Castor: Using Constraint Programming to Solve SPARQL Queries

Vianney le Clément de Saint-Marcq[1,2], Yves Deville[1], Christine Solnon[2], and Pierre-Antoine Champin[2]

[1] Université catholique de Louvain, ICTEAM institute,
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve (Belgium)
{vianney.leclement,yves.deville}@uclouvain.be
[2] Université de Lyon, Université Lyon 1, LIRIS, CNRS UMR5205,
69622 Villeurbanne (France)
{christine.solnon,pierre-antoine.champin}@liris.cnrs.fr

As the amount of available data continues to grow in the semantic web, so does the need for efficient query engines. SPARQL [3] is a query language for RDF graphs standardized by the W3C. It is implemented by many triple stores like Sesame, 4store, Virtuoso, etc. However, they are still orders of magnitude slower than traditional relational databases. A key challenge is that SPARQL queries are known to be NP-hard [2].

The execution model of current SPARQL engines is based on relational algebra. A query is subdivided in many small parts that are computed separately. The answer sets are then joined together. User-specified filters — which allow the users to add constraints to be satisfied by answer sets — are often processed after such join operations. Constraint Programming (CP), on the other hand, is able to actively exploit constraints during the search, thus reducing the search space. Such techniques have been used effectively on various NP-hard problems, e.g., graph matching problems that are closely related to SPARQL pattern matching [1].

We propose a new engine, Castor, which is a lightweight CP solver dedicated to solving SPARQL queries. This work has been presented at the CP community [4]. In this paper, we first briefly present the Constraint Programming framework. Then we show how SPARQL queries can be modeled in CP.

1 Constraint Programming

A Constraint Satisfaction Problem (CSP) is a triple (X, D, C) where X is a set of variables, D is a function mapping each variable x to a domain $D(x)$, i.e., the finite set of values that can be assigned to x, and C is a set of constraints. A solution to a CSP is an assignment of the variables X satisfying the set of constraints C.

Various techniques exist to solve CSPs. The one we are interested in for this work is a complete tree-based search. It is based on a divide-and-conquer strategy. The root node of the tree is the initial CSP. The children of a node are *smaller* CSPs such that the union of the solution sets of the children is equal to the solution set of the parent. Leaf nodes are CSPs whose solution set can be computed trivially, e.g., the domain of every variable is a singleton. The tree is usually explored with a depth-first search.

The generation of smaller CSPs is driven by the *search strategy*. The labeling strategy for example, chooses one variable x whose domain has at least two values. For each

value $v \in D(x)$, we create a new CSP (X, D', C) where $D'(x) = \{v\}$ and $D'(y) = D(y)$ for every $y \neq x$. Once we obtain a CSP where every domain is a singleton, the solution set is either that single assignment if it satisfies the constraints, or the empty set.

A key concept of CP is to use *propagators* to reduce the search space. A propagator is an algorithm associated with a constraint. It removes from the domains of the variables the values that are known to be inconsistent with the constraint. Propagators are used on every node of the search tree. If one domain becomes empty, the CSP has no solution and can be pruned from the search tree.

2 CSP Model of SPARQL Queries

A SPARQL query consists of two parts: a graph pattern and solution modifiers. The graph pattern describes the subgraph we are looking for in the dataset. The solution modifiers specify what to do with the found subgraphs. We will focus here on the matching of graph patterns. In this section, we model the SPARQL graph matching problem by means of CSPs.

Let P be a SPARQL graph pattern and G be an RDF graph, we denote $[\![P]\!]_G$ the set of solutions of matching the pattern P on graph G. A solution μ is an assignment of the variables of P to values (i.e., URIs, blank nodes and literals) occurring in G. Note that the assignment may be partial, leaving some variables of P unassigned. We also denote $\mu(P)$ the pattern obtained by replacing every occurrence of a variable assigned in μ by its value.

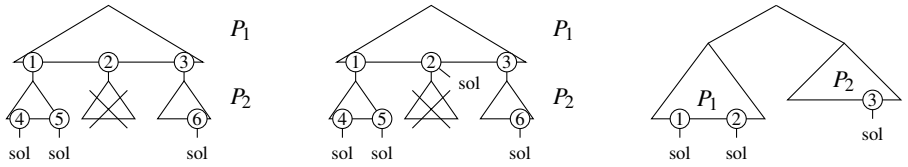
Basic Graph Patterns. A basic graph pattern (BGP) is a set of triple patterns, i.e., triples where subject, predicate and/or object may be a variable. Such a BGP can be directly translated to a CSP. The solutions of matching the BGP on an input graph G are the solutions of the CSP (X, D, C) such that

- $X = \text{vars}(BGP)$, the variables appearing in the BGP,
- all variables have the same domain, containing all URIs, blank nodes and literals of G,
- constraints ensure that every triple pattern of the BGP, once their variables are replaced by their value, belongs to the graph,
  i.e., $C = \left\{ \text{Member}\big((s,p,o), G\big) \;\middle|\; (s,p,o) \in BGP \right\}$, where Member is the set membership constraint.

Compound patterns. More advanced patterns, e.g., patterns with optional parts, cannot directly be translated into CSPs. Such patterns can produce solutions that are partial assignments, which are not allowed in the CSP framework. We can however compose the solution sets of subpatterns together such that it translates well to the complete tree search (see [4] for details).

Concatenation, optional and union patterns are respectively computed as follows and are depicted in Figure 1.

$$[\![P_1 \,.\, P_2]\!]_G = \{\, \mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1]\!]_G, \mu_2 \in [\![\mu_1(P_2)]\!]_G \,\}$$
$$[\![P_1 \,\texttt{OPTIONAL}\, P_2]\!]_G = [\![P_1 \,.\, P_2]\!]_G \cup \{\, \mu \in [\![P_1]\!]_G \mid [\![\mu(P_2)]\!]_G = \varnothing \,\}$$
$$[\![P_1 \,\texttt{UNION}\, P_2]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$$



(a) $[\![P_1 \,.\, P_2]\!]_G$ (b) $[\![P_1 \,\texttt{OPTIONAL}\, P_2]\!]_G$ (c) $[\![P_1 \,\texttt{UNION}\, P_2]\!]_G$

Fig. 1. Compound patterns are solved by composing the search trees of their subpatterns. Such search trees are represented by triangles. Circles at the bottom of a triangle are the solutions of the subpattern. Only solutions labeled "sol" are solutions of the compound pattern.

The concatenation operator computes the cartesian product of the solution sets of its subpatterns. We obtain such cartesian product by merging every pair of solutions assigning the same values to the common variables. Given a solution for the first subpattern (e.g., circle 1 in Fig. 1a), we compute the compatible solutions for the second subpattern by replacing the common variables by their value (e.g., circles 4 and 5 in Fig. 1a). If both subpatterns are basic graph patterns, we can compute the concatenation more efficiently by merging the sets of triple patterns and solving the CSP shown previously.

The optional operator adds to the set computed by the concatenation operator, the solutions of the first subpattern that could not be extended into a solution of the second subpattern. The union operator computes the solutions of its subpatterns separately.

Filters. The `FILTER` operator removes solutions of P that do not make the condition c true, i.e.,

$$[\![P \,\texttt{FILTER}\, c]\!]_G = \{\, \mu \in [\![P]\!]_G \mid c(\mu) \,\} \quad ,$$

where $c(\mu)$ is true if c is satisfied by μ. The SPARQL reference [3] defines the semantics of the constraints, also in the event of unbound variables.

The `FILTER` operator may be used a posteriori, to remove solutions which do not satisfy some constraints. This is usually done by existing SPARQL engines. However, such constraints may also be used during the search process in order to prune the search tree, which is an advantage of the CP technology.

When the `FILTER` operator is directly applied to a basic graph pattern *BGP*, the constraints may be simply added to the set of member constraints associated with the pattern, i.e., $[\![BGP \,\texttt{FILTER}\, c]\!]_G$ is equal to the set of solutions of the CSP $(X, D, C \cup \{c\})$, where (X, D, C) is the CSP associated with $[\![BGP]\!]_G$. Of course, finding all solutions to the CSP $(X, D, C \cup \{c\})$ is usually more quickly achieved than finding all solutions to (X, D, C) and then removing those which do not satisfy c. Filters applied on compound patterns can sometimes be *pushed* down onto subpatterns. Such query optimization is common in database engines.

3 Results and conclusion

To assess the feasibility and performances of our approach, we have developed Castor, a prototype SPARQL engine written in C++ embedding a dedicated CP solver. We have

compared Castor to Sesame using the SPARQL Performance Benchmark (SP^{2}Bench) [5]. The results are promising, competitive with Sesame and outperforming it on some queries (see Table 1).

Table 1. Speedup of Castor w.r.t. Sesame with in-memory store. The letter 'C' (resp. 'S' and '—') means only Castor (resp. Sesame and neither Castor nor Sesame) was able to solve the instance within the time limit of 30 minutes. Each row successively gives the number of triples in the dataset and the speedup of Castor w.r.t. Sesame on 8 difficult queries taken from [5].

	q2	q4	q5a	q5b	q6	q7	q8	q9
10k	6.75	2.95	94.15	6.51	5.13	1.21	61.52	3.60
50k	3.03	1.38	799.26	5.01	6.38	0.84	68.91	1.94
250k	1.54	0.41	C	3.93	C	1.24	39.32	1.34
1M	1.19	S	C	2.79	—	—	15.99	1.29
5M	1.19	—	C	2.00	—	—	3.81	1.24

Future work include performing more extensive evaluation Castor with different benchmarks and comparing to more engines, as well as implementing the full SPARQL specification. We will also experiment with specialized propagators to speed up filtering. For now, Castor uses a SQLite backend to store the triples while still requiring a considerable amount of RAM for its internal structures. Two directions can be followed here: either make a full-in-memory store speeding up the BGP matching, or explore compromises between efficiency and the size of the internal structures.

Acknowledgments. The first author is supported as a Research Assistant by the Belgian FNRS (National Fund for Scientific Research). This research is also partially supported by the Interuniversity Attraction Poles Programme (Belgian State, Belgian Science Policy) and the FRFC project 2.4504.10 of the Belgian FNRS. This work was done in the context of project SATTIC (ANR grant Blanc07-1 184534).

References

1. Baget, J.F.: RDF entailment as a graph homomorphism. In: The Semantic Web ISWC 2005, LNCS, vol. 3729, pp. 82–96. Springer (2005)
2. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. 34, 16:1–16:45 (September 2009)
3. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF (January 2008), http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/
4. le Clément de Saint-Marcq, V., Deville, Y., Solnon, C.: An efficient light solver for querying the semantic web. In: 17th Int. Conf. on Principles and Practice of Constraint Programming (CP 2011). LNCS, vol. 6876. Springer (2011)
5. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^{2}Bench: A SPARQL performance benchmark. In: IEEE 25th Int. Conf. Data Engineering ICDE '09. pp. 222–233 (2009)