

An Efficient Light Solver for Querying the Semantic Web

Vianney le Clément de Saint-Marcq^{1,2}, Yves Deville¹, and Christine Solnon²

¹ ICTEAM Research Institute,
Université catholique de Louvain,
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@liris.cnrs.fr

Abstract. The Semantic Web aims at building cross-domain and distributed databases across the Internet. SPARQL is a standard query language for such databases. Evaluating such queries is however NP-hard. We model SPARQL queries in a declarative way, by means of CSPs. A CP operational semantics is proposed. It can be used for a direct implementation in existing CP solvers. To handle large databases, we introduce a specialized and efficient light solver, Castor. Benchmarks show the feasibility and efficiency of the approach.

1 Introduction

The Internet has become the privileged means of looking for information in everyday's life. While the information abundantly available on the Web is increasingly accessible for human users, computers still have trouble making sense out of it. Developers have to rely on fuzzy machine learning techniques [5] or site-specific APIs (e.g., Google APIs), or resort to writing a specialized parser that has to be updated on every site layout change.

The Semantic Web is an initiative of the World Wide Web Consortium (W3C) to enable sites to publish computer-readable data aside of the human-readable documents. Merging all published Semantic Web data results in one large global database. The global nature of the Semantic Web implies a much looser structure than traditional relational databases. A loose structure provides the needed flexibility to store unrelated data, but makes querying the database harder. SPARQL [16] is a query language for the Semantic Web that has been standardized by the W3C. Evaluating SPARQL queries is known to be NP-hard [15].

The execution model of current SPARQL engines (e.g., Sesame [4], 4store [10] or Virtuoso [7]) 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 are often processed after such join operations. Constraint Programming (CP), on the other hand, is able to exploit filters as constraints during the search. A constraint-based query engine is thus well suited for the Semantic Web.

Contributions. Our first contribution is a declarative model based on CSPs and an operational semantics based on CP for solving SPARQL queries. Existing CP solvers however are not designed to handle the huge domains linked with the Semantic Web datasets. The second contribution of this work is a specialized lightweight solver, called Castor, for executing SPARQL queries. On standard benchmarks, Castor is competitive with existing engines and improves on complex queries.

Outline. The next section explains how data is represented in the Semantic Web and how to query the data. Section 3 and 4 show respectively the declarative model and the operational semantics to solve queries. Section 5 presents our lightweight solver implementing the operational semantics. Section 6 evaluates the feasibility and efficiency of the approach through a standard benchmark.

2 The Semantic Web and the SPARQL Query Language

The Resource Description Framework (RDF) [11] allows one to model knowledge as a set of triples (subject, predicate, object). Such triples express relations, described by predicates, between subjects and objects. The three elements of a triple can be arbitrary resources identified by Uniform Resource Identifiers (URIs)¹. Objects may also be literal values, such as strings, numbers, dates or custom data. An RDF dataset can be represented by a labeled directed multigraph as shown in Fig. 1.

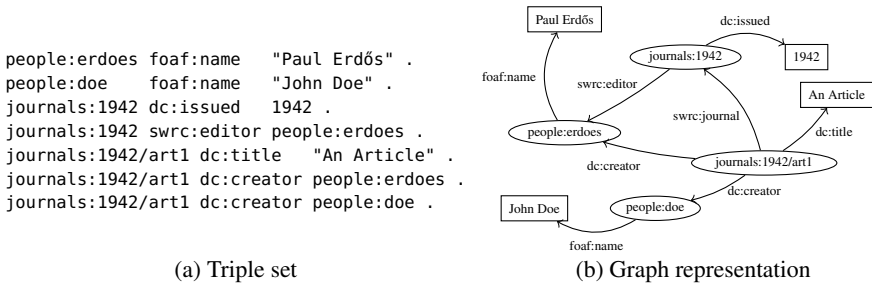


Fig. 1. Example RDF dataset representing a fictive journal edited by Paul Erdős and an article of the journal written by Erdős and Doe. Here, `people:erdoes` and `foaf:name` are URIs whereas `"Paul Erdős"` is a literal.

SPARQL [16] is a query language for RDF. A basic query is a set of triple patterns, i.e., triples where elements may be replaced by variables. Basic queries can be assembled in compound queries with composition, optional or alternative parts. Filters add

¹ More precisely, RDF makes use of URI References, but the differences are not relevant to this paper. The specification also allows subjects and objects to be blank nodes, i.e., resources without an identifier. Without loss of generality, blank nodes will be considered as regular URIs for the purpose of this paper.

constraints on the variables. A solution of a query is an assignment of variables to URIs or literals appearing in the dataset. Substituting the variables in the query by their assigned values in the solution gives a subset of the dataset. A SPARQL query may also define a subset of variables to return, a sort order, etc., but those are not relevant for this paper and are omitted.

More formally, let U , L and V be pairwise disjoint infinite sets representing URIs, literals, and variables, respectively. A SPARQL problem instance is defined by a pair (S, Q) such that $S \subset U \times U \times (U \cup L)$ is a finite set of triples corresponding to the dataset, and Q is a query. The syntax of queries is recursively defined as follows². The semantics will be defined in the next section.

- A basic query is a set of triple patterns (s, p, o) such that $s, p \in U \cup V$ and $o \in U \cup L \cup V$. The difference with RDF datasets is that we can have variables in place of URIs and literals.
- Let Q_1 and Q_2 be queries. $Q_1 \cdot Q_2$, Q_1 OPTIONAL Q_2 and Q_1 UNION Q_2 are compound queries.
- Let Q be a query and c be a constraint such that every variable of c occurs at least once in Q . Q FILTER c is a constrained query. The SPARQL constraint expression language used to define c includes arithmetic operators, boolean operators, comparisons, regular expressions for string literals and some RDF-specific operators.

Given a dataset S , we respectively denote U_S and L_S the set of URIs and literals that occur in S . Given a query Q , we denote $\text{vars}(Q)$ the set of variables appearing in Q .

3 A CSP Declarative Modeling of SPARQL Queries

A solution to a SPARQL problem instance (S, Q) is an assignment σ of variables of Q to values from $U_S \cup L_S$, i.e., a set of variable/value pairs. Given a solution σ and a query Q , we denote $\sigma(Q)$ the query obtained by replacing every occurrence of a variable assigned in σ by its value. The goal is to find all solutions. We denote $\text{sol}(S, Q)$ the set of all solutions to (S, Q) .

Contrarily to classical CSPs, a solution σ does not have to cover all the variables occurring in Q . For example, if a variable x appears only in an optional part that is not found in a solution σ , x will not appear in the solution σ . Such variables are said to be unbound.

In this section, we define the set of solutions of a SPARQL problem instance by means of CSPs, thus giving a denotational semantics of SPARQL queries. Note that, while doing so, we transform a declarative language, SPARQL, into another one based on CSPs which may be solved by existing solvers.

3.1 Basic Queries

A basic query BQ is a set of triple patterns (s, p, o) . In this simple form, an assignment σ is a solution if $\sigma(BQ) \subseteq S$.

² To keep things clear, we make some simplifications to the language. These assumptions do not alter the expressive power of SPARQL.

The SPARQL problem (S, BQ) may be viewed as a graph matching problem from a query graph associated with BQ to a target graph associated with S [3], as illustrated in Fig. 2. However, even the simple basic form of query is more general than classical graph matching, such as graph homomorphism or subgraph isomorphism. Variables on the edges (the predicates) can impose additional relationships between different edges. This problem is thus already NP-hard.

SELECT *

WHERE {

?p foaf:name ?name .

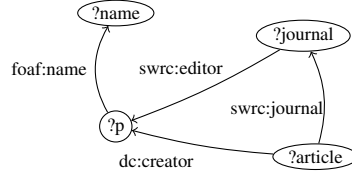
?journal swrc:editor ?p .

?article swrc:journal ?journal .

?article dc:creator ?p .

}

(a) SPARQL query



(b) Associated pattern graph

Fig. 2. Example of a basic query searching for journal editors having published an article in the same journal. Variables are prefixed by a question mark, e.g., ?name. Executing the query on the dataset of Fig. 1 results in the unique solution $\{(p, \text{people:erdoes}), (name, \text{"Paul Erdős"}), (journal, \text{journals:1942}), (article, \text{journals:1942/art1})\}$.

We formally define the set $\text{sol}(S, BQ)$ as the solutions of the CSP (X, D, C) such that

- $X = \text{vars}(BQ)$,
- all variables have the same domain, containing all URIs and literals of S , i.e., $\forall x \in X, D(x) = U_S \cup L_S$,
- constraints ensure that every triple of the query belongs to the dataset, i.e.,

$$C = \{ \text{Member}((s, p, o), S) \mid (s, p, o) \in BQ \} ,$$

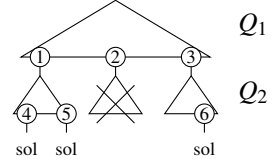
where Member is the set membership constraint.

3.2 Compound Queries

More advanced queries, e.g., queries with optional parts, cannot directly be translated into CSPs. Indeed some queries rely on the non-satisfiability of a subquery, which is coNP-hard. CSPs can only model NP problems.

$Q_1 \cdot Q_2$. Two patterns can be concatenated with the join or concatenation symbol $(\cdot)$. The solution set of a concatenation is the cartesian product of the solution sets of both queries. Such cartesian product is obtained by merging every pair of solutions assigning the same values to the common variables. Note that the operator is commutative, i.e., $Q_1 \cdot Q_2$ is equivalent to $Q_2 \cdot Q_1$. The set of solutions is defined as follows:

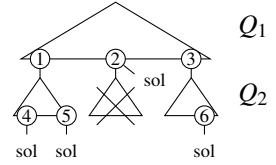
$$\begin{aligned} \text{sol}(S, Q_1 \cdot Q_2) = \\ \{ \sigma_1 \cup \sigma_2 \mid \sigma_1 \in \text{sol}(S, Q_1), \\ \sigma_2 \in \text{sol}(S, \sigma_1(Q_2)) \} . \end{aligned}$$



The figure on the right depicts an example. A triangle represents the search tree of a subquery. Circles at the bottom of a triangle are the solutions of the subquery. Circles 1, 2 and 3 represent $\text{sol}(S, Q_1)$. Solution 1 is extended into the solutions 4 and 5 in the search tree of $\text{sol}(S, \sigma_1(Q_2))$. Solutions 4, 5 and 6 are the solutions of the concatenation. If Q_1 and Q_2 are both basic queries, we can compute the concatenation more efficiently by merging both sets of triple patterns and solve the resulting basic query as shown in Section 3.1.

$Q_1 \text{ OPTIONAL } Q_2$. The OPTIONAL operator solves its left-hand side subquery Q_1 and *tries* to solve its right-hand side subquery Q_2 . If a solution of Q_1 cannot be extended into a solution of $Q_1 \cdot Q_2$, then that solution of Q_1 becomes a solution of the query too. More formally,

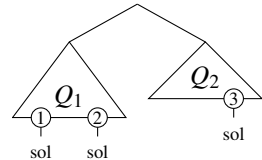
$$\begin{aligned} \text{sol}(S, Q_1 \text{ OPTIONAL } Q_2) = \\ \text{sol}(S, Q_1 \cdot Q_2) \cup \\ \{ \sigma \in \text{sol}(S, Q_1) \mid \text{sol}(S, \sigma(Q_2)) = \emptyset \} . \end{aligned}$$



Compared to the example for the concatenation operator, circle 2 in the figure becomes a solution of the compound query. The inconsistency check makes the search difficult. Indeed, in the simple case, Q_2 is a basic query and is thus solved by a CSP. However, as checking the consistency of a CSP is NP-hard, checking its inconsistency is coNP-hard. To ensure the semantics are compositional, we impose that if $Q_1 \text{ OPTIONAL } Q_2$ is a subquery of a query Q , then variables occurring in Q_2 but not in Q_1 ($\text{vars}(Q_2) \setminus \text{vars}(Q_1)$) do not appear elsewhere in Q . Such condition does not alter the expressive power of the language [1].

$Q_1 \text{ UNION } Q_2$. Disjunctions are introduced by the UNION operator. The solution set of the union of two queries is the union of the solution sets of both queries. The solutions of the two queries are computed separately:

$$\text{sol}(S, Q_1 \text{ UNION } Q_2) = \text{sol}(S, Q_1) \cup \text{sol}(S, Q_2) .$$



3.3 Filters

The FILTER operator removes solutions of Q not satisfying the constraint c , i.e.,

$$\text{sol}(S, Q \text{ FILTER } c) = \{ \sigma \in \text{sol}(S, Q) \mid c(\sigma) \} ,$$

where $c(\sigma)$ is true if c is satisfied by σ . The SPARQL reference [16] 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. A goal of this paper is to investigate the benefit of using CP, which actively exploits constraints to prune the search space, for solving SPARQL queries.

When the FILTER operator is directly applied to a basic query BQ , the constraints may be simply added to the set of member constraints associated with the query, i.e., $\text{sol}(S, BQ \text{ FILTER } c)$ is equal to the set of solutions of the CSP $(X, D, C \cup \{c\})$, where (X, D, C) is the CSP associated with (S, BQ) . 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 queries can sometimes be *pushed* down onto sub-queries [18]. For example $(Q_1 \text{ UNION } Q_2) \text{ FILTER } c$ can be rewritten as $(Q_1 \text{ FILTER } c) \text{ UNION } (Q_2 \text{ FILTER } c)$. Such query optimization is common in database engines.

4 A CP Operational Modeling of SPARQL Queries

The denotational semantics of SPARQL can be turned into an operational semantics using conventional CP solvers provided they allow posting constraints during the search. Examples of such solvers are Comet [6] or Gecode [8]. We detail the operational semantics of SPARQL queries, i.e., how the set $\text{sol}(S, Q)$ is computed. This model can be used for a direct implementation in existing solvers.

To run a query Q in a dataset S , we define a global array of CP variables $X = \text{vars}(Q)$. The initial domain of each variable $x \in X$ is $D(x) = U_S \cup L_S$. The set of constraints C is initially empty. To explain the posting of constraints and the search, we use Comet as a notation. The following code solves the query Q .

```
solveall<cp> {
} using {
  sol( $Q$ );
  output(); // print the solution
}
```

The first (empty) block posts the constraints, the second describes the search. The $\text{sol}(Q)$ function will be defined for every query type. It posts constraints and introduces choice points. Choice points are either explicit with the `try` keyword or implicit when labeling variables with `label`. When a failure is encountered, either explicitly with `cp.fail()` or implicitly during the propagation of a constraint, the search backtracks to the latest choice point and resumes the execution on the other branch. A back-track also occurs after outputting a solution at the end of the `using` block to search for other solutions. We assume a depth-first search expanding branches from left to right.

As we do not label all variables in every branch, the domain of some variables may still be untouched when outputting a solution. Such variables are considered unbound and are not included in the solution. Indeed, we always label all variables of a basic

query. Unbound variables do not appear in the basic queries along one branch, due to disjunctions introduced by UNION or inconsistent optional subqueries. No constraints are posted on such variables. Their domains are not reduced.

Figure 3a shows the `sol` function for a basic query with a filter. The filter is posted with the triples constraints and prunes the search tree from the beginning. In some cases, specific propagators can be used, e.g., for the comparison or arithmetic operators. In all cases we can fall back on an off-the-shelf SPARQL expression evaluator to propagate the condition with forward checking consistency, i.e., when all but one variables are assigned, propagation is realized on the domain of the uninstantiated variable.

Filters on compound queries however can only be checked after each solution of the subquery as shown in Fig. 3b. Note that the condition c is not posted as there may be unbound variables that need to be handled according to the SPARQL specification.

```
function sol( $BQ$ FILTER $c$ ) {
  forall(( $s, p, o$ ) in  $BQ$ )
    cp.post(Member(( $s, p, o$ ),  $S$ ));
  cp.post( $c$ );
  label(vars( $BQ$ ));
}
```

(a) Basic query with filter

```
function sol( $Q$ FILTER $c$ ) {
  sol( $Q$ );
  if( !  $c$  )
    cp.fail();
}
```

(b) Compound query with filter

Fig.3. Filters applied on basic queries are posted as constraints. In all other cases, they are checked after solving the subquery.

Concatenations are computed sequentially as shown in Fig. 4a. The OPTIONAL operator is similar to the concatenation and is shown in Fig. 4b. First, $\text{sol}(Q_1)$ is computed. Before executing the second subquery Q_2 , a choice point is introduced. The left branch computes $\text{sol}(Q_2)$, hence providing solutions to $Q_1 \cdot Q_2$. If it succeeds, the right branch is pruned. Otherwise, the right branch is empty and therefor $\text{sol}(Q_1)$ is returned as a solution. Note that this only works with depth first search exploring the left branch first. Finally, for the UNION operator the two subqueries are solved in two separate branches as shown in Fig. 4c.

It is clear that this operational semantics of SPARQL queries computes the set of solutions defined by the declarative modeling.

5 Castor: A Lightweight Solver for the Semantic Web

We now present Castor, a lightweight solver designed to compute SPARQL queries. A query does not involve many variables and constraints. The main challenge is to handle the huge domains associated with the variables. Existing CP solvers do not scale well in this context as shown in the experimental section. The key idea of Castor is to avoid maintaining and backtracking data structures that are proportional to the domain sizes. On the one hand we do not use advanced propagation techniques that need such

<pre> function sol(Q_1 . Q_2) { sol(Q_1); sol(Q_2); } </pre>	<pre> function sol(Q_1 OPTIONAL Q_2) { sol(Q_1); Boolean cons(false); try<cp> { sol(Q_2); cons := true; } { if(cons) cp.fail(); } } </pre>	<pre> function sol(Q_1 UNION Q_2) { try<cp> { sol(Q_1); } { sol(Q_2); } } </pre>
(a) Concatenation	(b) Optional	(c) Union

Fig. 4. Compound queries are solved recursively

expensive structures. On the other hand backtracking is a cheap operation allowing us to explore large trees fast enough to compensate for the loss of propagation.

In this section, we first present the database schema we use to store an RDF dataset. Then, we explain the three major components of the solver: the variables and the representation of their domains, the constraints and their propagators, and the search techniques used to explore the tree.

5.1 Database Schema

To run a query on a dataset, we need data structures to represent the dataset. We settled on an SQLite database. Such a relational database provides efficient lookups through the use of indexes. We use a standard schema designed for RDF applications [9]. It mainly consists of two tables.

- One table contains the set of all values occurring in the dataset, i.e., $U_S \cup L_S$. The values are numbered sequentially starting from 1.
- Another table contains the triples. The table has three columns containing only the identifier number of the value. Indexes are created on all column combinations to allow fast lookups in the table.

We only consider the value identifiers in the solver. We thus loose information about what the values represent. To get such information back quickly, e.g., for evaluating an expression, we load the table of values in memory before starting the search. We estimate a value to take on average 80 bytes. Large datasets contain around 10^8 values, taking 8 GB of memory. Having such amount of memory available is not uncommon in today's servers.

5.2 Variables and Domains Representation

Variables in Castor are integers taking values from 1 up to the number of values in the dataset. There is no direct relation between two numbers. As such, the ordering of the

values in the domain of a variable does not matter if bound consistency is not considered. We exploit this property in the data structures of the domains. When backtracking, we only need to restore the sizes of each domain. Such structures are also used in the code computing subgraph isomorphisms presented in [20].

We represent the finite domain $D(x)$ of a variable x by its size and two arrays `dom` and `map`. The size first values of `dom` are in the domain of the variable, the others have been removed (see Fig. 5). The `map` array maps values to their position in the `dom` array.

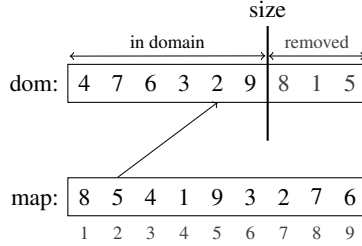


Fig. 5. Example representation of the domain $\{2, 3, 4, 6, 7, 9\}$, such that $\text{size} = 6$, when the initial domain is $\{1, \dots, 9\}$. The size first values in `dom` belong to the domain; the last values are those which have been removed. The `map` array maps values to their position in `dom`. For example, value 2 has index 5 in the `dom` array. In such representation, only the size needs to be restored on backtrack.

The following invariants are enforced.

- Arrays `dom` and `map` are coherent, i.e., $\text{map}[v] = i \Leftrightarrow \text{dom}[i] = v$.
- The domain $D(x)$ is the set of the first size values of `dom`, i.e., $D(x) = \{\text{dom}[i] \mid i \in \{1, \dots, \text{size}\}\}$.
- Any reduction of the domain does not modify the previously removed values (i.e., the values from $\text{size} + 1$ up to the end of the `dom` array).

The last invariant allows us to restore only size when backtracking. Indeed, the partition between removed values and values left in the domain will be the same. The order of the values before size may have changed however. The last invariant is respected when using depth-first search, since we keep removing values along one branch before backtracking.

The basic operations on the domain all have a constant time complexity. Checking if a value is still in the domain can be done with the property $v \in D(x) \Leftrightarrow \text{map}[v] \leq \text{size}$. To remove a value, we swap it with the latest value in the domain and decrease size . For example, to remove value 3 in Fig. 5, we swap the values 3 and 9 in `dom`, update `map` accordingly and decrease size by one.

To restrict the domain to a set of values, we *mark* each values to keep, i.e., we swap the value with the left-most non-marked value in `dom` and increase the count of marked values. We then set the domain size to the count of marked values. The complete operation has a linear time complexity w.r.t. the number of values kept.

5.3 Constraints and Propagators

There are two kinds of constraints in SPARQL queries: triple patterns and filters. Filters on compound queries are only checked after assigning all their variables. Filters on basic queries and triple patterns are posted and exploited during the search. As for domain representation, the goal is to minimize trailable structures that need to be backtracked. In the current prototype of Castor, no such structures exist for constraints.

A constraint in Castor is an object that implements two methods: *propagate* and *restore*. When the constraint is created, it registers to events of the variables. The *propagate* method is called when one of the registered events occurs. The *restore* method is called when the search backtracks. Currently, each variable has two events: *bind*, occurring when the domain becomes a singleton, and *change*, occurring when the domain has changed. To know which values have been removed from a variable since the last execution of the propagator, we store (locally to the constraint) the size of the domain at the end of the *propagate* method. Removed values are between the new and the old size in the *dom* array at the next call of the method. The *restore* method is used to reset the stored sizes after a backtrack. Propagators are called until the fix-point is reached.

Triple patterns. A triple pattern is a table constraint. It reacts on the *bind* event of the variables. When a variable is bound, we fetch all the consistent triples from the SQLite database and restrict the domains of the remaining unbound variables.

Filters. Checking filters on compound queries is done by an expression evaluator following SPARQL specifications. The evaluator considers all variables with a domain size larger than 1 as unbound. Filters on basic queries are posted together with the triple patterns. The propagator achieves forward checking consistency. As soon as all variables but one are bound, we iterate over the values in the domain of the unbound variable, keeping only values making the expression true.

Some filters can be propagated more efficiently with specialized algorithms. The propagator for $x \neq y$ waits for a value to be assigned to one of the two variables and removes it from the domain of the other variable. There is no need to iterate over all values in the domain. The constraint $x = y$ achieves arc consistency by removing from $D(y)$ the values that have been removed from $D(x)$ and vice versa, reacting to the change event.

5.4 Search

The search tree is defined by using a labeling strategy. At each node, a variable is chosen and a child node is created for each of the values in the domain of the variable. The standard smallest domain heuristic is used for choosing the variable. The order of the values is defined by their current order in the *dom* array representation.

The search tree is explored with a depth-first search algorithm. Such exploration is required for efficient backtracking of the domains (Section 5.2) and efficient inconsistency check of optional subqueries (Section 4).

To enable posting constraints during the search, we introduce *subtrees*. A subtree has a set of constraints and a set of variables to label. It iterates over all assignments of the variables satisfying the constraints, embedding the backtrack trail. At each assignment, Castor can create a new subtree or output the solution, depending on the query. When a subtree has been completely explored, the domains of the variables are restored to their state when the subtree was created and the constraints are removed. The search can then continue in the previous subtree.

6 Experimental Results

To assess the feasibility and the performances of our approach, we have run queries from the SPARQL Performance Benchmark (SP²Bench) [17]. SP²Bench consists of a deterministic program generating an RDF dataset of configurable size, and 12 representative queries. The datasets represent relationships between fictive academic papers and their authors, following the model of academic publications in the DBLP database. The benchmark includes both basic and compound queries, but only makes use of simple comparison filters. We removed unsupported solution modifiers like `DISTINCT` and `ORDER BY` from the queries. We focus on the queries identified as difficult by the SP²Bench authors (q4, q5, q6 and q7) as well as one simpler query (q2) and two queries involving the `UNION` operator (q8 and q9). We thus consider 8 queries as q5 comes in two flavors.

We compare the performances of three engines: the state-of-the-art SPARQL engine Sesame [4], the lightweight solver Castor described in Section 5 and a direct implementation of the operational semantics in Comet [6]. The Comet implementation loads the whole dataset in memory. It uses the built-in table constraint for the triple patterns and built-in expressions for the filters. Sesame was run both using an on-disk store and an in-memory store.

We have generated 6 datasets of 10k, 25k, 250k, 1M and 5M triples. We have performed three cold runs of each query over all the generated datasets, i.e., the engines were restarted and the databases cleared between two runs. Such setting corresponds to the one used by the authors of SP²Bench. All experiments were conducted on an Intel Pentium 4 2.40 GHz computer running Ubuntu Linux 10.10 with 2 GB of DDR-400 RAM and a 160 GB Maxtor 6Y160P0 ATA/133 disk. We report the time spent to solve the queries, not including the time needed to load the datasets. We checked that all engines find the exact same set of solutions.

Figure 6 shows the execution time of the considered queries. Note that both axes have logarithmic scales. We now discuss the results for each query.

Simple query. Query q2 has the form $BQ_1 \text{ OPTIONAL } BQ_2$. BQ_1 is a basic query with 9 variables and 9 triple patterns. The optional part BQ_2 has a single triple pattern with only one variable not appearing in BQ_1 . Executing subquery BQ_2 can thus be done by one access to the database. Sesame and Castor perform equally well. Comet however suffers from the heavy data structures.

Filters. Queries q4 and q5a are similar. Both are basic queries with one filter. Query q4 has 7 variables, 8 triple patterns and a filter $x_1 < x_2$ on two variables x_1 and x_2 . Query

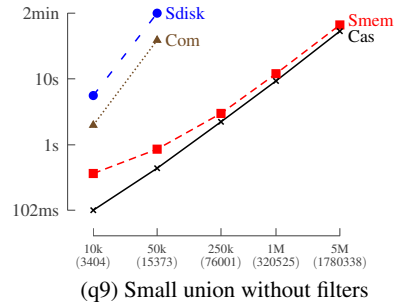
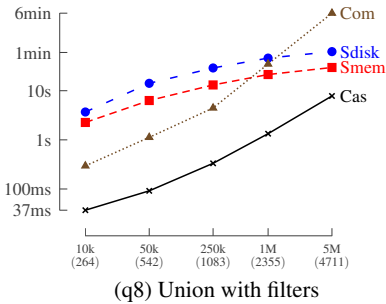
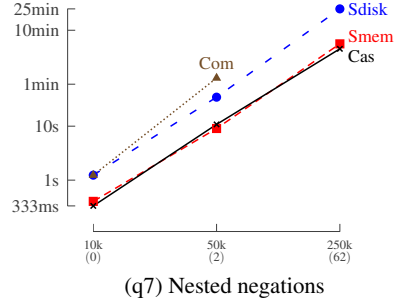
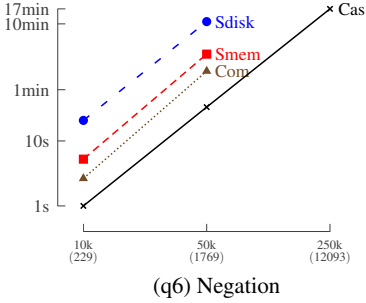
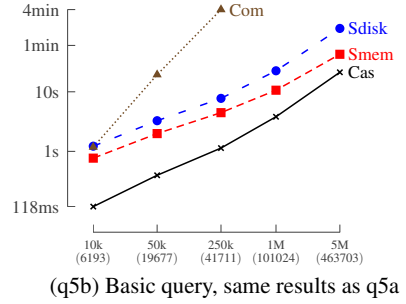
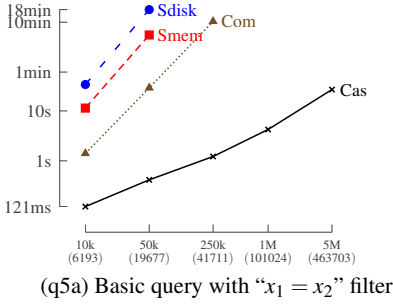
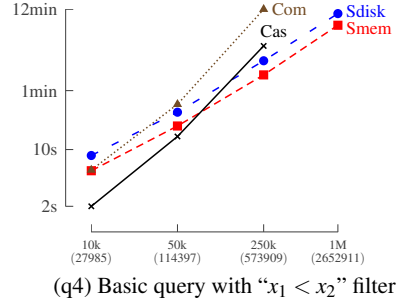
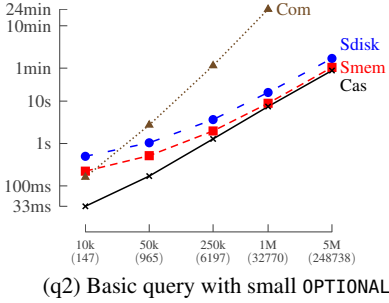


Fig. 6. Experimental results for Sesame with on-disk store (Sdisk), Sesame with in-memory store (Smem), Comet (Com) and Castor (Cas). The x-axis represents the dataset size in terms of number of triples. The y-axis is the query execution time. Both axes have a logarithmic scale. The number of solutions is written in parentheses.

q5a has 6 variables, 6 triple patterns and a filter $x_1 = x_2$. The CP engines are able to outperform Sesame on q5a thanks to their efficient propagation of the equality constraint. We suspect this constraint to be post-processed in Sesame. Query q4 however shows the opposite situation. It has many more solutions than q5a ($2.65 \cdot 10^6$ versus $1.01 \cdot 10^5$ for the dataset with 1M triples). As such, filtering is not the bottleneck anymore. Solving the query mostly involves pure database access.

The two flavors of q5, q5a and q5b, compute exactly the same set of solutions. The latter encodes the equality constraint into its 5 triple patterns using 4 variables. Unsurprisingly, the CP engines perform similarly on both queries as they exploit the filters early-on during the search. Sesame handles the filter-less query much better than q5a. This shows the relevance of our approach, especially considering filters are present in about half of the real-world queries [2].

Negations. A negation in SPARQL is a compound query that has the form $(Q_1 \text{ OPTIONAL } Q_2) \text{ FILTER } (!\text{bound}(x))$, where x is a variable appearing only in Q_2 . The filter removes all solutions assigning a value to x , i.e., we keep only solutions of Q_1 that cannot be extended into solutions of $Q_1 \cdot Q_2$. Query q6 is one such negation with additional filters inside Q_2 . Query q7 has no additional filters, but Q_2 is itself a nested negation. Counter-intuitively, q6 is actually more difficult than q7. Possible reasons are given in [17]. Castor has better results than Sesame for the former query and behaves similarly to Sesame on the latter.

Unions. The compound queries q8 and q9 use the UNION operator. The former adds inequality filters in both its subqueries. The subqueries of the latter contain only two triple patterns each. Yet, q9 generates many solutions. Neither Comet with its heavy structures nor Sesame with its on-disk store are able to go beyond 50k triples. Castor and Sesame with in-memory store are close to each other. In query q8, the two alternative subqueries have some duplicate triple patterns. Exploiting such property might explain the relative flatness of Sesame’s execution time compared to Castor.

Conclusion. Table 1 shows the relative speed of Castor w.r.t. Sesame using an in-memory store. The goal of Castor is to use CP to solve very constrained queries, i.e., queries where filters eliminate many solutions. Such queries (e.g., q5a and q6) are handled much more efficiently by Castor than by Sesame. On queries relying more on database access (e.g., q2 and q9), the CP approach is still competitive.

Table 1. Speedup of Castor w.r.t. Sesame with in-memory store. The letter ‘C’ (resp. ‘S’) means only Castor (resp. Sesame) was able to solve the instance within the time limit.

	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

7 Discussion

We proposed a declarative modeling and operational semantics for solving SPARQL queries using the Constraint Programming framework. We introduced a specialized lightweight solver implementing the semantics. We showed that the approach outperforms the state of the art on very constrained queries, and is competitive on most other queries.

Related work. Mamoulis and Stergiou have used CSPs to solve complex XPath queries over XML documents [13]. XML documents can be viewed as graphs, like RDF data³, but with an underlying tree structure. Such structure is used by the authors to design specific propagators. However, they cannot be used for SPARQL queries.

Mouhoub and Feng applied constraint programming to solve combinatorial queries in relational databases [14]. Such queries involve joining multiple tables subject to relatively complex arithmetic constraints. The problem is similar to SPARQL. However, the authors do not deal with large datasets. Their experiments are limited to tables with 800 rows. Such size is not realistic for RDF data.

Other work aims at extending the standard SQL query language to support explicit constraint satisfaction expressions [12,19]. This allows to solve CSPs within relational databases.

Future work. Two paths are possible. On the one hand, we can create a full-in-memory engine, getting rid of the SQLite database. More advanced propagators for the table constraint could then be used. While such engine would not scale well, it could still be of interest for very complex queries on small to medium-sized datasets. On the other hand, we can make a heavier database usage, eliminating the need to load all values in memory. The current propagators for triple patterns and equality constraints already do not need to know the meaning of a value. The query can also be preprocessed to reduce the initial domain before the variables are created to further reduce the memory consumption.

In both cases, specialized propagators need to be written for the various SPARQL expressions. Other consistency levels, e.g., bound consistency, may be considered for such tasks. Different variable selection heuristics can be investigated. More comprehensive benchmarks with other engines also needs to be done.

Acknowledgments. The authors want to thank the anonymous reviewers for their insightful comments. 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. Angles, R., Gutierrez, C.: The expressive power of SPARQL. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 114–129. Springer, Heidelberg (2008)

³ XML is in fact one of the syntaxes of RDF.

2. Arias, M., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An empirical study of real-world SPARQL queries. In: 1st International Workshop on Usage Analysis and the Web of Data (USEWOD 2011), in Conjunction with WWW 2011 (2011)
3. Baget, J.F.: RDF entailment as a graph homomorphism. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 82–96. Springer, Heidelberg (2005)
4. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF Schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002)
5. Cafarella, M.J., Halevy, A., Madhavan, J.: Structured data on the web. *Commun. ACM* 54, 72–79 (2011)
6. Dynamic Decision Technologies Inc.: Comet (2010), <http://www.dynadec.com>
7. Erling, O., Mikhailov, I.: RDF support in the Virtuoso DBMS. In: Networked Knowledge – Networked Media. SCI, vol. 221, pp. 7–24. Springer, Heidelberg (2009)
8. Gecode Team: Gecode: Generic constraint development environment (2006), <http://www.gecode.org>
9. Harris, S., Shadbolt, N.: SPARQL query processing with conventional relational database systems. In: Dean, M., Guo, Y., Jun, W., Kaschek, R., Krishnaswamy, S., Pan, Z., Sheng, Q.Z. (eds.) WISE 2005 Workshops. LNCS, vol. 3807, pp. 235–244. Springer, Heidelberg (2005)
10. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered RDF store. In: 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2009), at ISWC 2009 (2009)
11. Klyne, G., Carroll, J.J., McBride, B.: Resource description framework (RDF): Concepts and abstract syntax (2004), <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>
12. Lohfert, R., Lu, J., Zhao, D.: Solving SQL constraints by incremental translation to SAT. In: Nguyen, N.T., Borzowski, L., Grzech, A., Ali, M. (eds.) IEA/AIE 2008. LNCS (LNAI), vol. 5027, pp. 669–676. Springer, Heidelberg (2008)
13. Mamoulis, N., Stergiou, K.: Constraint satisfaction in semi-structured data graphs. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 393–407. Springer, Heidelberg (2004)
14. Mouhoub, M., Feng, C.: CSP techniques for solving combinatorial queries within relational databases. In: Nguyen, N.T., Szerbicki, E. (eds.) Intelligent Systems for Knowledge Management. SCI, vol. 252, pp. 131–151. Springer, Heidelberg (2009)
15. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 16:1–16:45 (2009)
16. Prud’hommeaux, E., Seaborne, A.: SPARQL query language for RDF (January 2008), <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>
17. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP²Bench: A SPARQL performance benchmark. In: Proc. IEEE 25th Int. Conf. Data Engineering, ICDE 2009, pp. 222–233 (2009)
18. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Proceedings of the 13th International Conference on Database Theory, ICDT 2010, pp. 4–33. ACM, New York (2010)
19. Siva, S., Wang, L.: A SQL database system for solving constraints. In: Proceeding of the 2nd PhD Workshop on Information and Knowledge Management, PIKM 2008, pp. 1–8. ACM, New York (2008)
20. Solnon, C.: Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence* 174(12–13), 850–864 (2010)