz., x), then establish the concept the subject belongs to (viz., Person), and then for each property that the user desires a column in the result, has to assign an alias and establish a condition, that can be deemed as invisible and/or optional if desired (viz., property sex with alias *isMale* and value equal to `false`). Notice how the user interface hides the low-level details of IRIs from the user.

In Fig. 5, we can see visual specification of the SPARQL query of Listing 1.2. In this case, as this totalization query must compute a single number (i.e., the average weight of the men), only one field has to be made visible and the result column for this property has to be named (viz., *averageWeight*). More importantly, in this kind of query a totalization function has to be selected (viz., *Average*).

Subject	Concept	Property	Alias	Order	Visible	Function	Operator	Value	Optional	Result
x	Person	Person/personID	id	<none>	☑	<none>	<none>		☐	
x	Person	Person/name	name	descending	☑	<none>	starts with	M	☐	
x	Person	Person/sex	isMale	<none>	☑	<none>	=	false	☐	
x	Person	Person/birthDate	bd	<none>	☑	<none>	>=	2001-01-01	☐	
x	Person	Person/birthDate	bd	<none>	☐	<none>	<=	2001-12-31	☐	
x	Person	Person/weight	weight	<none>	☑	<none>	<	70	☐	
x	Person	Person/name	name	<none>	☐	<none>	contains	r	☐	
x	Person	Person/name	name	<none>	☐	<none>	ends with	y	☐	

Fig. 4. Querying people with several conditions

Subject	Concept	Property	Alias	Order	Visible	Function	Operator	Value	Optional	Result
x	Person	Person/weight	weight	<none>	☑	Average	<none>		☐	averageWeight
x	Person	Person/sex	isMale	<none>	☐	<none>	=	true	☐	

Fig. 5. Finding the average weight of the men

In Fig. 6, we can see the visual specification of the SPARQL query of Listing 1.3. This kind of query shows how to partition a set of individuals using the values of a property (in this case *sex*). As the *sex* property is of Boolean type, the set of people is partitioned into two disjoint subsets (assuming that the sex for all people is determined), this is done by using the *Group* function. For each sex, the usage of several totalization functions are shown: *Average*, *Max*, *Min*, *Count*, and *Sum* for computing the average and maximum weight, least date of birth, the number of people and the sum of their weights. Notice that variables for the results must be defined (viz., *averageWeight*, *maximumWeight*, *leastBirthDate*, *personCount*, and *weightSum*.

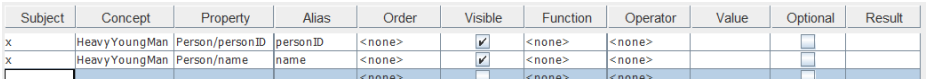
Subject	Concept	Property	Alias	Order	Visible	Function	Operator	Value	Optional	Result
x	Person	Person/sex	isMale	<none>	☑	Group by	<none>		☐	
x	Person	Person/weight	weight	<none>	☑	Average	<none>		☐	averageWeight
x	Person	Person/weight	weight	<none>	☑	Max	<none>		☐	maximumWeight
x	Person	Person/birthDate	bd	<none>	☑	Min	<none>		☐	leastBirthDate
x	Person	Person/personID	id	<none>	☑	Count	<none>		☐	personCount
x	Person	Person/weight	weight	<none>	☑	Sum	<none>		☐	weightSum

Fig. 6. Totalizing functions according to sex

In Fig. 7, we see that querying a hierarchy of classes is straightforward as the inheritance of properties (attributes) is computed seamlessly. In this case, it is shown how the names and identifiers of people can be used for the class HeavyYoungMan which is a subclass (sub-concept) of Person. Notice that in particular, this is the visual presentation of the SPARQL query of Listing 1.4.

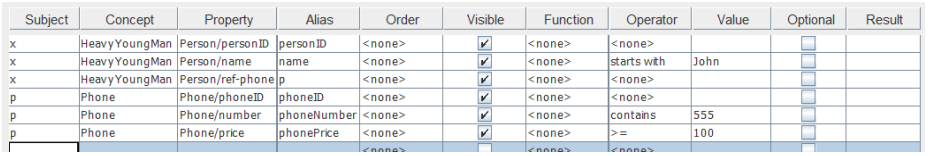
In Fig. 8, we see how an association between classes can be queried (this is the visualization of the SPARQL query in Listing 1.5). In particular, two variables for the subjects have to be defined: x for people and p for phones. Notice in the third row how x is associated with p by means of the Person/ref-phone property.

Finally, in Fig. 9, we can observe the visual expression of the SPARQL query in Listing. 1.6 showing how to perform a totalization over an association. Notice



Subject	Concept	Property	Alias	Order	Visible	Function	Operator	Value	Optional	Result
x	HeavyYoungMan	Person/personID	personID	<none>	☑	<none>	<none>		☐	
x	HeavyYoungMan	Person/name	name	<none>	☑	<none>	<none>		☐	

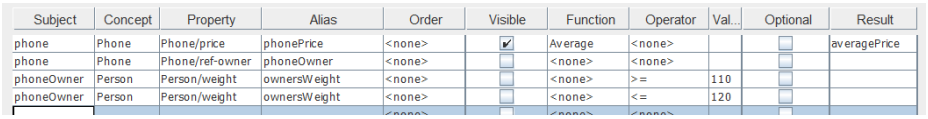
Fig. 7. Querying a hierarchy: Find the name of heavy men



Subject	Concept	Property	Alias	Order	Visible	Function	Operator	Value	Optional	Result
x	HeavyYoungMan	Person/personID	personID	<none>	☑	<none>	<none>		☐	
x	HeavyYoungMan	Person/name	name	<none>	☑	<none>	starts with	John	☐	
x	HeavyYoungMan	Person/ref-phone	p	<none>	☑	<none>	<none>		☐	
p	Phone	Phone/phoneID	phoneID	<none>	☑	<none>	<none>		☐	
p	Phone	Phone/number	phoneNumber	<none>	☑	<none>	contains	555	☐	
p	Phone	Phone/price	phonePrice	<none>	☑	<none>	>=	100	☐	

Fig. 8. Querying an association: Find the heavy men with their phones

how again two different variables for the subject have to be defined for indicating the association between subject and objects in RDF triples and also how the *averagePrice* variable in the result column has to be declared.



Subject	Concept	Property	Alias	Order	Visible	Function	Operator	Val...	Optional	Result
phone	Phone	Phone/price	phonePrice	<none>	☑	Average	<none>		☐	averagePrice
phone	Phone	Phone/ref-owner	phoneOwner	<none>	☐	<none>	<none>		☐	
phoneOwner	Person	Person/weight	ownersWeight	<none>	☐	<none>	>=	110	☐	
phoneOwner	Person	Person/weight	ownersWeight	<none>	☐	<none>	<=	120	☐	

Fig. 9. Querying an association: Find the average price of phones of people weighing between 110 and 120 kg

4 Related Work

Swipe [7] implements a search-by-example approach to query Wikipedia where naive users can enter query conditions directly on the Infobox of a Wikipedia page, and then Swipe uses these conditions to generate equivalent SPARQL queries and execute them on DBPedia. As Swipe, our system makes querying ontologies user-friendly but our system is more general as it is not limited to DBPedia. Our system could do something similar by, given a Wikipedia page, first downloading the associated DBPedia OWL ontology and loading it in GF, then expressing the query on the GF wizard and executing it. Like DBPedia, iSparQL end-point [8], our system allows also us to enter a SPARQL query in text form to be submitted against the current ontology loaded in the program. Diaz et al. [9] present SPARQLByE (for SPARQL by Example) which is a front-end for DBPedia where a naive user can input positive and negative examples of what he desires, and then the system uses a reverse engineering heuristic to induce a SPARQL query. As our system, SPARQLByE abstracts full IRIs and works with joins and optional statements. Horridge and Musen [10] present Snap-SPARQL, a Java framework for working with SPARQL and OWL, that includes a parser, axiom template API, SPARQL algebra implementation, and graphical user interface components for reading, processing, and executing SPARQL

queries. Our system does this by using an auxiliary library and provides a visual interface for the composition of queries. In brief, our solution provides a concrete way of writing SPARQL queries over legacy data expressed as an OWL ontology without requiring the user to know explicitly SPARQL syntax and it is available as a downloadable standalone application unlike many of the solutions reviewed here that are custom built for specific ontologies. However, referring to external ontologies is not supported in the current version of GF's implementation.

5 Conclusions and Future Work

We presented an extension for the GF framework for ontology integration to allow a naive user to build SPARQL queries visually by using a Query-By-Example approach. We presented several examples of how the approach works. The limitations of our approach include that in its current state it is only capable of working with a single data source comprised of an OWL ontology loaded into memory. Then it does not allow to make use of several data sources at the same time nor make the query refer to other data sources. We have not tested our implementation with naive users to account for its usability in real cases. Part of our current research is focused on solving these matters.

Acknowledgments. This work was supported by Secretaría General de Ciencia y Técnica, Universidad Nacional del Sur, Argentina, and by Comisión de Investigaciones Científicas de la Provincia de Buenos Aires (CIC-PBA).

References

1. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American **284**(5) (2001) 34–43
2. Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., Zakharyaschev, M.: Ontology-Based Data Access – A Survey. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18). (2018) 5511–5519
3. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language for RDF W3C recommendation 21 march 2013 (2013) https://www.w3.org/TR/rdf-sparql-query/.
4. Gómez, S.A., Fillottrani, P.R.: Ontology Metrics and Evolution in the GF Framework for Ontology-Based Data Access. In: Computer Science – CACIC 2021. Springer International (2022)
5. Zloof, M.M.: Query by Example. In: NCC (proceedings). Volume 44. Anaheim, California: AFIPS (May 1975)
6. Baader, F., Horrocks, I., Lutz, C., Sattler, U.: An Introduction to Description Logic. Cambridge University Press (2017)
7. Atzori, M., Zaniolo, C.: Swipe: searching wikipedia by example. In: Proceedings of the 21st International Conference on World Wide Web. (2012) 309–312
8. Grobe, M.: RDF, Jena, SparQL and the Semantic Web. In: SIGUCCS '09: Proceedings of the 37th annual ACM SIGUCCS fall conference: communication and collaboration. (oct 2009) 131–138
9. Diaz, G., Arenas, M., Benedikt, M.: SPARQLByE: querying RDF data by example. Proceedings of the VLDB Endowment **9** (09 2016) 1533–1536
10. Horridge, M., Musen, M.: Snap-SPARQL: A Java Framework for Working with SPARQL and OWL. In: International Experiences and Directions Workshop on OWL. (04 2016) 154–165