SPAM: A SPARQL Analysis and Manipulation Tool

Andrés Letelier
Dept. of Computer Science
PUC Chile
aileteli@uc.cl

Jorge Pérez
Dept. of Computer Science
Universidad de Chile
jperez@dcc.uchile.cl

Reinhard Pichler Sebastian Skritek
Faculty of Informatics
Technische Universität Wien
[pichler,skritek]@dbai.tuwien.ac.at

ABSTRACT

SQL developers are used to work with elaborate tools which help them in writing queries. In contrast, the creation of tools to assist users in the development of SPARQL queries is still in its infancy. In this system demo, we present the SPARQL Analysis and Manipulation (SPAM) tool, which provides help for the development of SPARQL queries. The main features of the SPAM tool comprise an editor with both text and graphical interface, as well as various functions for the static and dynamic analysis of SPARQL queries.

1. INTRODUCTION

The RDF data model and the SPARQL query language are two of the core technologies of the data layer of the Semantic Web. Both languages have gone through W3C standardization processes and are nowadays widely adopted and used in several areas, as diverse as bio-informatics, social networks, and data integration. Moreover, the study of these two standards has also gained momentum in the database community, as can be witnessed by the number of publications on these topics in some major database venues during the last years [1, 11, 9, 7, 10, 2, 3, 6, 4, 5].

Although RDF repositories and SPARQL query engines have reached some level of maturity, there is still a lack of tools to support SPARQL query developers. Contrast this with a typical relational DB engine, in which tools supporting SQL query creation, analysis and optimization are an integral part of any developer environment. This lack of support can be explained in several terms. First, we are in the presence of a new scenario in which previous DB knowledge is not always directly applicable. Second, as was expected for a new data model, the main research and development efforts have been pursued towards efficient storing and querying of RDF data [1, 11, 9, 3]. Moreover, compared with the amount of research on the analysis of relational queries, much less attention has been devoted to the analysis, manipulation and optimization of SPARQL queries.

In this paper we attempt to partially fill the aforementioned gap by presenting the *SPARQL Analysis and Manipulation (SPAM)* tool, which was created as a specific system to help the SPARQL developer in creating, manipulating and optimizing SPARQL queries.

Our system focuses on the *optional matching* feature of SPARQL which has also been the focus of most of the theoretical work regarding this language [7, 10, 2, 6]. The idea behind optional matching, realized in SPARQL by the OPT operator, is to allow information to be added if the information is available in the data source instead of just failing to give an answer whenever some part of the query does not match. Recent experimental works [8] show that the use of the OPT operator in practice is substantial. Moreover, the importance of the OPT operator has also been recognized from a database theory point of view. It has been shown that the combined complexity of SPARQL query evaluation (i.e., checking if some set of variable bindings is a solution) raises from PTIME-membership for queries not using OPT, to PSPACE-completeness when OPT is considered [7, 10]. As a way of coping with this high complexity, the class of *well-designed* SPARQL *graph patterns* was introduced as a fundamental fragment of OPT queries with good behavior for query evaluation [7]. Well-designed graph patterns are obtained by forbidding some unnatural forms of interaction between optional and non-optional parts in a query by imposing simple syntactic restrictions. It was shown that the complexity of the evaluation problem for the well-designed fragment is coNP-complete [7]. Moreover, in [7, 2] the authors made the case that non-well-designed SPARQL graph patterns tend to be unnatural, as they contradict the intended meaning of the OPT operator.

Our tool is heavily based on previous theoretical work on SPARQL optimization [7, 10, 6], and more specifically on query rewriting, static analysis, containment and equivalence results provided in [6]. In [6] we introduced a tree representation of SPARQL queries, called *pattern trees*, that captures the well-designed fragment of the language. These pattern trees are reminiscent of relational query plans and were introduced as a first step towards an algebra for logical optimization of SPARQL queries, very much like the relational algebra used in relational database systems to construct and manipulate query plans. Moreover, in [6] we proposed several transformation rules that work on the structural level of pattern trees, and have the potential of minimizing the number of OPT operators used in SPARQL queries (for details of the pattern-tree representation and rule applications, see Section 2).

Among other functionalities, in our system the user can select or edit a SPARQL query, check if the query is well-designed, transform it into the pattern-tree representation, and choose among several applications of transformation rules available. These rules are used to manipulate the structure of the pattern tree, which can later be transformed back into a SPARQL query in the standard syntax. The tool also allows the user to execute the pattern tree (query) against sample RDF data previously provided.

Another important functionality of our tool is the test for containment and equivalence of SPARQL queries. In our previous work [6], we provided containment and equivalence checks for SPARQL queries that use the OPT operator. As in [6], for the *containment* of queries we consider the *subsumption* relation: one answer to a query subsumes another answer if the former extends the latter with more variable bindings. It has been argued that subsumption is a meaningful way of comparing the result of SPARQL queries containing the OPT operator [2]. We incorporate the subsumption check in our tool. Given two queries, our tool can decide whether one query is subsumed by the other, and if not, it provides a counterexample that can be further analyzed by the SPARQL developer. As shown in [6], the subsumption test allows us to also provide an equivalence test for SPARQL, which we also incorporate in our tool.

The rest of the paper is organized as follows. In Section 2 we provide more details on the theoretical background that supports our tool. Section 3 briefly describes the architecture of our system. Section 4 provides details of the demonstration scenario from the point of view of a SPARQL query developer. Some conclusions are provided in Section 5.

2. STATIC ANALYSIS AND OPTIMIZATION

Let us briefly recall the basics of RDF and SPARQL (for details on the formalization of RDF and SPARQL we refer the reader to [7, 2]). The data model underlying RDF data is a (directed, arc-labeled) graph. An RDF graph is composed of *RDF triples* which are tuples of the form (s, p, o) where s, p, and o are URIs. The basic constructor of a SPARQL query is the *triple pattern* which is an RDF triple that can have variables, usually named by using a question mark as prefix, as with $?X$. The core of the SPARQL language is the *graph pattern matching* facility which is an expression composed of triple patterns plus some operators. Our tool supports SPARQL graph patterns that use conjunctions of triple patterns, realized by the AND operator, plus optional parts realized by the OPT operator. A graph pattern P is *well-designed* if for every subpattern $P' = (P_1 \text{ OPT } P_2)$ of P it holds that every variable $?X$ occurring inside P_2 and outside P' also occurs inside P_1 [7].

The evaluation of a SPARQL graph pattern results in a set of *variable bindings*, obtained by matching the graph pattern against an RDF graph. Because of the optionality feature, some variables might be *unbound* after query evaluation [7].

Pattern trees and optimization rules

In [6] we introduced a tree representation of SPARQL graph patterns. The tree representation has as nodes conjunctions of triple patterns, called basic graph patterns (BGPs), with the structure of the tree representing the optional parts of the queries (OPT operators). The following example shows the tree representation in action.

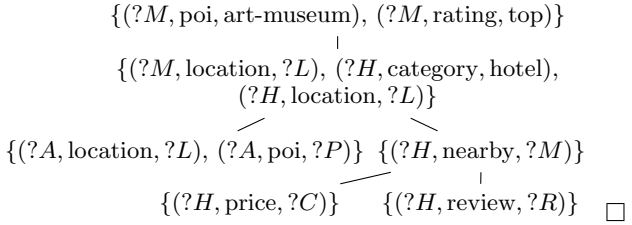
EXAMPLE 2.1. Consider a travel website storing information on hotels and points of interest (poi). Imagine one is interested in information helping some art-fan to plan her next holidays. Then, first of all, one is interested in a list of the top art-museums and could start with a graph pattern

$$P_1: \ (?M, \text{poi}, \text{art-museum}) \ \text{AND} \ (?M, \text{rating}, \text{top})$$

Next, one is obviously interested in information on hotels located in the same city as the museums, e.g. expressed by

$$P_2: \ (?M, \text{location}, ?L) \ \text{AND} \ (?H, \text{location}, ?L) \ \text{AND} \ (?H, \text{category}, \text{hotel})$$

However, one does not want the list of museums be reduced to those museums where there is also information about hotels in the same city available. The resulting query looks like $(P_1 \text{ OPT } P_2)$, that is, we are optionally retrieving data about hotels if the information is present. Next, even if the main goal of the trip is to see the museum, also a list of further points of interest located in the same city is obviously desirable. Hence consider P_3: $(?A, \text{location}, ?L)$ AND $(?A, \text{poi}, ?P)$. Again, this is additional information which would be nice to have, but if it is not available the previous results should be returned nevertheless. Hence, the resulting query looks like $((P_1 \text{ OPT } P_2) \text{ OPT } P_3)$. In addition, for all hotels that are actually located close to the museum, the query should return the price of a room, and available reviews of other guests of the hotel also as optional information. According to the formalization in [6], the whole graph pattern can be represented as the pattern tree shown below.



□

In [6] we introduced a top-down evaluation for pattern trees and showed that for every well-designed SPARQL graph pattern the evaluation of the original query coincides with the top-down evaluation of its corresponding pattern tree. We further showed that several transformation rules can be applied to pattern trees in order to minimize the number of nodes, and thus, minimizing the number of OPT operators in the query. We name rules as R1, R2, and R3, with the following intuitive description:

R1 If a triple pattern t belongs to node n and to a descendant n' of n, then delete t from n'.

R2 If node n does not introduce any new variable w.r.t. its ancestors, then *push* copies of n into its children.

R3 If there is a *homomorphism* from node n to the branch from the root to n, then merge node n with its parent.

EXAMPLE 2.2. While the query in Example 2.1 is a natural translation of the user intention into a SPARQL pattern, it has some potential for optimization. First, by the homomorphism h defined as $h(?A) = ?M$, $h(?L) = ?L$ and $h(?P)$ = 'art-museum' we can apply rule R3 to the node with BGP $\{(?A, \text{location}, ?L), (?A, \text{poi}, ?P)\}$. Further, the node $\{(?H, \text{nearby}, ?M)\}$ does not introduce any new variables, which allows us to apply R2. Hence the above pattern tree can be reduced to the much smaller pattern tree:

$$\{(?M, \text{poi}, \text{art-museum}),\ (?M, \text{rating}, \text{top})\}$$
$$|$$
$$\{(?M, \text{location}, ?L),\ (?H, \text{category}, \text{hotel}),$$
$$(?H, \text{location}, ?L),\ (?A, \text{location}, ?L),\ (?A, \text{poi}, ?P)\}$$
$$\swarrow \qquad \searrow$$
$$\{(?H, \text{nearby}, ?M),\ (?H, \text{price}, ?C)\} \qquad \{(?H, \text{nearby}, ?M),\ (?H, \text{review}, ?R)\}$$

□

Subsumption and equivalence

In [6] we use the tree representation to provide containment and equivalence tests for SPARQL graph patterns. For the containment of queries we consider the *subsumption* relation [2]. As mentioned above, solutions for queries containing the OPT operator may bind only a subset of the variables in the query, and it has thus been argued that subsumption is a meaningful way to compare SPARQL queries [2]: a solution μ_1 subsumes another solution μ_2, if μ_1 *extends* μ_2 with more variable bindings. More generally, a SPARQL query T_1 subsumes another SPARQL query T_2 if, for every RDF graph G, every solution of T_2 is subsumed by some solution of T_1. In [6] we provided a subsumption test for SPARQL pattern trees based on homomorphism checks.

3. SYSTEM OVERVIEW

The system is written in Java in a 3-tier design (Figure 1). The bottom layer consists of the Jena framework and the ARQ query engine (http://jena.sourceforge.net/ARQ), both used as a black box for parsing queries and RDF graphs. Our system currently uses the native Jena functionalities to load RDF data, but the Jena framework allows to use any relational engine as backend. On top of this layer, we have built the main functionalities, namely, building the pattern tree, checking for rule applicability, testing for equivalence and subsumption, and evaluating pattern trees over a given RDF dataset. The core functionality is also supported by the RDF engine. For example, homomorphisms between nodes are found by evaluating special queries constructed by our system. A GUI to provide access to these functionalities has been implemented using the Swing toolkit. The main functionalities and the interaction between the user and the GUI are described in the following section.

4. DEMONSTRATION SCENARIO

We will showcase the SPAM tool by imitating the work session of a SPARQL developer. The tool provides an interactive graphical user interface and offers the user several

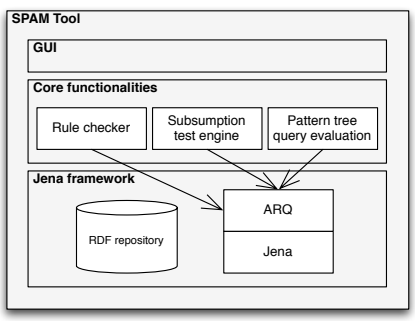


Figure 1: System architecture

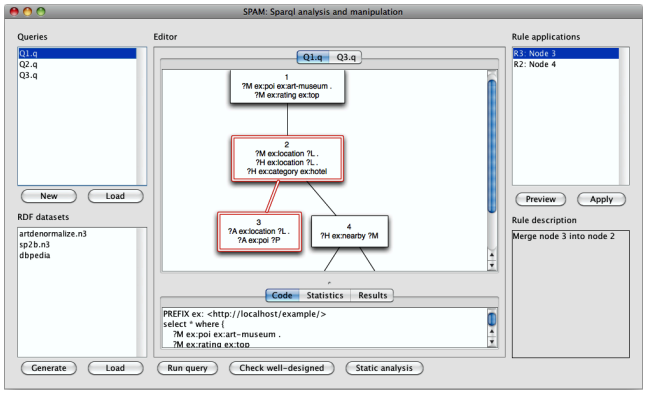


Figure 2: The main screen of the SPAM tool, showing the list of currently opened queries and databases at the left, the pattern tree and textual representation of the selected query in the middle, and the list of possible rule applications at the right.

functionalities for analyzing and manipulating a SPARQL graph pattern. Below we use the term "SPARQL query" (or simply "query") to refer to a "SPARQL graph pattern".

The SPAM tool allows the user to enter a SPARQL query in two ways: either input or load a query in standard SPARQL syntax or use the graphical editor to develop a new SPARQL query. In either case, the SPARQL query is presented to the user in the tree representation described in Section 2. The user may choose to edit the basic graph pattern (BGP) at any node in this tree representation, add new nodes or delete existing nodes. The user can trigger the SPAM tool to check if the current SPARQL query is well-designed. In case of a negative result, the tool collects all violations of the well-designedness condition. The corresponding paths in the tree representation are highlighted. At any time, the user may save the current SPARQL query or directly run it against some RDF graph that can be specified to the SPAM tool. The screen for the session with the SPARQL query from Example 2.1 is shown in Figure 2.

We next illustrate in this system demonstration how the user can apply the static analysis functionality described in Section 2 to optimize a SPARQL query: above all, the user may request the SPAM tool to try to eliminate OPT-operators via either rule R2 or R3 from [6]. Recall the SPARQL query from Example 2.2 where both rules were applicable. In such a case, the SPAM tool indicates which OPT-operators can be eliminated. The user has to confirm for each possible OPT-elimination that this rule application should indeed be carried out. This setting is depicted in Figure 2, where the two nodes merged by the rule selected by the user (at the top-right of the screen) are highlighted. Recall that we also introduced a rule R1 in [6] to delete triples from some BGP if the same triple already occurs in the BGP of some ancestor node. As explained in [6], this deletion may lead to a violation of the well-designedness condition. However, by assuming the top down evaluation of pattern trees [6], this has no effect on the query result. Since the SPAM tool ultimately stores well-designed queries in proper SPARQL syntax, it does not actually delete these redundant triples but only highlights such triples as an item of information to the user. When the query evaluation is done

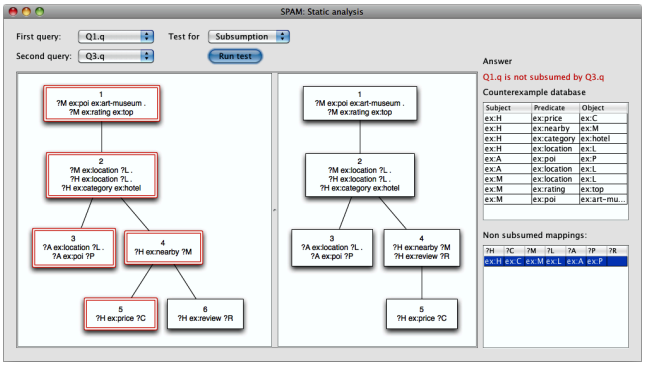


Figure 3: Screen showing the result of a subsumption test: since one query is not subsumed by the other, a counterexample is provided.

by the SPAM tool itself, such redundant triples are safely excluded from the evaluation process, since the SPAM tool implements the top-down evaluation shown in [6].

The query evaluation engine of the SPAM tool can also be used for a dynamic analysis of the current SPARQL query. By running a query against a given RDF graph, the SPAM tool may gather statistical information that is displayed to the user. This statistical information comprises, for instance, the total number of variable bindings computed at each node in the tree representation and the total CPU time needed for evaluating the BGP at each node. This statistical data may help the SPARQL developer in identifying performance bottlenecks of a SPARQL query. In case the user has no particular RDF graph in mind when developing a SPARQL query, the SPAM tool also offers the functionality of connecting to a benchmark tool to automatically generate a sample RDF graph. In this demonstration, we shall use the RDF data generator of the SP^{2}Bench [9] for this purpose. At any point in the session with the SPAM tool, the current query can be altered according to the feedback received from the tool and the same functionality (like static and dynamic analysis) can be applied again.

Finally, we demonstrate how the SPAM tool can be used to compare two SPARQL queries. The SPAM tool supports the test of both properties, subsumption and equivalence. The two queries to be tested for equivalence or subsumption may be entered to the SPAM tool as described above in the first step of our demonstration, i.e., either read in from an external file or edited with the SPAM tool. In our demonstration, we shall illustrate typical use cases of both tests. A typical use case for the equivalence test arises if the two queries result from the same SPAM session. In our demonstration, we shall carry out some manual changes to a given complex SPARQL query and then check if the resulting query is still equivalent to the original one. If this is not the case, then the SPAM tool outputs a "counterexample", i.e., a sample database together with a mapping that is a solution of one query but not a solution of the other query.

We shall also demonstrate the subsumption test on a typical use case, which arises when an additional data source becomes available. A typical goal of the SPARQL developer is then to extend a query Q_1 to a query Q_2 in such a way that it optionally retrieves the additional information thus provided. This extension should be able to find new variable bindings but, of course, it should not "lose" any old ones. This is captured by the subsumption property described in Section 2. Analogously to the equivalence test, the SPAM tool outputs a "counterexample" if subsumption is violated, i.e., a sample database together with a mapping that is a solution of query Q_1 but cannot be extended to a solution of query Q_2. Figure 3 shows the result of such a (negative) subsumption test between the query from Example 2.1 and a slightly modified version of this query.

5. CONCLUSION

In this system demo, we showcase the SPARQL Analysis and Manipulation (SPAM) tool, which assists the user in various aspects of the development of SPARQL queries – building upon the theoretical foundations of static analysis and optimization of SPARQL laid in [6]. Two main lines of extensions of the SPAM tool are on top of our agenda for future work. On the one hand, we want to integrate additional functionality into the tool such as further hints to the user for improving a given SPARQL query, e.g., hints concerning the creation of indexes to speed up the evaluation of the basic graph patterns at problematical nodes of the tree representation of a well-designed SPARQL graph pattern. On the other hand, we are planning to extend the theoretical underpinning of the SPAM tool. Above all, we thus aim at an extension of the algebra and algorithms from well-designed SPARQL patterns to larger fragments of SPARQL – allowing, e.g., filters, the union-operator, etc.

Acknowledgements

This work was funded in part by Marie Curie action IRSES under Grant No. 24761 (Net2), and by the Vienna Science and Technology Fund (WWTF) through project ICT08-032. Jorge Pérez was supported by Fondecyt grant 11110404 and by VID grant U-Inicia 11/04 Universidad de Chile.

6. REFERENCES

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[2] M. Arenas and J. Pérez. Querying semantic web data with SPARQL. In *PODS*, pages 305–316, 2011.

[3] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *SIGMOD*, pages 145–156, 2011.

[4] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang. Rewriting queries on SPARQL views. In *WWW*, pages 655–664, 2008.

[5] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for SPARQL. In *ICDE*, 2012. To appear.

[6] A. Letelier, J. Pérez, R. Pichler, and S. Skritek. Static analysis and optimization of semantic web queries. In *PODS*, pages 89–100, 2012.

[7] J. Pérez, M. Arenas, and C. Gutierrez.. Semantics and complexity of SPARQL. *ACM TODS*, 34(3), 2009.

[8] F. Picalausa and S. Vansummeren. What are real SPARQL queries like? In *SWIM*, page 7, 2011.

[9] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^{2}Bench: A SPARQL performance benchmark. In *ICDE*, pages 222–233, 2009.

[10] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *ICDT*, pages 4–33, 2010.

[11] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.