Castor: A Constraint-Based SPARQL Engine with Active Filter Processing

Vianney le Clément de Saint-Marcq[1,2,3], Yves Deville[1], Christine Solnon[2,4], and Pierre-Antoine Champin[2,3]

[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, LIRIS, CNRS UMR5205, 69622 Villeurbanne France
{christine.solnon,pierre-antoine.champin}@liris.cnrs.fr

[3] Université Lyon 1, F-69622 Villeurbanne France

[4] INSA Lyon, F-69622 Villeurbanne France

Abstract. Efficient evaluation of complex SPARQL queries is still an open research problem. State-of-the-art engines are based on relational database technologies. We approach the problem from the perspective of Constraint Programming (CP), a technology designed for solving NP-hard problems. Such technology allows us to exploit SPARQL filters early-on during the search instead of as a post-processing step. We propose Castor, a new SPARQL engine based on CP. Castor performs very competitively compared to state-of-the-art engines.

1 Introduction

As semantic web technologies adoption grows, the fields of application become broader, ranging from general facts from Wikipedia, to scientific publications metadata, government data, or biochemical interactions. The Resource Description Framework (RDF) [9] provides a standard knowledge representation model, a key component for interconnecting data from various sources. SPARQL [12] is the standard language for querying RDF data sources. Efficient evaluation of such queries is important for many applications.

State-of-the-art SPARQL engines (e.g., Sesame [5], Virtuoso [6] or 4store [7]) are based on relational database technologies. They are mostly designed for scalability, i.e., the ability to handle increasingly large datasets. However, they have difficulties to solve complex queries, even on small datasets.

We approach SPARQL queries from a different perspective. We propose Castor, a new SPARQL engine based on Constraint Programming (CP). CP is a technology for solving NP-hard problems. It has been shown to be efficient for graph matching problems [20,15], which are closely related to SPARQL [3]. Castor is very competitive with the state-of-the-art engines and outperforms them on complex queries.

Contributions. A first technical description of this work has been published in [13]. The present paper presents a number of enhancements of the first Castor prototype, namely:

– more efficient data structures based on a total ordering of RDF values,
– the translation of solution modifiers to the constraints framework,
– the replacement of the SQLite backend by native triple indexes based on the RDF-3x engine.

Finally, we have conducted more comprehensive benchmarks, now comparing Castor with Virtuoso and 4store.

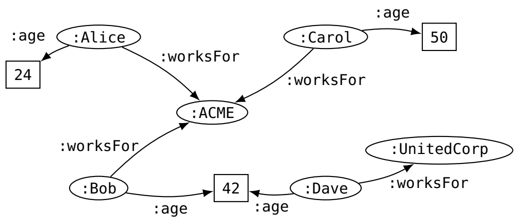
Outline. The next section describes the SPARQL language and how it is implemented in state-of-the-art engines. Section 3 presents our CP approach of SPARQL queries. Section 4 shows the major parts of our system. Section 5 contains the experimental results.

2 Background

Data in the semantic web are represented by a graph [9]. Nodes are identified by URIs[1] and literals, or they may be blank. Edges are directed and labeled by URIs. We will call such a graph an RDF dataset. Figure 1b shows an example of RDF dataset. Note that we can equivalently represent the dataset as a set of triples (Fig. 1a). Each triple describes an edge of the graph. The components of a triple are respectively the source node identifier (the subject), the edge label (the predicate) and the destination node identifier (the object).

```
:Alice :worksFor :ACME .
:Alice :age      24 .
:Bob   :worksFor :ACME .
:Bob   :age      42 .
:Carol :worksFor :ACME .
:Carol :age      50 .
:Dave  :worksFor :UnitedCorp .
:Dave  :age      42 .
```

(a) Triple set



(b) Graph representation

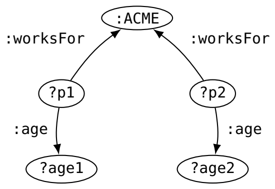
Fig. 1. RDF dataset example representing fictive employees

SPARQL [12] is the standard query language for RDF. The basis of a query is a triple pattern, i.e., a triple whose components may be variables. A set of triple patterns is called a basic graph pattern (BGP) as it can be represented by a pattern graph to be matched in the dataset. A solution of a BGP is an assignment of every variable to an RDF value, such that replacing the variables by their assigned values in the BGP yields a subset of the dataset viewed as a triple set. From now on, we will use the triple set representation of the dataset. More complex patterns can be obtained by composing BGPs together and by adding filters. Figure 2 shows an example SPARQL query with one BGP and one filter.

[1] For the sake of readability, throughout the paper we abbreviate URIs to CURIEs (http://www.w3.org/TR/curie/).

```
SELECT * WHERE {
  ?p1 :worksFor :ACME .   (P1)
  ?p1 :age      ?age1 .   (P2)
  ?p2 :worksFor :ACME .   (P3)
  ?p2 :age      ?age2 .   (P4)
  FILTER(?age1 < ?age2)   (F)
}
```

(a) SPARQL query



(b) Associated pattern graph

Fig. 2. SPARQL query example on the dataset shown in Fig. 1. The query returns all pairs of employees working at ACME, the first one being younger than the second one.

Formally, let U, B, L and V be pairwise disjoint infinite sets representing URIs, blank nodes, literals and variables, respectively. An RDF dataset is a finite set of triples $G \subset (U \cup B) \times U \times (U \cup B \cup L)$. We respectively denote U_G, B_G and L_G the finite set of URIs, blank nodes and literals occurring in G.

A SPARQL query consists of two parts: a graph pattern and solution modifiers. The graph pattern is defined recursively as follows.

– A basic graph pattern is a set of triple patterns $P = BGP \subset (U \cup V) \times (U \cup V) \times (U \cup L \cup V)$. Without loss of generality, that definition forbids blank nodes from appearing in a graph pattern: blank nodes can be replaced by variables (at least as long as we do not use any SPARQL entailment regime). We denote V_P the set of variables in P.
– Let P be a graph pattern and c be a SPARQL expression such that every variable of c occurs in P.[2] P `FILTER` c is a constrained pattern. We denote V_c the set of variables in c.
– Let P_1 and P_2 be graph patterns. $P_1 \,.\, P_2$, P_1 `OPTIONAL` P_2 and P_1 `UNION` P_2 are compound patterns. We will ignore compound patterns as they are not relevant for the contributions of this paper. However, they are handled by Castor as discussed in [13].

A solution of a graph pattern P with respect to a dataset G is a mapping $\mu : V_P \rightarrow U_G \cup B_G \cup L_G$. We denote $[\![P]\!]_G$ the set of all solutions. Let $\mu(P)$ (resp. $\mu(c)$) be the pattern (resp. expression) obtained by replacing every occurrence of a variable ?x $\in V_P$ (resp. V_c) with its value $\mu(\text{?x})$. The solutions of a basic graph pattern BGP are

$$[\![BGP]\!]_G = \{\mu \mid \mu(BGP) \subseteq G\} \ .$$

The solutions of a graph pattern P constrained by an expression c are

$$[\![P\,\texttt{FILTER}\,c]\!]_G = \{\mu \in [\![P]\!]_G \mid \mu(c) \text{ evaluates to true}\} \ .$$

When evaluating a SPARQL query, the solution set of the graph pattern is transformed into a list. Solution modifiers are then applied in the following order.

[2] The condition on the variables appearing in c restricts the language to *safe* filters, without limiting its expressive power [2].

1. `ORDER BY` sorts the list of solutions,
2. `SELECT` projects the solution on a set of variables, i.e., the domain of the solution mappings are restricted to the specified set of variables,
3. `DISTINCT` removes duplicate solutions,
4. `OFFSET n` removes the *n* first solutions of the list,
5. `LIMIT n` keeps only the *n* first solutions of the list.

State-of-the-art SPARQL engines rely on relational database technologies to store the datasets and execute the queries. Such systems can be divided in three categories [8].

– *Triple stores* store the whole dataset in one three-column table. Each row represents one triple. Examples in this category are Sesame, 4store [7], Virtuoso [6], RDF-3x [10] and Hexastore [19].
– *Vertically partitioned tables* maintain one two-column table for each predicate. The resulting smaller tables are sometimes more convenient than the single large table of triple stores. However, there is a significant overhead when variables appear in place of predicates in the query. An example is SW-Store [1].
– *Schema-specific systems* map legacy relational databases to RDF triples using a user-specified ontology. Queries are translated to SQL and performed on the relational tables. Native RDF datasets can be transformed to relational tables if the user provides the structure. Thus, such systems do not handle well the schema-less nature of RDF.

For the purpose of this paper, we will focus on triple stores, which are very popular and well-performing generic engines.

The solutions for a single triple pattern can be retrieved efficiently from a triple store using redundant indexes. Combining multiple triple patterns however involves joining the solution sets together, i.e., merging mappings that assign the same value to common variables. Such operations can be more or less expensive depending on the order in which they are performed. Query engines carefully construct a join graph optimizing the join order. The join graph is then executed bottom-up, starting from the leaves, the triple patterns, and joining the results together. Filters are applied once all their variables appear in a solution set.

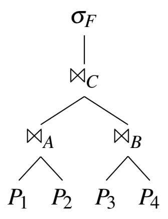
The join graph optimization problem has been largely studied for relational databases (e.g., [11,16]). Many results were also adapted to semantic web databases (e.g., [17]).

Figure 3 shows an example of executing a query in a triple store. Here, the filter can only be applied at the very last stage of the evaluation, as it involves variables from different parts of the query.

3 Constraint-Based View of SPARQL Queries

The relational database approach to SPARQL queries focuses on the triple patterns to build the solutions. We propose another view focusing on the variables.

A solution to a query is an assignment of the variables of the query to values of the dataset. The set of values that can be assigned to a variable is called its *domain*. The domain of a variable is initially the set of all URIs, blank nodes and literals occurring



(a) Join graph

$$\begin{aligned}
[\![P_1]\!] = [\![P_3]\!] &= \{\,(\texttt{:A}), (\texttt{:B}), (\texttt{:C})\,\} \\
[\![P_2]\!] = [\![P_4]\!] &= \{\,(\texttt{:A},\texttt{24}), (\texttt{:B},\texttt{42}), (\texttt{:C},\texttt{50}), (\texttt{:D},\texttt{42})\,\} \\
[\![\bowtie_A]\!] = [\![\bowtie_B]\!] &= \{\,(\texttt{:A},\texttt{24}), (\texttt{:B},\texttt{42}), (\texttt{:C},\texttt{50})\,\} \\
[\![\bowtie_C]\!] &= \{\,(\texttt{:A},\texttt{24},\texttt{:A},\texttt{24}), (\texttt{:A},\texttt{24},\texttt{:B},\texttt{42}), (\texttt{:A},\texttt{24},\texttt{:C},\texttt{50}), \\
&\quad (\texttt{:B},\texttt{42},\texttt{:A},\texttt{24}), (\texttt{:B},\texttt{42},\texttt{:B},\texttt{42}), (\texttt{:B},\texttt{42},\texttt{:C},\texttt{50}), \\
&\quad (\texttt{:C},\texttt{50},\texttt{:A},\texttt{24}), (\texttt{:C},\texttt{50},\texttt{:B},\texttt{42}), (\texttt{:C},\texttt{50},\texttt{:B},\texttt{42})\,\} \\
[\![\sigma_F]\!] &= \{\,(\texttt{:A},\texttt{24},\texttt{:B},\texttt{42}), (\texttt{:A},\texttt{24},\texttt{:C},\texttt{50}), (\texttt{:B},\texttt{42},\texttt{:C},\texttt{50})\,\}
\end{aligned}$$

(b) Bottom-up evaluation

Fig. 3. Executing the query from Fig. 2 in a triple store evaluates the join graph bottom-up. Note that, depending on the used join algorithms, some intermediate results may be produced lazily and need not be stored explicitly. The URIs of the employees are abbreviated by their first letter.

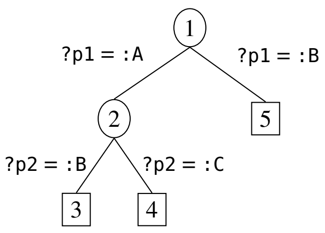
in the dataset. We construct solutions by selecting for each variable a value from its domain and checking that the obtained assignment satisfies the triple patterns and the filters (i.e., the *constraints*).

Constructing all solutions can be achieved by building a search tree. Each node contains the domains of the variables. The root node contains the initial domains. At each node of the tree, a variable is assigned to a value in its domain (i.e., its domain is reduced to a singleton), and constraints are propagated to reduce other variable domains. Whenever a domain becomes empty, the branch of the search tree is pruned. The form of the search tree thus depends on the choice of variable at each node and the order of the children (i.e., how the values are enumerated in the domain of the variables). Let us consider for example the query of Fig. 2. When assigning `?age1` to `42`, we can propagate the constraint `?age1 < ?age2` to remove from the domain of `?age2` every value which is not greater than 42 and, if all values are removed, we can prune this branch.

This is the key idea of constraint programming: prune the search tree by using the constraints to remove inconsistent values from the domains of the variables. Each constraint is used successively until the fix-point is reached. This process, called *propagation*, is repeated at every node of the tree. There are different levels of propagation. An algorithm with higher complexity will usually be able to prune more values. Thus, a trade-off has to be found between the achieved pruning and the time taken.

Figure 4 shows the search tree for the running example. At the root node, the triple patterns restrict the domains of `?p1` and `?p2` to only `:Alice`, `:Bob` and `:Carol`, i.e., the employees working at ACME, and `?age1` and `?age2` to $\{\texttt{24},\texttt{42},\texttt{50}\}$. The filter removes value `50` from `?age1`, as there is no one older than 50. Similarly, `24` is removed from `?age2`. Iterating the process, we can further remove `:Carol` from `?p1` and `:Alice` from `?p2`. Compared to the relational database approach, we are thus able to exploit the filters at the beginning of the search.

The tree is explored in a depth-first strategy. Hence only the path from the root to the current node is kept in memory. In most constraint programming systems, instead of keeping copies of the domains along the path to the root, one maintains the domains of the current node and a *trail*. The trail contains the minimal information needed to restore the current domains to any ancestor node.



(a) Search tree

Node	*D*(?p1)	*D*(?age1)	*D*(?p2)	*D*(?age2)
①	{:A,:B}	{24,42}	{:B,:C}	{42,50}
②	:A	24	{:B,:C}	{42,50}
$\boxed{3}$	:A	24	:B	42
$\boxed{4}$	:A	24	:C	50
$\boxed{5}$	:B	42	:C	50

(b) Domains

Fig. 4. Executing the query in Fig. 2 with constraint programming explores the search tree top-down. The triple patterns and filters are used at every node to reduce the domains of the variables. The URIs of the employees are abbreviated by their first letter.

4 Implementation

We evaluated the constraint-based approach using a state-of-the-art CP solver in [13]. While such implementation delivered some results, the cost of restoring the domains in generic solvers is too high for large datasets. Hence, we have built a specialized lightweight solver called Castor.

Castor is a prototype SPARQL engine based on CP techniques. When executing a query, a domain is created for every variable of the query, containing all values occurring in the dataset. For efficiency, every value is represented by an integer. Constraints correspond to the triple patterns, filters and solution modifiers. The associated pruning functions, called *propagators* are registered to the domains of the variables on which the constraints are stated. The propagators will then be called whenever the domains are modified. The search tree is explored in a depth-first strategy. A leaf node where every domain is a singleton is a solution, which is returned by the engine.

In this section, we describe the major components of Castor: the constraints and their propagators, the representation of the domains, the mapping of RDF values to integers, and the triple indexes used to store the dataset and propagating the triple pattern constraints.

4.1 Constraints

Constraints and propagators are the core of a CP solver. SPARQL queries have three kinds of constraints: triple patterns, filters and solution modifiers. The associated propagators can achieve different levels of consistency, depending on their complexity and properties of the constraint. We first show the different levels of consistency that are achieved by Castor. Then, we explain the propagators for the different constraints.

To be correct, a propagator should at least ensure that the constraint is satisfied once every variable in the constraint is bound (i.e., its domain is a singleton). However, to reduce the search space, propagators can prune the domains when variables are unbound. Propagators can be classified by their achieved level of consistency [4], i.e., the amount of pruning they can achieve.

– A propagator achieving forward-checking consistency does nothing until all variables in the constraint are bound, except one. It then iterates over the domain of the unbound variable, removing all values that do not satisfy the constraint. The required operation on the domains is the removal of a value.
– Bound consistency ensures that the bounds (i.e., the minimum and maximum value) of the domains of the variables in the constraint are consistent. A value is consistent if there exists a solution of the constraint with that value. The required operation on the domains is the update of a bound (i.e., increasing the lower bound or decreasing the upper bound).
– Domain consistency ensures that every value in the domains of the variables in the constraint are consistent. The required operation on the domains is the removal of a value. Domain consistency is the strongest level of consistency we consider. However, propagators achieving domain consistency usually have a higher complexity and could require maintaining auxiliary data structures.

Triple Patterns. A triple pattern is a constraint involving three variables, one for each component. For ease of reading, we consider constants to be variables whose domains are singletons. The pruning is performed by retrieving all the triples from the dataset where the components of the bound variables correspond to the assigned value. Values of domains of unbound variables that do not appear in the resulting set of triples are pruned. If the pruning is performed with only one unbound variable, we achieve forward-checking consistency. Castor achieves more pruning by performing the pruning when one or two variables are unbound. If all three variables are bound, the propagator checks if the triple is in the dataset and empties a domain if this is not the case.

Filters. Castor has a generic propagator for filters achieving forward-checking consistency. When traversing the domain of the unbound variable, we can check if the filter is satisfied by evaluating the SPARQL expression as described by the W3C recommendation [12]. It provides a fallback to easily handle any filter, but is not very efficient. When possible, specialized algorithms are preferred.

For example, the propagator for the `sameTERM(?x, ?y)` filter can easily achieve domain consistency. The constraint states that `?x` and `?y` are the same RDF term. The domains of both variables should be the same. Hence, when a value is removed from one domain, the propagator removes that value from the other domain.

Propagators for monotonic constraints [18], e.g., `?x < ?y`, can easily achieve bound consistency. Indeed, for constraint `?x < ?y`, we have $\max(\texttt{?x}) < \max(\texttt{?y})$ and $\min(\texttt{?x}) < \min(\texttt{?y})$. The pruning is performed by adjusting the upper bound of `?x` and the lower bound of `?y`.

Solution Modifiers. The `DISTINCT` keyword in SPARQL removes duplicates from the results. Such operation can also be handled by constraints. When a solution is found, a new constraint is added stating the any further solution must be different from the current one. The propagator achieves forward-checking consistency, i.e., when all variables but one are bound, we remove the value of the already found solution from the domain of the unbound variable.

When the `ORDER BY` and `LIMIT` keywords are used together, the results shall only include the n best solutions according to the specified ordering. After n solutions have been found, we add a new constraint stating that any new solution must be "better" than the worst solution so far. Such technique is known as branch-and-bound.

4.2 Mapping RDF Values to Integers

To avoid juggling with heavy data structures representing the RDF values, we map every value occurring in the dataset to a numerical identifier. Such mapping is also common in triple stores. The domains in Castor contain only those identifiers. An on-disk dictionary allows the retrieval of the associated value when needed.

Let $\mathrm{id}(v)$ be the identifier mapped to the RDF value v. To efficiently implement a bounds consistent propagator for the `<` filter, we want $v_1 <_F v_2 \Rightarrow \mathrm{id}(v_1) < \mathrm{id}(v_2)$, where $<_F$ is the `<` operator of SPARQL expressions. The SPARQL specification only defines $<_F$ between numerical values, between simple literals, between strings, between Boolean values, and between timestamps. The $<_F$ operator thus defines a partial order.

To efficiently implement the `ORDER BY` solution modifier, we also want $v_1 <_O v_2 \Rightarrow \mathrm{id}(v_1) < \mathrm{id}(v_2)$, where $<_O$ is the partial order defined in the SPARQL specification. This order introduces a precedence between blank nodes, URIs and literals. Literals are ordered with $<_F$. Hence, $v_1 <_F v_2 \Rightarrow v_1 <_O v_2$.

To map each RDF value to a unique identifier, we introduce a total order $<_T$ that is compatible with both partial orders, i.e.,

$$\forall (v_1, v_2) \in (U \cup B \cup L) \times (U \cup B \cup L), v_1 <_O v_2 \Rightarrow v_1 <_T v_2 \ .$$

Values are partitioned into the following classes, shown in ascending order. The ordering of the values inside each class is also given. When not specified, or to solve ambiguous cases, the values are ordered by their lexical form.

1. Blank nodes: ordered by their internal identifier
2. URIs
3. Plain literals without language tags
4. xsd:string literals
5. Boolean literals: first false, then true values
6. Numeric literals: ordered first by their numerical value, then by their type URI
7. Date/time literals: ordered chronologically
8. Plain literals with language tags: ordered first by language tag, then by lexical form
9. Other literals: ordered by their type URI

We map the values of a dataset to consecutive integers starting from 1, such that $v_1 <_T v_2 \Leftrightarrow \mathrm{id}(v_1) < \mathrm{id}(v_2)$.

4.3 Variables and Domains

A domain is associated with every variable, representing the set of values that can be assigned to the variable. During the search, the domain gets reduced and restored. The data structures representing the domain should perform such operations efficiently. There are two kinds of representations. The *discrete* representation keeps track of every single value in the domain. The *bounds* representation only keeps the lowest and highest value of the domain according to the total order defined in Section 4.2. We propose a dual view, leveraging the strengths of both representations.

Discrete Representation. The domain is represented by its size and two arrays `dom` and `map`. The `size` first elements 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.

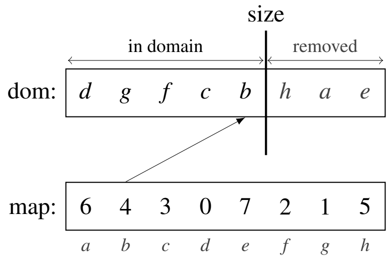


Fig. 5. Example representation of the domain $\{b,c,d,f,g\}$, such that `size` $= 5$, when the initial domain is $\{a,\ldots,h\}$. The `size` first values in `dom` belong to the domain; the last values are those which have been removed. The `map` array maps values to their position in `dom`. For example, value b has index 4 in the `dom` array. In such representation, only the size needs to be kept in the trail.

To remove a value, we swap it with the last value of the domain (i.e., the value directly to the left of the `size` marker), reduce `size` by one and update the `map` array. Such operation is done in constant time.

Alternatively, we can restrict the domain to the intersection of itself and a set S. We move all values of S which belong to the `size` first elements of `dom` at the beginning of `dom` and set `size` to the size of the intersection. Such operation is done in $O(|S|)$, with $|S|$ the size of S. Castor uses the restriction operation in propagators achieving forward-checking consistency.

Operations on the bounds however are inefficient. This major drawback is due to the unsorted `dom` array. Searching for the minimum or maximum value requires the traversal of the whole domain. Increasing the lower bound or decreasing the upper bound involves removing every value between the old and new bound one by one.

As the order of the removed values is not modified by any operation, the domain can be restored in constant time by setting the `size` marker back to its initial position. The trail, i.e., the data structure needed to restore the domain to any ancestor node of the search tree, is thus a stack of the sizes.

Note that this is not the standard representation of discrete domains in CP. However, the trail of standard representations is too heavy for our purpose and size of data.

Bounds Representation. The domain is represented by its bounds, i.e., its minimum and maximum values. In contrast to the discrete representation, the bound representation is an approximation of the exact domain. We assume all values between the bounds are present in the domain.

In such a representation, we cannot remove a value in the middle of the domain as we cannot represent a hole inside the bounds. However, increasing the lower bound or decreasing the upper bound is done in constant time.

The data structure for this representation being small (only two numbers), the trail contains copies of the whole data structure. Restoring the domains involves restoring both bounds.

Dual View. Propagators achieving forward-checking or domain consistency remove values from the domains. Thus, they require a discrete representation. However, propagators achieving bounds consistency only update the bounds of the domains. For them to be efficient, we need a bounds representation. Hence, Castor creates two variables x_D and x_B (resp. with discrete and bound representation) for every SPARQL variable ?x. Constraints are stated using only one of the two variables, depending on which representation is the most efficient for the associated propagator. In particular, monotonic constraints are stated on bounds variables whereas triple pattern constraints are stated on discrete variables.

An additional constraint $x_D = x_B$ ensures the correctness of the dual approach. Achieving domain consistency for this constraint is too costly, as it amounts to perform every operation on the bounds on the discrete representation. Instead, the propagator in Castor achieves forward-checking consistency, i.e., once one variable is bound the other will be bound to the same value. As an optimization, when restricting a domain to its intersection with a set S, we filter out values of S which are outside the bounds and update the bounds of x_B. Such optimization does not change the complexity of the operation, as it has to traverse the whole set S anyway.

4.4 Triple Indexes

The propagator of the triple pattern constraint needs to retrieve a subset of triples from the dataset, where some components have a specific value (see Section 4.1). The efficiency of such operation depends on the way the triples are stored on the disk. Castor makes use of indexes to retrieve the triples. An index sorts the triples in lexicographical order and provides efficient retrieval of triples with a fixed prefix. For example, the SPO index sorts the triples first by subject, then by predicate and last by object. It can be used to retrieve all triples with a specific subject or all triples with specific subject and predicate. It can also be used to check whether a triple is part of the dataset. Castor has three indexes: SPO, POS and OSP to cover all possible combinations.

The data structure underlying an index is based on the RDF-3x engine [10]. The sorted triples are compressed and packed in pages (i.e., a block of bytes of fixed size, in our case 16KB). Those pages are the leaves of a B+-tree. The tree allows one to find the leaf containing the first requested triple in logarithmic time. After the decompression, we can find the triple using a binary search algorithm.

Propagators are called multiple times at every node of the search tree. Thus, a set of triples can be requested many times. To reduce the overhead of decompressing the leaf pages containing the triples, we introduce a small least-recently-used cache of decompressed pages. The size of the cache is currently arbitrarily fixed to 100 pages.

5 Experimental Results

To assess the performances of our approach, we have run the SPARQL Performance Benchmark (SP^{2}Bench) [14]. SP^{2}Bench consists of a deterministic dataset generator, and 12 representative queries to be executed on the generated datasets. The datasets represent relationships between fictive academic papers and their authors, following the model of academic publications in the DBLP database.

We compare the performances of four engines: Sesame 2.6.1 [5], Virtuoso 6.1.4 [6], 4store 1.1.4 [7] and our own Castor. Sesame was configured to use its native on-disk store with three indexes (spoc, posc, ospc). The other engines were left in their default configuration. We did not include RDF-3x in the comparison as it is unable to handle the filters appearing in the queries. For queries involving filters, we have also tested a version of Castor that does not post them as constraints, but instead evaluate them in a post-processing step.

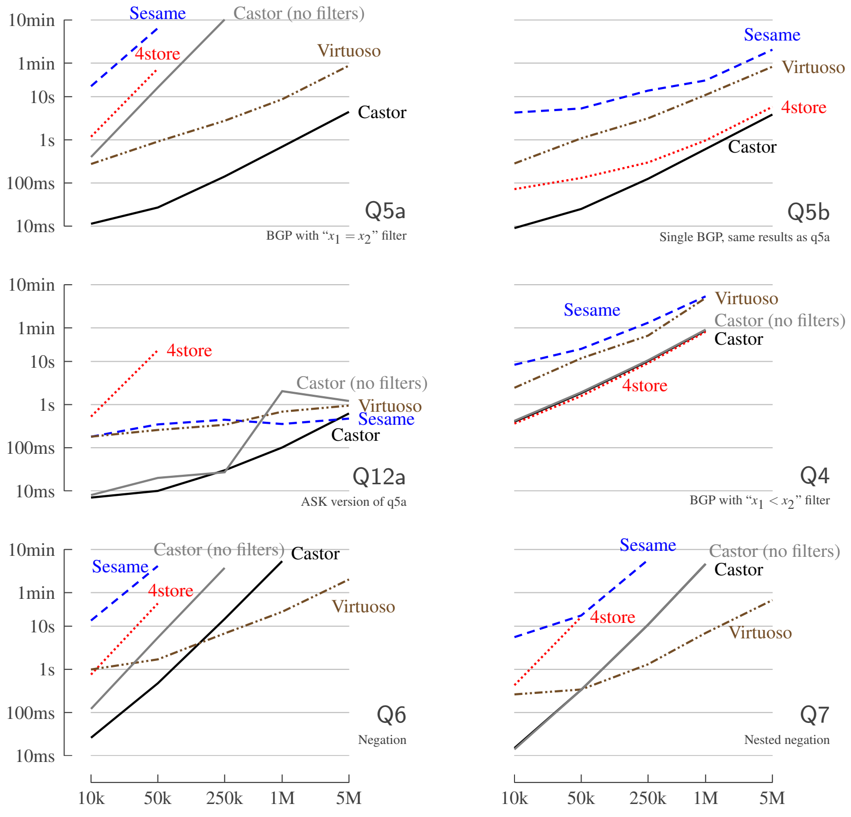


Fig. 6. Castor is competitive and often outperforms state-of-the-art SPARQL engines on complex queries. The x-axis represents the dataset size in terms of number of triples. The y-axis is the query execution time. Both axes have a logarithmic scale.

We have generated 6 datasets with 10k, 25k, 250k, 1M and 5M triples. We have performed three cold runs of each query over all the generated datasets, i.e., between two runs the engines were restarted and the system caches cleared with "`sysctl -w vm.drop_caches=3`". We have set a timeout of 30 minutes. Please note that cold runs may not give the most significant results for some engines. E.g., Virtuoso aggressively fills its cache on the first query in order to perform better on subsequent queries. However, such setting corresponds to the one used by the authors of SP^{2}Bench, so we have chosen to use it as well. All experiments were conducted on an Intel Pentium 4 3.2 GHz computer running ArchLinux 64bits with kernel 3.2.6, 3 GB of DDR-400 RAM and a 40 GB Samsung SP0411C SATA/150 disk with ext4 filesystem. We report the time spent to execute the queries, not including the time needed to load the datasets.

The authors of SP^{2}Bench have identified four queries that are more challenging than the others: Q4, Q5a, Q6 and Q7. The execution time of those queries, along with two variations of Q5a, are reported in Figure 6.

Q5a and Q5b compute the same set of solutions. Q5a enforces the equality of two variables with a filter, whereas Q5b uses a single variable for both. Note that such optimization is difficult to do automatically, as equivalence does not imply identity in SPARQL. For example, `"42"^^xsd:integer` and `"42.0"^^xsd:decimal` compare equal in a filter, but are not the same RDF term and may thus not be matched in a BGP. Detecting whether one can replace the two equivalent variables by a single one requires a costly analysis of the dataset, which is not performed by any of the tested engines. Sesame and 4store timed out when trying to solve Q5a on the 250k and above datasets. Virtuoso does not differentiate equivalent values and treats equality as identity. Such behavior breaks the SPARQL standard and can lead to wrong results. Castor does no query optimization, but still performs equally well on both variants thanks to its ability to exploit the filter at every node of the search tree. Q12a replaces the `SELECT` keyword by `ASK` in Q5a. The solution is a boolean value reflecting whether there exists a solution to the query. Thus, we only have to look for the first solution. However, Castor still needs to initialize the search tree, which is the greatest cost. Virtuoso and 4store behave similarly to Q5a, but Sesame is able to find the answer much more quickly.

Executing Q4 results in many solutions (e.g., for the 1M dataset, Q4 results in 2.5×10^6 solutions versus 3.5×10^4 solutions for Q5a). The filter does not allow for much pruning, as shown by the very similar performances between the two variants of Castor. Nevertheless, Castor is still competitive with the other engines. None of the engines were able to solve the query for the 5M dataset in less than 30 minutes.

Table 1. Castor is the fastest or second fastest engine for nearly every query. The ranking of the engines is shown for each query. The last column is the average rank for every engine.

Query	1	2	3a	3b	3c	4	5a	5b	6	7	8	9	10	11	12a	12b	Mean
Castor	2	2	2	2	2	2	**1**	**1**	2	2	**1**	**1**	2	3	2	**1**	1.8
4store	**1**	**1**	**1**	**1**	**1**	**1**	3	2	3	4	3	2	**1**	**1**	4	3	2.0
Virtuoso	4	3	3	3	3	3	2	3	**1**	**1**	2	3	3	2	3	2	2.6
Sesame	3	4	4	4	4	4	4	4	4	3	4	4	4	4	**1**	4	3.7

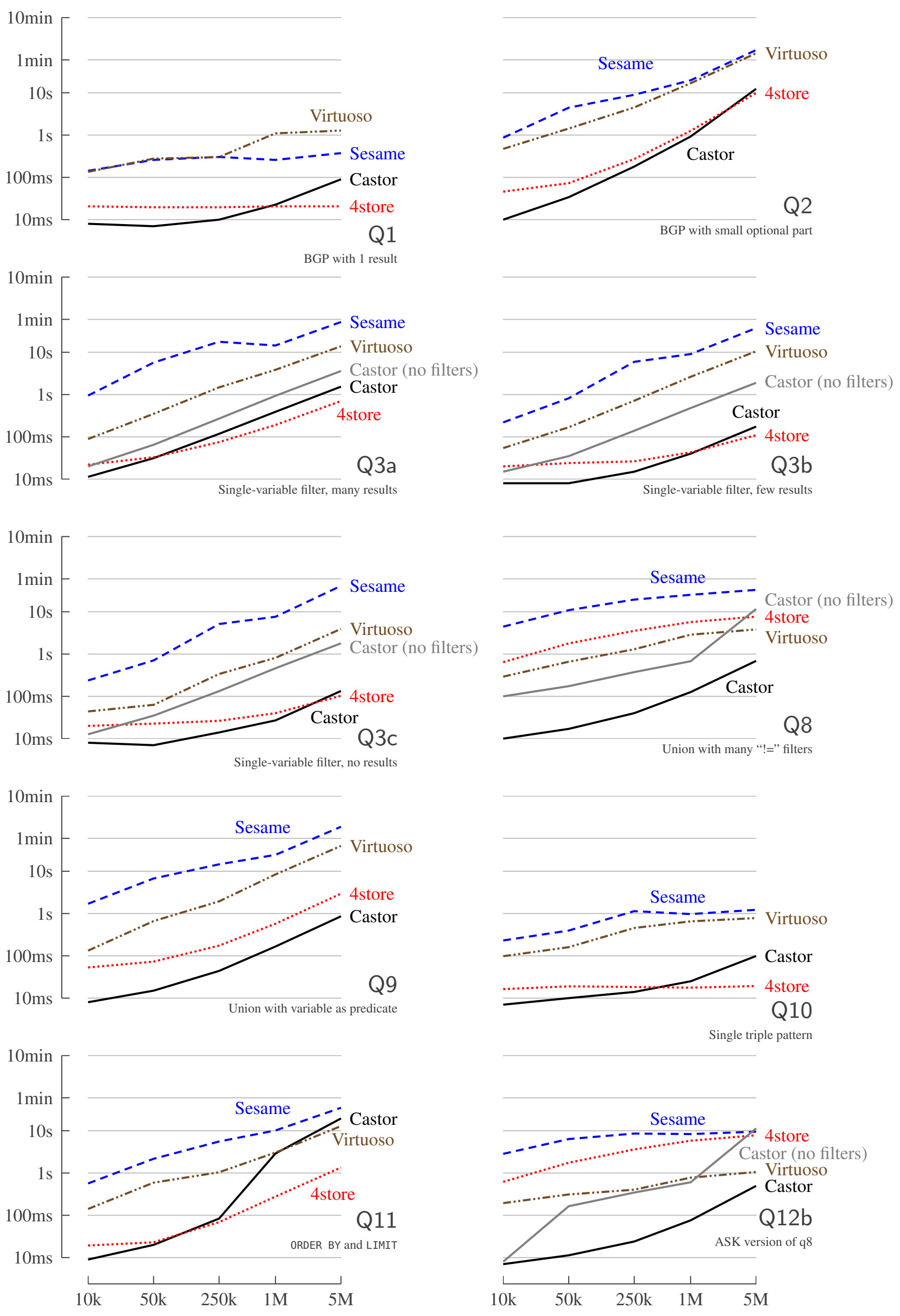


Fig. 7. On simpler queries, Castor is also very competitive with state-of-the-art SPARQL engines. The x-axis represents the dataset size in terms of number of triples. The y-axis is the query execution time. Both axes have a logarithmic scale.

Figure 7 shows the results for the other queries, except Q12c. Query Q12c involves an RDF value that is not present in the dataset. It is solved in constant time by all engines equally well. For all queries, Castor is competitive with the other engines or outperforming them. The sharp decrease of performances of Castor in Q11 between the 250k and the 1M datasets is due to the fixed size of the triple store cache. The hit ratio drops from 99.6% to 40.4%.

For each query, we sort the engines in lexicographical order, first by the largest dataset solved, then by the execution time on the largest dataset. The obtained ranks are shown in Table 1. Castor is ranked first for 5 queries out of 16, and second for all other queries but one. The 4store engine is ranked first on 8 queries, but does not fare as well on the other queries. In most of the queries where 4store is ranked first, the execution time of Castor is very close to the execution time of 4store. Virtuoso performs well on some difficult queries (Q6 and Q7), but is behind for the other queries. Sesame performs the worst of the tested engines.

6 Conclusion

We presented a Constraint Programming approach to solving SPARQL queries. In contrast to the relational database approach, we are able to exploit filters early-on during the search without requiring advanced query optimization. We showed the main design decision in the implementation of Castor, our prototype SPARQL engine based on CP techniques. We compared Castor with state-of-the-art engines, showing the feasibility and performance of our approach.

Castor is however still an early prototype. It has room for several improvements and extensions. For example, the form of the search tree is defined by basic heuristic functions. At each node, we select the variable with the smallest domain. An alternative might be selecting central variables of star-shaped queries first. Also no research has been done yet to find an optimal ordering of the propagators: they are simply called successively until a fix-point is reached. To reach the fix-point more quickly, it would be better to call first the propagators performing more pruning. Maybe selectivity estimates, a tool used in relational databases [17], could be used to order the propagators. This raises the more general question of whether and how optimization techniques used in relational databases can be combined with the CP approach.

Acknowledgments. The authors want to thank the anonymous reviewers for their constructive 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.

References

1. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for semantic web data management. The VLDB Journal 18, 385–406 (2009)

2. 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. 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. Bessiere, C.: Handbook of Constraint Programming, ch. 3. Elsevier Science Inc. (2006)
5. 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)
6. Erling, O., Mikhailov, I.: RDF Support in the Virtuoso DBMS. In: Pellegrini, T., Auer, S., Tochtermann, K., Schaffert, S. (eds.) Networked Knowledge - Networked Media. Studies in Computational Intelligence, vol. 221, pp. 7–24. Springer, Heidelberg (2009)
7. 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)
8. Hose, K., Schenkel, R., Theobald, M., Weikum, G.: Database foundations for scalable RDF processing. In: Polleres, A., d'Amato, C., Arenas, M., Handschuh, S., Kroner, P., Ossowski, S., Patel-Schneider, P. (eds.) Reasoning Web 2011. LNCS, vol. 6848, pp. 202–249. Springer, Heidelberg (2011)
9. 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/`
10. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proc. VLDB Endow. 1, 647–659 (2008)
11. Piatetsky-Shapiro, G., Connell, C.: Accurate estimation of the number of tuples satisfying a condition. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD 1984, pp. 256–276. ACM, New York (1984)
12. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF (January 2008), `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`
13. le Clément de Saint-Marcq, V., Deville, Y., Solnon, C.: An Efficient Light Solver for Querying the Semantic Web. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 145–159. Springer, Heidelberg (2011)
14. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^{2}Bench: A SPARQL performance benchmark. In: Proc. IEEE 25th Int. Conf. Data Engineering, ICDE 2009, pp. 222–233 (2009)
15. Solnon, C.: Alldifferent-based filtering for subgraph isomorphism. Artificial Intelligence 174(12-13), 850–864 (2010)
16. Steinbrunn, M., Moerkotte, G., Kemper, A.: Heuristic and randomized optimization for the join ordering problem. The VLDB Journal 6, 191–208 (1997)
17. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL basic graph pattern optimization using selectivity estimation. In: Proceeding of the 17th International Conference on World Wide Web, WWW 2008, pp. 595–604. ACM, New York (2008)
18. Van Hentenryck, P., Deville, Y., Teng, C.M.: A generic arc-consistency algorithm and its specializations. Artificial Intelligence 57(2-3), 291–321 (1992)
19. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endow. 1, 1008–1019 (2008)
20. Zampelli, S., Deville, Y., Solnon, C.: Solving subgraph isomorphism problems with constraint programming. Constraints 15, 327–353 (2010)