Snap-SPARQL: A Java Framework for working with SPARQL and OWL

Matthew Horridge and Mark Musen

Stanford Biomedical Informatics Research Group
Stanford University, California, USA

Abstract. We present Snap-SPARQL, which is a Java framework for working with SPARQL and OWL. The framework includes a parser, axiom template API, SPARQL algebra implementation, and graphical user interface components for reading, processing and executing SPARQL queries under the SPARQL 1.1 OWL Entailment Regime. While the framework was originally designed to support the implementation of a SPARQL teaching aid in the form of a Protégé plugin, we believe that it is more generally useful and may be of interest to developers and researchers working on SPARQL 1.1 OWL entailment regime implementations and optimisations. The framework is open source and pluggable.

1 Introduction

In March 2013 the World Wide Web Consortium published the SPARQL 1.1 Recommendation—a set of nine documents that specify a query language and protocol for querying and manipulating RDF graphs [2]. Although a point increment, this latest version of SPARQL includes many new language features such as aggregates, sub-queries, a new suite of builtin functions, and path expressions. Besides these many language enhancements, SPARQL 1.1 also includes a sub-specification that describes how SPARQL queries should be evaluated under different *entailment regimes* [3].

An entailment regime is a specification that precisely defines how SPARQL queries should be answered with respect to a given entailment relation. The SPARQL 1.1 specification defines several out-of-the-box entailment regimes, which include the *Simple*, *RDF-Schema*, and *OWL 2 Direct Semantics* entailment regimes. For any given SPARQL query, the set of answers depends upon the entailment regime in question, and the answers for one entailment regime may be different to the answers for a different entailment regime. For example, consider the RDF graph below[1]

```
:PaloAlto :isLocatedInState :California .
:California :isLocatedIn :USA .
:isLocatedInState rdfs:subPropertyOf :isLocatedIn .
:isLocatedIn rdf:type owl:TransitiveProperty .
```

The following query asks for the pairs `?x` and `?y` where `?x` is located in `?y`.

[1] Where we omit prefix declarations for the sake of brevity.

```
SELECT ?x ?y
WHERE {
        ?x :isLocatedIn ?y
}
```

Under the *Simple* entailment regime, the set of bindings for `?x` and `?y` contains the single binding ⟨`:California, :USA`⟩, while under the *RDFS* entailment regime the set of bindings includes the bindings for the Simple regime plus ⟨`:PaloAlto, California:`⟩ (entailed in part by the sub-property axiom), and finally, under the *OWL 2 Direct Semantics* entailment regime the set also includes the previous two bindings plus ⟨`:PaloAlto, USA:`⟩ (entailed in part by the transitive property axiom).

As far as end users are concerned, the practical impact of the SPARQL 1.1 Entailment Regimes specification is that the answers to SPARQL 1.1 queries can potentially include information that follows *implicitly* from the dataset that is being queried. In other words, query answers may now be obtained from *inferred* information, and for OWL users, this means that SPARQL 1.1 can sensibly be used for querying OWL ontologies.

While having a standardised query language for OWL is a huge plus, users also need tools in order to be able to write and execute queries. Furthermore, not only are middleware tools that perform the actual query answering needed, but there is also a need for user-facing tools that assist users in writing queries that are well-formed for both a given entailment regime and a given dataset (set of ontologies). With this in mind, we present a framework called Snap-SPARQL that assists end-users of environments like Protégé in writing SPARQL queries that are well-formed for the OWL entailment regime.

The Snap-SPARQL framework consists of (1) a SPARQL parser, which parses queries that are well formed for the OWL entailment regime and also provides error descriptions that allow meaningful auto-correction/completion to be provided, (2) data-structures for representing basic graph patterns at the level of *axiom templates*, or axioms that may contain variables, (3) an executable implementation of the bulk of the SPARQL 1.1 query algebra that provides the various SPARQL "bells and whistles" beyond basic graph pattern evaluation, (4) a graphical user interface component that contains syntax highlighting and auto-completion, which assists end-users of tools like Protégé in writing SPARQL queries that are well-formed for the OWL Entailment Regime, and (5) a Protégé plugin that uses the aforementioned components to enable Protégé users to query OWL ontologies that are contained in a Protégé workspace.

Finally, the framework is pluggable. It allows reasoners that implement the OWL API [4] reasoner interfaces such as ELK, FaCT++, HermiT, JFact and Pellet to be plugged into it. It also allows basic graph pattern evaluation implementations such as OWL-BGP or the Derivo SPARQL-DL engine[2] to be used for the core graph pattern (axiom template) evaluation operations.

[2] http://www.derivo.de/en/resources/sparql-dl-api.html

```
PREFIX  rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX     : <http://owl.man.ac.uk/2005/07/sssw/people#>

SELECT ?personName ?petName
WHERE {
        ?person rdf:type :person ;
                rdfs:label ?personName ;
                :has_pet ?pet .
        ?pet rdf:type :cat ;
             rdfs:label ?petName

        FILTER (STRSTARTS(?petName, "T"))

        OPTIONAL {
                ?pet foaf:gender ?gender .
        }
}
```

Fig. 1: An example SPARQL query. The query asks for people who have pets that are cats, and lists them along with their names and, if known, gender.

2 Preliminaries

SPARQL is based on Turtle Syntax (Terse RDF Triple Language) [7]. SPARQL queries typically consist of various clauses and blocks, which specify *basic graph patterns* to be matched along with keywords that join, filter and extend the solution sequences to these patterns.

An example SPARQL query, targeted at the *people and pets* tutorial ontology is shown in Figure 1. The solutions to this query are the people who have cats that have names that begin with the letter "T", along with the name of the cat and, if known, its gender. Curly brackets denote *group graph patterns*, and contiguous triple patterns form *basic graph patterns*. In this query, there are two basic graph patterns split by the OPTIONAL keyword, which performs a *left join* on the solution sequences to the two basic graph patterns. The query prologue contains various *prefix declarations* consisting of a prefix label and a prefix. Various variables are used within the query body, but only three are projected: ?personName ?petName and ?gender.

3 Components and Functionality

The Snap-SPARQL framework contains several components that may be split into programmer-facing and end-user-facing components. We provide a few brief details of each of these components below.

3.1 An Axiom Template API

Axiom templates are essentially OWL axioms that allow variables to be placed in the positions of IRIs and Literals. For example, {ClassAssertion(:person ?x), AnnotationAssertion(rdfs:label ?x ?y)} is a set containing two axioms templates[3]. The first template is a ClassAssertion axiom that contains the variable ?x in the individual position. Hence bindings for ?x are the entailed instances of :person. The second template is an AnnotationAssertion axiom that contains two variables ?x and ?y in the subject and object position respectively. Intuitively, bindings for ?y are the labels of bindings for ?x. Together, the templates form a query for the labels of the instances of :person.

According to the OWL entailment regime specification, for a query to be considered well-formed it must be possible to "lift" the basic graph patterns in that query into OWL axiom templates. In other words, the basic graph patterns must corresponds to a triple-based serialisation of some OWL axioms.

Axiom templates essentially provide a high-level view of basic graph patterns that makes sense at the level of OWL, and they essentially abstract away from the triple-based syntax of SPARQL. Snap-SPARQL provides an API for working with axiom templates that allows implementors to avoid dealing with triples and the intricacies of the triple-based serialisation of OWL axioms. The axiom template API itself is inspired by the axiom and class expression structures in the OWL API, and it closely follows the OWL functional syntax specification in design.

3.2 A SPARQL Parser

The framework includes a parser that consumes SPARQL syntax and transforms it into high-level axiom templates and other data structures that represent SPARQL queries at an abstract level that is syntax independent. The parser was designed with supporting context-sensitive auto-completion in mind and it forms part of an editing kit that is part of the framework (see Section 3.5)

At the time of writing the parser supports most of the SPARQL 1.1 specification in terms of language features. However, some features, such as property path expressions, are not supported. In terms of parsing axiom templates, the parsing of complex class expressions is not *currently* supported, but this is planned as part of future work (see Section 4).

Because Snap-SPARQL is designed to handle queries under the OWL entailment regime, the parser will only consume queries that conform to the syntactic restrictions of this regime rather than more general SPARQL queries. In particular, queries must be written so that they can be parsed into well-formed axiom templates, so sets of triples that do not correspond to the triple-based serialisation of OWL axioms, such as {?x rdfs:subClassOf :pete.} (where :pete is an individual) or {?x :hasParent "Mary".} (where :hasParent is an object property), will cause an error. Uses of punning within group graph patterns (between curly braces) will also raise an error, in accordance with the OWL entailment regime specification.

[3] For the sake of brevity, we have written these axiom templates using a variant of the OWL Functional Syntax

Finally, the parser is designed to make it easy for end-users of tools such as Protégé to write SPARQL queries. So for example, the parser does not require all terms appearing in a query to be declared (typed with `rdf:type`)—if it can determine the type of a term from the underlying ontology then it will do so. Similarly, in some cases the type of variables may unambiguous, for example given {?x rdfs:subClassOf :cat.}, the variable ?x must be of the type owl:Class, i.e. a class variable, and the parser does not require this variable to be typed.

3.3 An implementation of the SPARQL Algebra

The SPARQL Algebra is a set of operators that together can be used to form *SPARQL Algebra Expressions*. An example of an algebra expression, that corresponds to the concrete SPARQL query shown in Figure 1, is shown below. An algebra expression, together with data, represents a high-level abstract view of a SPARQL query that is independent from syntactic shortcuts or variations, and syntax-level keywords and punctuation. The algebra is used to define the semantics of SPARQL and it can also be used to derive a canonical procedure for query answering.

```
(Project
    (OrderBy
        (ToList
            (Filter
                (LeftJoin
                    (Bgb
                        ClassAssertion(:person, ?person)
                        AnnotationAssertion(rdfs:label, ?person, ?personName)
                        ObjectPropertyAssertion(:has˙pet, ?person, ?pet)
                        ClassAssertion(:cat, ?pet)
                        AnnotationAssertion(rdfs:label, ?pet, ?petName)
                    )
                    (Bgp
                        DataPropertyAssertion(foaf:gender ?pet ?gender)
                    )
                )
                BuiltIn(STRSTARTS, ?petName, "T"^^xsd:string)
            )
        )
        OrderCondition(ASC, ?petName)
    )
    ?personName
    ?petName
)
```

Fig. 2: An example of a SPARQL Algebra Expression. This particular algebra expression corresponds to the concrete SPARQL query shown in Figure 1.

Snap-SPARQL provides an implementation of the SPARQL query algebra for the purposes of query analysis and query plan optimisation, and also to act as a reference implementation for query execution. In the first case, developers of query engines may use the algebra API to generate more optimal query plans. In the second case, developers of implementations that provide basic graph pattern evaluation can simply concentrate on the algorithms for pattern matching and leave algebra operations, such as join, filter, extend, orderby and project to the framework.

3.4 Support for Pluggable Basic Graph Pattern Matching

At the most basic level, answering SPARQL queries involves computing solution-sequences to Basic Graph Patterns (Axiom Templates) and then processing these solution sequences in accordance with the SPARQL algebra mentioned in the previous section. The core operations here are performed by Graph Pattern matching implementations and these are pluggable in Snap-SPARQL. The framework ships with the *Derivo's SPARQL-DL*[4] as the default implementation, but it should also possible to plugin in some other off-the-shelf pattern matcher, such as OWL-BGP [6], with minimal effort. The benefit of this pluggable approach, is that researchers and developers who are interested in supporting the OWL entailment regime, and testing optimisations for query answering under this regime, can focus on axiom template evaluation implementation without worrying about other SPARQL features. In addition to this, Snap-SPARQL enables them to make their implementations available to a wider community, for use in Protégé, without too much extra effort.

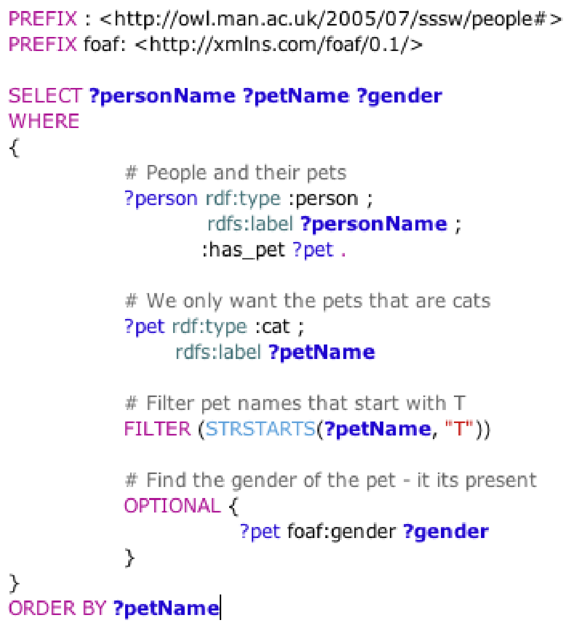
3.5 A SPARQL Editor

Writing SPARQL queries can be challenging for users. It requires them: (1) to be fairly well-versed with Turtle syntax in order to construct the basic graph patterns that form the core of any query, (2) to understand the various SPARQL keywords and how these can be used, (3) to correctly setup prefix names and prefixes and then use them consistently in the body of the query, and (4) to use the domain vocabulary in question such that the queries actually make sense. The situation becomes more challenging when basic graph patterns must be well-formed for a given entailment regime such as the OWL entailment regime.

In order to assist users in writing SPARQL queries the framework provides an editor component that can be reused in third party tools. The editor provides the kinds of features that one would expect in a modern development environment such as syntax high-lighting and auto-completion. These features are described in more detail below:

Syntax highlighting of keywords, variables and built-in vocabulary. An example of the syntax highlighting is shown in Figure 3, where highlighting has been applied to the example query shown in Figure 1 (note that the highlighting example also includes the addition of comments). Highlighting is applied to keywords, builtin vocabulary, builtin functions and variable names. Furthermore, projected and non-projected variables are distinguished with bold and regular weight fonts.

[4] http://www.derivo.de/en/resources/sparql-dl-api.html

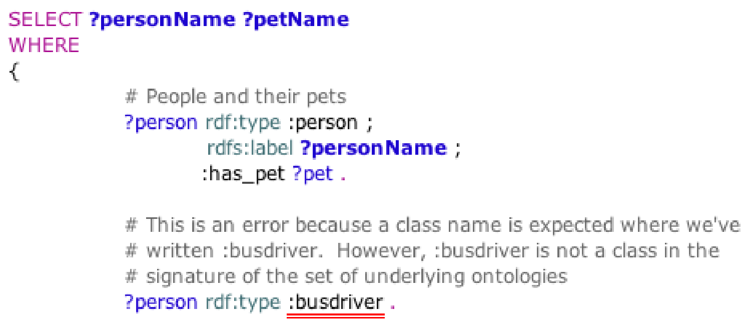


```
PREFIX : <http://owl.man.ac.uk/2005/07/sssw/people#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?personName ?petName ?gender
WHERE
{
            # People and their pets
            ?person rdf:type :person ;
                    rdfs:label ?personName ;
                   :has_pet ?pet .

            # We only want the pets that are cats
            ?pet rdf:type :cat ;
                 rdfs:label ?petName

            # Filter pet names that start with T
            FILTER (STRSTARTS(?petName, "T"))

            # Find the gender of the pet - it its present
            OPTIONAL {
                        ?pet foaf:gender ?gender
            }
}
ORDER BY ?petName
```

Fig. 3: An example of the syntax highlighting used in Snap-SPARQL. Keywords, variables, comments, functions and built-in vocabulary are highlighted. The tool distinguishes between projected and non-projected variables. For example, ?personName and ?petName are highlighted in bold because they are projected into the query result.

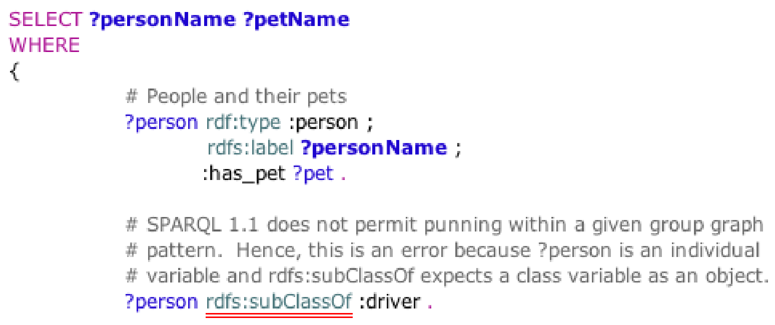
Auto-suggestion for `PREFIX` declarations. Dealing with prefixes in SPARQL can be painful. Indeed, this is an issue that related tools have addressed (see Section 5). Snap-SPARQL provides auto-completion for prefixes based on the IRIs of entities in the signature dataset ontology documents. Well-known prefix names, such as `dce:`, `foaf:` or `dbo:`, are suggested if the corresponding prefixes are present in the underlying dataset. Once prefixes have been declared, the editor will offer completions for terms in the body of the query based on these prefixes rather than offering full IRIs.

Error highlighting. The editor performs highlighting for different kinds of syntax/semantic errors. Two examples are shown in Figure 4 and Figure 5, where a double red underline indicates an error. The editor derives error information, both type and position, from the Snap-SPARQL parser. This means that it is capable of detecting several categories of errors, that go beyond SPARQL syntax violation errors that may be detected by a bog-standard SPARQL parser. For example, while both Figure 4 and Figure 5 show queries that are syntactically correct according to the SPARQL grammar, however they contain other kinds of errors. Specifically, Figure 4, shows an error where the *class name* :busdriver is not contained in the signature of the dataset ontologies. Figure 5 shows a kind of grammatical error that is only an error under the OWL entailment regime—in



```
SELECT ?personName ?petName
WHERE
{
        # People and their pets
        ?person rdf:type :person ;
                rdfs:label ?personName ;
                :has_pet ?pet .

        # This is an error because a class name is expected where we've
        # written :busdriver.  However, :busdriver is not a class in the
        # signature of the set of underlying ontologies
        ?person rdf:type :busdriver .
```

Fig. 4: An example of error checking and highlighting. Here, the name :busdriver has been used in a position that should be filled by a class (given the context provided by preceding statements). However, the signature of the underlying set of ontologies does not contain a corresponding term.

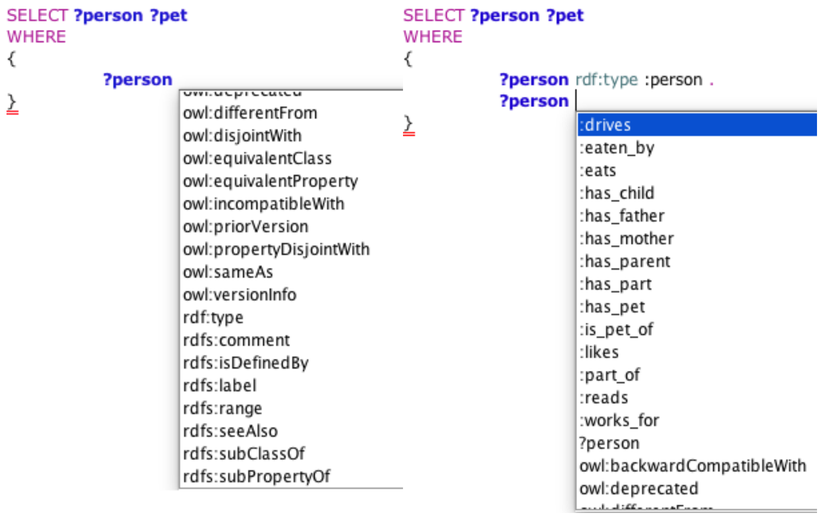


```
SELECT ?personName ?petName
WHERE
{
        # People and their pets
        ?person rdf:type :person ;
                rdfs:label ?personName ;
                :has_pet ?pet .

        # SPARQL 1.1 does not permit punning within a given group graph
        # pattern.  Hence, this is an error because ?person is an individual
        # variable and rdfs:subClassOf expects a class variable as an object.
        ?person rdfs:subClassOf :driver .
```

Fig. 5: Another example of error checking and highlighting. Here, the predicate rdfs:subClassOf has been used in a position that should be filled by an object property, data property or annotation property since :person is an individual variable (given the context provided by preceding statements).

this case, the predicate rdfs:subClassOf cannot be used with the variable ?person because this variable will bind to individuals based on its context.

Context-sensitive auto-completion. In conjunction with the aforementioned error checking, the editor offers context-sensitive auto-completion. Figure 6 shows two examples. On the left-hand side, the variable ?person has not been declared and its type cannot be derived from the context of the query. This means that the editor offers a large choice of possible completions that allow for the variable being a class, property, individual etc. On the-right hand side (after a bit more typing) the parser has a larger context to work with, and it is able to determine that ?person is an individual variable, hence the choice at the cursor is limited to object, data or annotation properties and builtin vocabulary that also applies to individuals such as rdf:type, owl:sameAs and owl:differentFrom. It should be noted that the auto-completion functionality is available in all con-



```
SELECT ?person ?pet
WHERE
{
            ?person
}
```

```
SELECT ?person ?pet
WHERE
{
            ?person rdf:type :person .
            ?person
}
```

Fig. 6: An example of auto-completion. Auto-completion is provided for keywords, variable names, function names, and terms at all positions.

texts, including for triple subjects, predicates and objects, as well as key words, function names and punctuation.

3.6 A Protégé Plugin

The final Snap-SPARQL component is a Protégé plugin that exposes all of the previously described functionality to end-users of Protégé. In particular, the plugin provides the editing capability described above along with a mechanism to view query results. The plugin is fairly tightly integrated into the Protégé environment, as it uses the ontologies that are loaded into the active Protégé workspace, along with the currently selected reasoner for the purpose of providing inferred information to the basic graph pattern evaluator component of the framework. Finally, the plugin is compatible with Protégé 5, it is open source and may be downloaded from `http://github.com/protegeproject/snap-sparql-plugin`.

4 Limitations and Future Work

While the Snap-SPARQL framework supports most of the SPARQL 1.1 query language, in particular the features that make sense in the context of the OWL entailment regime there are some features that are not supported. We briefly discuss some of these limitations along with future implementation plans.

SPARQL 1.1 contains *property path expressions* that allow regular-expression-like paths of properties to be matched. However, these are not supported by the Snap-SPARQL framework. While this would be a significant limitation under simple entailment, it is not clear how much of a limitation it actually is under the OWL entailment regime. This is because, one of motivations for property path expressions is that they enable queries to be written whose answers involve some kind of "transitivity" such as `{?x rdfs:subClassOf+ ?y}` or `{?x :partOf+ ?y}`. In these cases, under the OWL entailment regime, transitivity comes "for free" according to the semantics of the language, for example if A is a subclass of B and B is a subclass of C, then A is also a subclass of C. For more complex cases that involve choices e.g. the lack of property path expressions imposes some inconvenience and queries such as `{?x rdfs:label | dce:title ?y}`, will need to be written by the user, if possible.

SPARQL Update makes it possible to modify graphs in the query dataset. In the case of OWL, this would involve adding and removing axioms to a set of ontologies. While it seems like SPARQL Update for OWL ontologies would be useful, Snap-SPARQL does not currently support this feature.

Specification of specific datasets using the `GRAPH` keyword is not supported. The framework currently assumes that the query dataset consists of a set ontology documents that are specified by an imports closure in accordance with the OWL semantics. We may support more selective queries using the GRAPH keyword (perhaps restricted to asserted information) in the future.

While Snap-SPARQL supports SPARQL's `MINUS` keyword, meaning that one solution sequence can be subtracted from another thereby providing a form of negation by failure, it does not currently support `NOT EXISTS`. We intend to add support for this.

Snap-SPARQL does not currently support complex OWL class expressions. At the moment, queries are essentially limited to SPARQL-DL [10] queries. These kind of queries correspond to mixed ABox/TBox queries over class hierarchies, property hierarchies, disjoint classes, property domain, property range, class assertions, property assertions, same individual, different individuals and annotation assertions. All of this still operates under the OWL entailment regime. Part of the reason for the lack of support for complex class expressions is that these are tricky to write in a triple-based syntax such as turtle, for example ?x SubClassOf hasPart some ?part corresponds to {?x rdfs:subClassOf [rdf:type owl:Restriction; owl:onProperty :hasPart; owl:someValuesFrom ?part]}. We are therefore currently considering whether to support complex class expressions via Terp [9], whereby Manchester Syntax [5] can be embedded into SPARQL queries, or by some form of auto-completion that automatically inserts sets of triple patterns that correspond to well-formed class expressions.

Finally, as part of future work, we are considering supporting explanation of query results using justification based explanation techniques, and we are also considering a web-based version of the editor, in particular for WebProtégé.

5 Related Work

The challenges of writing SPARQL queries using a plain text editor (as provided by many SPARQL end points) are somewhat obvious and have not gone unnoticed by the RDF/SPARQL community. Two tools that are related to the work here are Flint Editor[5] and YASGUI [8], where the latter is based on the former. Both of these tools are Web-based and make it possible to query public SPARQL endpoints. They provide the usual affordances such as syntax highlighting and some form of autocompletion. However, neither of these tools are "entailment regime aware", in particular they are not OWL entailment regime aware. This means that queries are based upon the asserted graph. Furthermore, they do not provide a distinction between domain vocabulary and vocabulary used to encode OWL axioms. They do not perform error checking to the extent that Snap-SPARQL does, which means that they will happily accept nonsensical queries such as { `?x rdf:type :John` }, where John is an individual name, or { `:C :hasPart ?p` }, where `:C` is a class name. Finally, related to the last point, auto-completion is somewhat impoverished when compared to Snap-SPARQL and offers nonsensical suggestions in some cases. For example, attempting to complete `''SELECT ?x WHERE { ?x rdfs:subClassOf dbo:Company . ?x dbo:''` offers any kind of predicate that is present in the graph along with language vocabulary—it offers object properties, even though `?x` must be a class variable, and it also offers language vocabulary such as rdfs:range or rdfs:domain, which does not make sense in this context. On the other hand, both of these tools are fully-fledged general SPARQL 1.1 query tools, they are Web-based, understand commonly used prefixes and offer querying of public SPARQL endpoints.

OWL-BGP [6] is a library for performing basic graph pattern matching under the OWL entailment regime. It also includes an axiom template API and triple consumer for lifting triples into axiom templates before query evaluation takes place. Clients typically do not interact with OWL-BGP directly, as the library itself does not contain a SPARQL algebra implementation. Instead, OWL-BGP interfaces with the Jena RDF framework [1], which provides this functionality. It's possibly the case that OWL-BGP could also be plugged into Snap-SPARQL without too much difficulty.

The Derivo SPARQL-DL library[6] provides basic graph pattern matching under the OWL entailment regime for SPARQL-DL queries. Snap-SPARQL currently uses this library as the default implementation for basic graph pattern matching.

Pellet [11] is an OWL reasoner that supports SPARQL queries over ABoxes. For mixed TBox and ABox queries Pellet falls back to the native Jena RDF SPARQL implementation.

Finally, Stardog[7] is a commercial graph (RDF) database from Complexible that supports OWL reasoning. It supports SPARQL as its native query language, and essentially allows the SPARQL-DL queries to be answered under the OWL entailment regime.

[5] `http://openuplabs.tso.co.uk/demos/sparqleditor`

[6] http://www.derivo.de/en/resources/sparql-dl-api.html

[7] `http://stardog.com`

6 Availability

Snap-SPARQL has been developed as part of the Protégé Project. It is open source and freely available at https://github.com/protegeproject/snap-sparql-query.

Acknowledgements This work was supported by Grants GM103316 and R01GM086587 from the National Institute of General Medical Sciences of the United States National Institutes of Health.

References

1. Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations. In Stuart Feldman, Mike Uretsky, Mark Najork, and Craig Wills, editors, *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83, New York, NY, USA, May 2004. ACM.
2. Birte Glimm and Chimezie Ogbuji. SPARQL 1.1 Entailment Regimes. Technical report, World Wide Web Consortium, 2012.
3. Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. Technical report, World Wide Web Consortium, 2012.
4. Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL ontologies. *Semantic Web*, 2(1):11–21, February 2011.
5. Matthew Horridge and Peter F. Patel-Schneider. Manchester OWL Syntax for OWL 1.1. In *OWL: Experiences and Directions (OWLED)*, 2008.
6. Ilianna Kollia, Birte Glimm, and Ian Horrocks. SPARQL query answering over OWL ontologies. In *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29-June 2, 2011, Proceedings, Part I*, volume 6643, pages 382–396. Springer, 2011.
7. Eric Prud'hommeaux, Gavin Carothers, David Beckett, and Tim Berners-Lee. Turtle terse RDF triple language. Technical report, W3C – World Wide Web Consortium, February 2014.
8. Laurens Rietveld and Rinke Hoekstra. YASGUI: not just another SPARQL client. In Philipp Cimiano, Miriam Fernández, Vanessa Lopez, Stefan Schlobach, and Johanna Völker, editors, *The Semantic Web: ESWC 2013 Satellite Events - ESWC 2013 Satellite Events, Montpellier, France, May 26-30, 2013, Revised Selected Papers*, volume 7955 of *Lecture Notes in Computer Science*, pages 78–86. Springer, May 2013.
9. Evren Sirin, Blazej Bulka, and Michael Smith. Terp: Syntax for OWL-friendly SPARQL queries. In Evren Sirin and Kendall Clark, editors, *Proceedings of the 7th International Workshop on OWL: Experiences and Directions (OWLED 2010), San Francisco, California, USA, June 21-22, 2010*, volume 614 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.
10. Evren Sirin and Bijan Parsia. Sparql-dl: Sparql query for owl-dl. In *OWL: Experiences and Directions (OWLED)*, 2007.
11. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2), 2007.