HAL
open science

Un solveur léger efficace pour interroger le Web Sémantique

Vianney Le Clément de Saint-Marcq, Yves Deville, Christine Solnon, Pierre-Antoine Champin

To cite this version:

Vianney Le Clément de Saint-Marcq, Yves Deville, Christine Solnon, Pierre-Antoine Champin. Un solveur léger efficace pour interroger le Web Sémantique. JFPC 2012, May 2012, Toulouse, France. hal-00809859

HAL Id: hal-00809859

https://inria.hal.science/hal-00809859

Submitted on 10 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Un Solveur Léger Efficace pour Interroger le Web Sémantique *

V. le Clément de Saint-Marcq[1,2,3] **Y. Deville**[1] **C. Solnon**[2,4] **P.-A. Champin**[2,3]

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

Résumé

Le Web Sémantique vise à construire des bases de données inter-domaines et distribuées à travers l'internet. SPARQL est un langage de requêtes standard pour ces bases de données. L'évaluation de telles requêtes est cependant NP-difficile. Nous modélisons les requêtes SPARQL de manière déclarative, au moyen de CSPs. Une sémantique opérationnelle CP est proposée. Elle peut être utilisée pour une implémentation directe dans des solveurs CP existants. Pour traiter des bases de données de grande taille, nous introduisons un solveur spécialisé, léger et efficace, Castor. Les benchmarks montrent la faisabilité et l'efficacité de l'approche.

1 Introduction

L'Internet est devenu le moyen privilégié de recherche d'information dans la vie de tous les jours. Bien que l'information disponible en abondance sur le Web soit de plus en plus accessible pour les utilisateurs humains, les ordinateurs ont encore du mal à la comprendre. Les développeurs doivent s'appuyer sur des techniques d'apprentissage machine probabilistes [5] ou des APIs dédiées à certains sites (par exemple, Google API), ou recourir à l'écriture d'un analyseur spécialisé devant être mis à jour à chaque changement de mise en page.

Le Web Sémantique est une initiative du World Wide Web Consortium (W3C) pour permettre aux sites de publier des données lisibles par la machine à côté des documents destinés à l'être humain. La fusion de toutes les données publiées sur le Web Sémantique résulte en une grande base de données globale. Ce caractère global du Web sémantique implique une structure beaucoup plus souple que les bases de données relationnelles traditionnelles. Cette structure permet de stocker des données non liées, mais rend l'interrogation de la base plus difficile. SPARQL [16] est un langage de requête pour le Web Sémantique qui a été standardisé par le W3C. Évaluer les requêtes SPARQL est connu pour être NP-difficile [15].

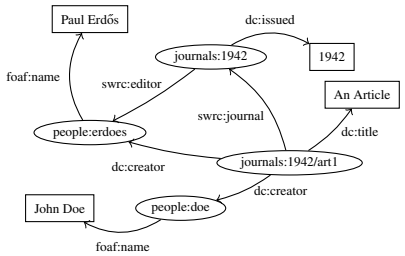
Le modèle d'exécution des moteurs SPARQL actuels (par exemple, Sesame [4], 4store [9] ou Virtuoso [7]) est basé sur l'algèbre relationnel. Une requête est subdivisée en sous-requêtes qui sont calculées séparément. Les ensembles de réponse sont ensuite fusionnés. Les filtres additionnels spécifiés par l'utilisateur sont souvent traités après ces opérations de jointure. La Programmation par Contraintes (CP), par contre, est en mesure d'exploiter les filtres comme des contraintes lors de la recherche. Un moteur de recherche basé sur les contraintes est donc bien adapté pour le Web Sémantique.

Contributions. Notre première contribution est un modèle déclaratif basé sur le formalisme des problèmes de satisfaction de contraintes (CSP) et une sémantique opérationnelle basée sur la programmation par contraintes (CP) pour résoudre des requêtes SPARQL. Les solveurs CP existants ne sont toutefois pas conçus pour traiter les immenses domaines en lien avec les bases de données du Web sémantique. La deuxième contribution de ce travail est un solveur spécialisé léger, appelée Castor, pour l'exécution de requêtes SPARQL. Sur des benchmarks standards, Castor est compétitif avec les moteurs existants et améliore l'état de l'art sur des requêtes complexes.

*Ce papier est une version étendue de travaux publiés à CP 2011 et ESWC 2012.

```
people:erdoes foaf:name   "Paul Erdős" .
people:doe    foaf:name   "John Doe" .
journals:1942 dc:issued   1942 .
journals:1942 swrc:editor people:erdoes .
journals:1942/art1 dc:title   "An Article" .
journals:1942/art1 dc:creator people:erdoes .
journals:1942/art1 dc:creator people:doe .
```

(a) Ensemble de triplets



(b) Représentation par un graphe

FIGURE 1 – Exemple de base RDF représentant un journal fictif édité par Paul Erdős et un article de ce journal écrit par Erdős et Doe. Ici, `people:erdoes` et `foaf:name` sont des URIs alors que `"Paul Erdős"` est un littéral.

Plan. La section suivante explique comment les données sont représentées dans le Web Sémantique et la façon d'interroger les données. Les section 3 et 4 montrent respectivement le modèle déclaratif et la sémantique opérationnelle pour résoudre les requêtes. La section 5 présente notre solveur léger implémentant la sémantique opérationnelle. La section 6 évalue la faisabilité et l'efficacité de notre approche au travers d'un benchmark standard.

2 Le Web Sémantique et le Langage de Requêtes SPARQL

Le *Resource Description Framework* (RDF) [10] permet de modéliser la connaissance comme un ensemble de triplets (sujet, prédicat, objet). Ces triplets expriment des relations, décrites par les prédicats, entre sujets et objets. Les trois éléments d'un triplet peuvent être des ressources arbitraires identifiées par des *Uniform Resource Identifiers* (URI) [1]. Les objets peuvent également être des valeurs littérales, telles que des chaînes de caractères, des nombres, des dates ou des données spécialisées. Une base de données RDF peut être représentée par un multigraphe étiqueté orienté comme le montre la figure 1.

SPARQL [16] est un langage de requête pour RDF. Une requête basique est un ensemble de motifs de triplets, à savoir des triplets où les éléments peuvent être remplacés par des variables. Les requêtes basiques peuvent être assemblées dans des requêtes composées avec éventuellement des parties optionnelles ou alternatives. Les filtres ajoutent des contraintes sur les variables. Une solution d'une requête est une affectation de variables à des URIs ou littéraux figurant dans la base de données. La substitution des variables dans la requête par leurs valeurs attribuées dans la solution donne un sous-ensemble de la base de données. Une requête SPARQL peut également définir un sous-ensemble de variables à renvoyer, un ordre de tri, etc. mais, ceci n'étant pas pertinent pour ce papier, a été omis.

Plus formellement, soient U, L et V des ensembles infinis disjoints deux-à-deux représentant respectivement les URIs, les littéraux et les variables. Une instance du problème SPARQL est définie par une paire (S,Q) telle que $S \subset U \times U \times (U \cup L)$ est un ensemble fini de triplets correspondant à la base de données, et Q est une requête. La syntaxe des requêtes est définie récursivement comme suit [2]. La sémantique sera définie dans la section suivante.

- Une requête basique est un ensemble de motifs de triplets (s,p,o) tels que $s,p \in U \cup V$ et $o \in U \cup L \cup V$. La différence avec les bases RDF est que nous pouvons avoir des variables à la place des URI et des littéraux.
- Soient Q_1 et Q_2 des requêtes. Q_1 . Q_2, Q_1 `OPTIONAL` Q_2 et Q_1 `UNION` Q_2 sont des requêtes composées.
- Soient Q une requête et c une contrainte de sorte que chaque variable de c apparaît au moins une fois dans Q. Q `FILTER` c est une requête contrainte. Le langage d'expressions de SPARQL utilisé pour définir c comprend des opérateurs arithmétiques, booléens, et de comparaison, des expressions régulières pour les littéraux et certains opérateurs spécifiques à RDF.

Étant donnée une base de données S, U_S et L_S dénotent respectivement les ensembles d'URIs et de littéraux qui apparaissent dans S. Étant donnée une requête Q, $\text{vars}(Q)$ dénote l'ensemble des variables apparaissant dans Q.

3 Un Modèle Déclaratif CSP des Requêtes SPARQL

Une solution à une instance de problème SPARQL (S,Q) est une affectation σ de variables de Q à des valeurs de $U_S \cup L_S$, en d'autres termes un ensemble de paires variable/valeur. Étant données une solution σ et une requête Q, $\sigma(Q)$ dénote la requête obtenue en remplaçant chaque occurrence d'une variable affectée dans σ par sa valeur. Le but est de trouver toutes les solutions. On note $\text{sol}(S,Q)$ l'ensemble de toutes les solutions à (S,Q).

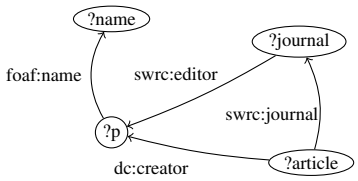
Contrairement aux CSPs classiques, une solution σ ne doit pas nécessairement couvrir toutes les variables apparaissant dans Q. Par exemple, si une variable x apparaît uni-

1. Plus précisément, RDF fait usage de *URI references*, mais la différence n'est pas pertinente au présent papier. La spécification permet aussi de remplacer les sujets et les objets par des nœuds vierges, c'est à dire des ressources sans identificateur. Sans perte de généralité, les nœuds vierges seront considérés comme URIs réguliers dans cet article.

2. Par souci de clarté, nous apportons quelques simplifications au langage. Ces hypothèses n'altèrent pas l'expressivité de SPARQL.

```
SELECT *
WHERE {
  ?p foaf:name ?name .
  ?journal swrc:editor ?p .
  ?article swrc:journal ?journal .
  ?article dc:creator ?p .
}
```

(a) Requête SPARQL



(b) Graphe motif associé

FIGURE 2 – Exemple de requête basique recherchant les éditeurs de journaux ayant publié un article dans le même journal. Les variables sont préfixées d'un point d'interrogation, par exemple `?name`. L'exécution de la requête sur la base de données de la figure 1 donne la solution unique {(p,people:erdoes), (name,"Paul Erdős"), (journal,journals:1942), (article,journals:1942/art1)}.

quement dans une partie optionnelle qui n'a pas été trouvée dans une solution s, x n'apparaîtra pas dans la solution σ. Ces variables sont dites non liées.

Dans cette section, nous définissons l'ensemble des solutions d'une instance de problème SPARQL au moyen de CSPs, donnant ainsi une sémantique dénotationnelle des requêtes SPARQL. Notez que, ce faisant, nous transformons un langage déclaratif, SPARQL, en un autre basé sur les CSPs qui peuvent être résolus par des solveurs existants.

3.1 Requêtes Basiques

Une requête basique BQ est un ensemble de motifs de triplets (s,p,o). Dans cette forme simple, une affectation σ est une solution si $\sigma(BQ) \subseteq S$.

Le problème SPARQL (S,BQ) peut être considéré comme un problème d'appariement de graphes d'un graphe requête associé à BQ à un graphe cible associé à S [3], comme illustré dans la figure 2. Cependant, même cette forme basique de requête est plus générale que les appariements de graphes classiques, tels que l'homomorphisme de graphes ou l'isomorphisme de sous-graphes. Les variables sur les arcs (les prédicats) peuvent imposer des relations additionnelles entre différents arcs. Ce problème est donc déjà NP-difficile.

Nous définissons formellement l'ensemble $\mathrm{sol}(S,BQ)$ comme les solutions du CSP (X,D,C) tel que

- $X = \mathrm{vars}(BQ)$,
- toutes les variables ont le même domaine, contenant tous les URIs et littéraux de S, c'est à dire $\forall x \in X, D(x) = U_S \cup L_S$,
- les contraintes assurent que chaque triplet de la requête appartient à la base de données, c'est à dire

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

où Member est la contrainte d'appartenance à un ensemble.

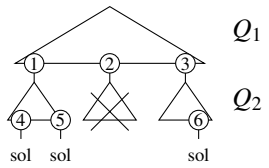
3.2 Requêtes Composées

Les requêtes plus avancées, comportant par exemple des parties optionnelles, ne peuvent être traduites directement en CSP. En effet, certaines requêtes s'appuient sur la non-satisfiabilité d'une sous-requête, qui est coNP-difficile. Les CSP ne peuvent modéliser que des problèmes NP.

Q_1 **.** Q_2**.** Deux requêtes peuvent être concaténées avec le symbole de jointure ou de concaténation (`.`). L'ensemble solution de la concaténation est le produit cartésien des ensembles solution des deux requêtes. Un tel produit cartésien est obtenu en fusionnant chaque paire de solutions affectant les mêmes valeurs aux variables communes. Notez que l'opérateur est commutatif, c.à.d. $Q_1 \,.\, Q_2$ est équivalent à $Q_2 \,.\, Q_1$. L'ensemble des solutions est défini par :

$$\mathrm{sol}(S, Q_1 \,.\, Q_2) = \{\, \sigma_1 \cup \sigma_2 \mid \sigma_1 \in \mathrm{sol}(S,Q_1), \\ \sigma_2 \in \mathrm{sol}(S, \sigma_1(Q_2)) \,\} \ .$$

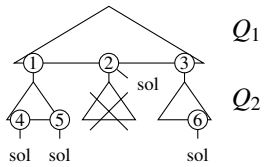
La figure ci-dessous représente un exemple. Un triangle représente un arbre de recherche d'une sous-requête. Les cercles au bas d'un triangle sont les solutions de la sous-requête. Les cercles 1, 2 et 3 représentent $\mathrm{sol}(S,Q_1)$. La solution 1 est étendue en les solutions 4 et 5 dans l'arbre de recherche de $\mathrm{sol}(S,\sigma_1(Q_2))$. Les solutions 4, 5 et 6 sont les solutions de la concaténation. Si Q_1 et Q_2 sont toutes deux des requêtes basiques, nous pouvons calculer la concaténation plus efficacement en fusionnant les deux ensembles de motifs de triplets et résoudre la requête basique résultante comme expliqué dans la section 3.1.



Q_1 **OPTIONAL** Q_2**.** L'opérateur `OPTIONAL` résout la sous-requête de gauche Q_1 et *essaie* de résoudre la sous-requête de droite Q_2. Si une solution de Q_1 ne peut être étendue en une solution de $Q_1 \,.\, Q_2$, alors cette solution de Q_1 devient

aussi une solution de la requête. Plus formellement,

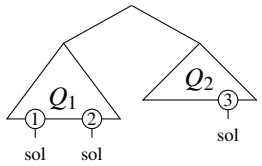
$$\mathrm{sol}(S, Q_1\,\texttt{OPTIONAL}\,Q_2) = \mathrm{sol}(S, Q_1\,.\,Q_2) \cup \{\,\sigma \in \mathrm{sol}(S, Q_1) \mid \mathrm{sol}(S, \sigma(Q_2)) = \varnothing\,\} \quad .$$



Par rapport à l'exemple de l'opérateur de concaténation, le cercle 2 dans la figure devient une solution de la requête composée. Le test d'incohérence rend la recherche difficile. En effet, dans le cas simple, Q_2 est une requête basique et est donc modélisée par un CSP. Toutefois, puisque la vérification de la cohérence d'un CSP est NP-difficile, la vérification de son incohérence est coNP-difficile. Afin de garantir la compositionnalité de la sémantique, nous imposons que si $Q_1\,\texttt{OPTIONAL}\,Q_2$ est une sous-requête d'une requête Q, les variables apparaissant dans Q_2 mais pas dans Q_1 ($\mathrm{vars}(Q_2) \setminus \mathrm{vars}(Q_1)$) n'apparaissent pas ailleurs dans Q. Cette condition n'altère pas l'expressivité du langage [1].

Q_1 **UNION** Q_2**.** Les disjonctions sont introduites par l'opérateur UNION. L'ensemble solution de l'union de deux requêtes est l'union des ensembles solution des deux requêtes. Les solutions des deux requêtes sont calculées séparément :

$$\mathrm{sol}(S, Q_1\,\texttt{UNION}\,Q_2) = \mathrm{sol}(S, Q_1) \,\cup\, \mathrm{sol}(S, Q_2) \quad .$$



3.3 Filtres

L'opérateur FILTER retire les solutions de Q ne satisfaisant pas la contrainte c, à savoir :

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

où $c(\sigma)$ est vrai si c est satisfait par σ. La référence SPARQL [16] définit la sémantique des contraintes, également en cas de variables non liées.

L'opérateur FILTER peut être utilisé a posteriori, pour enlever les solutions qui ne satisfont pas à certaines contraintes. C'est ce qui est fait habituellement par les moteurs SPARQL existants. Cependant, ces contraintes peuvent aussi être utilisées pendant le processus de recherche afin d'élaguer l'arbre de recherche. Un des buts de cet article est d'étudier le bénéfice de l'utilisation de CP, qui exploite activement les contraintes pour réduire l'espace de recherche, pour résoudre les requêtes SPARQL.

Lorsque l'opérateur FILTER est appliqué directement à une requête basique BQ, les contraintes peuvent être simplement ajoutées à l'ensemble des contraintes d'appartenance associé à la requête, c.à.d. $\mathrm{sol}(S, BQ\,\texttt{FILTER}\,c)$ est égal à l'ensemble solution du CSP $(X, D, C \cup \{c\})$, où (X, D, C) est le CSP associé à (S, BQ). Bien sûr, trouver toutes les solutions du CSP $(X, D, C \cup c)$ est généralement plus rapide que trouver toutes les solutions pour (X, D, C), et puis supprimer celles qui ne satisfont pas c.

Les filtres appliqués sur des requêtes composés peuvent parfois être *poussés* dans les sous-requêtes [18]. Par exemple $(Q_1\,\texttt{UNION}\,Q_2)\,\texttt{FILTER}\,c$ peut être réécrit sous la forme $(Q_1\,\texttt{FILTER}\,c)$ UNION $(Q_2\,\texttt{FILTER}\,c)$. De telles optimisations de requête sont courantes dans les moteurs de base de données.

4 Une Modélisation CP Opérationnelle de requêtes SPARQL

La sémantique dénotationnelle de SPARQL peut être transformée en une sémantique opérationnelle avec des solveurs CP conventionnels à condition que ceux-ci permettent de poster des contraintes lors de la recherche. Des exemples de tels solveurs sont Comet [6] ou Gecode [8]. Nous détaillons la sémantique opérationnelle de requêtes SPARQL, c.à.d. comment l'ensemble $\mathrm{sol}(S, Q)$ est calculé. Ce modèle peut être utilisé pour une implémentation directe dans les solveurs existants.

Pour exécuter une requête Q dans une base de données S, nous définissons un tableau global de variables CP $X = \mathrm{vars}(Q)$. Le domaine initial de chaque variable $x \in X$ est $D(x) = U_S \cup L_S$. L'ensemble des contraintes C est initialement vide. Pour expliquer le postage des contraintes et la recherche, nous utilisons Comet comme notation. Le code suivant résout la requête Q :

```
solveall<cp> {
} using {
  sol(Q);
  output(); // affiche la solution
}
```

Le premier bloc (vide) poste les contraintes, le second décrit la recherche. La fonction sol(Q) sera définie pour chaque type de requête. Elle poste des contraintes et introduit des points de choix. Les points de choix sont soit explicites avec les mots-clés try, soit implicites lors de l'étiquetage des variables avec label. Quand un échec est rencontré, soit explicitement avec cp.fail(), soit implicitement lors de la propagation d'une contrainte, la recherche retourne au dernier point de choix et reprend l'exécution dans l'autre branche. Un retour en arrière se produit également après avoir affiché une solution à la fin du bloc using

```
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) Requête basique avec filtre

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

(b) Requête composée avec filtre

FIGURE 3 – Les filtres appliquées aux requêtes basiques sont postées comme des contraintes. Dans tous les autres cas, ils sont vérifiés après avoir résolu la sous-requête.

```
function
sol(Q1 . Q2) {
  sol(Q1);
  sol(Q2);
}
```

(a) Concaténation

```
function
sol(Q1 UNION Q2) {
  try<cp> {
    sol(Q1);
  }|{
    sol(Q2);
  }
}
```

(c) Union

```
function
sol(Q1 OPTIONAL Q2) {
  sol(Q1);
  Boolean cons(false);
  try<cp> {
    sol(Q2);
    cons := true;
  }|{
    if(cons)
      cp.fail();
  }
}
```

(b) Optional

FIGURE 4 – Les requêtes composées sont résolues récursivement.

pour chercher les autres solutions. Nous supposons une recherche en profondeur d'abord, développant les branches de gauche à droite.

Comme nous n'étiquetons pas toutes les variables dans chaque branche, les domaines de certaines variables peuvent encore être intacts lors de l'affichage d'une solution. Ces variables sont considérées comme non liées et ne sont pas incluses dans la solution. En effet, nous étiquetons toujours toutes les variables d'une requête basique. Les variables non liées n'apparaissent pas dans les requêtes basique le long d'une branche, en raison de disjonctions introduites par `UNION` ou de sous-requêtes optionnelles incohérentes. Aucune contrainte n'est postée sur ces variables. Leurs domaines ne sont donc pas réduits.

La figure 3 montre la fonction `sol` pour une requête basique avec un filtre. Le filtre est posté avec les contraintes de triplets et élague l'arbre de recherche dès le début. Dans certains cas des propagateurs spécifiques peuvent être utilisés, par exemple pour les opérateurs arithmétiques ou de comparaison. Dans tous les cas, nous pouvons nous rabattre sur un évaluateur d'expressions SPARQL existant pour propager la condition par *forward checking*, c'est à dire lorsque toutes les variables sauf une sont affectées.

Les filtres sur les requêtes composées ne peuvent toutefois être vérifiés qu'après chaque solution de la sous-requête comme le montre la figure 3b. Notez que la condition c n'est pas postée puisqu'il peut y avoir des variables non liées qui ont besoin d'être traitées selon la spécification SPARQL.

Les concaténations sont calculées de façon séquentielle, comme indiqué dans la figure 4a. L'opérateur `OPTIONAL` est similaire à la concaténation et est représenté dans la figure 4b. D'abord $\mathrm{sol}(Q_1)$ est calculé. Avant d'exécuter la seconde sous-requête Q_2, un point de choix est introduit. La branche de gauche calcule $\mathrm{sol}(Q_2)$, fournissant donc les solutions de $Q_1 \,.\, Q_2$. Si elle réussit, la branche de droite est élaguée. Sinon la branche de droite est vide et donc $\mathrm{sol}(Q_1)$ est retourné comme une solution. Notez que cela ne fonctionne qu'avec une recherche en profondeur d'abord explorant la branche de gauche en premier. Enfin pour l'opérateur `UNION`, les deux sous-requêtes sont résolues dans deux branches distinctes, comme le montre la figure 4c.

Il est clair que cette sémantique opérationnelle de requêtes SPARQL calcule l'ensemble de solutions définie par la modélisation déclarative.

5 Castor : un Solveur Léger pour le Web Sémantique

Nous présentons maintenant Castor, un solveur léger, conçu pour résoudre des requêtes SPARQL. Une requête n'implique pas beaucoup de variables et de contraintes. Le principal défi cohére à traiter les domaines immenses associés aux variables. Les solveurs CP existants ne s'adaptent pas bien à ce contexte comme indiqué dans la section expérimentale. L'idée clé de Castor est d'éviter de maintenir et de restaurer des structures de données qui sont proportionnelles aux tailles des domaines. D'une part, nous n'utilisons pas de techniques de propagation avancées qui ont besoin de ce genre de structures coûteuses. D'autre part, la restauration des domaines est une opération peu coûteuse qui nous permet d'explorer de vastes arbres de recherche assez vite pour compenser la perte de propagation.

Dans cette section, nous expliquons les trois principales

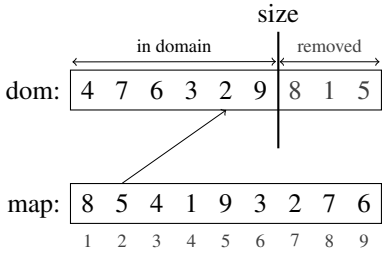


FIGURE 5 – Exemple de représentation du domaine $\{2,3,4,6,7,9\}$, où $\texttt{size} = 6$ et où le domaine initial est $\{1,\ldots,9\}$. Les `size` premières valeurs de `dom` appartiennent au domaine ; les valeurs suivantes sont celles qui ont été retirées. Le tableau `map` fait le lien entre les valeurs et leur position dans `dom`. Par exemple, la valeur 2 a l'indice 5 dans `dom`. Dans cette représentation, uniquement la taille doit être restaurée lors d'un retour en arrière.

composantes du solveur : les variables et la représentation de leurs domaines, les contraintes et leur propagateurs, et les techniques de recherche utilisées pour explorer l'arbre.

5.1 Variables et Représentation des Domaines

Les variables dans Castor sont des nombres entiers prenant les valeurs de 1 jusqu'au nombre de valeurs dans la base de données. L'ordre des valeurs est cohérent avec l'opérateur de comparaison de SPARQL. Deux types de représentation sont considérés pour les domaines. La représentation *discrète* maintient chaque valeur du domaine, mais ignore l'ordre des valeurs. La représentation *bornée* ne maintient que la valeur minimale et maximale. Nous proposons une approche duale, tirant parti des points forts des deux représentations.

Représentation discrète. Nous représentons un domaine fini $D(x)$ d'une variable x par sa taille `size` et deux tableaux `dom` et `map`. Les `size` premières valeurs de `dom` sont dans le domaine de la variable, les autres ont été retirées (voir figure 5). Le tableau `map` fait le lien entre les valeurs et leur position dans le tableau `dom`. De telles structures sont aussi utilisées dans le code pour calculer des isomorphismes de sous-graphes décrit dans [20].

Les invariants suivants sont vérifiés.

- Les tableaux `dom` et `map` sont cohérents, c.à.d. $\texttt{map}[v] = i \Leftrightarrow \texttt{dom}[i] = v$.
- Le domaine $D(x)$ est l'ensemble des `size` premières valeurs de `dom`, c.à.d. $D(x) = \{\, \texttt{dom}[i] \mid i \in \{1,\ldots,\texttt{size}\}\,\}$.
- Toute réduction du domaine ne modifie pas les valeurs retirées précédemment (c.à.d., les valeurs de l'indice $\texttt{size}+1$ jusqu'à la fin du tableau `dom`).

Le dernier invariant nous permet de ne restaurer que `size` quand on revient en arrière. En effet, le partitionnement entre les valeurs retirées et les valeurs encore dans le domaine sera identique. L'ordre des valeurs avant `size` peut avoir changé. Le dernier invariant est respecté lors d'une recherche en profondeur d'abord, car nous retirons des valeurs le long d'une branche avant de revenir en arrière.

Les opérations de base sur le domaine ont toutes une complexité en temps constant. Vérifier si une valeur est dans un domaine est réalisé par la propriété $v \in D(x) \Leftrightarrow \texttt{map}[v] \leq \texttt{size}$. Pour supprimer une valeur, nous l'échangeons avec la dernière valeur dans le domaine et nous décrémentons `size`. Par exemple, pour supprimer la valeur 3 dans la figure 5, nous permutons 3 et 9 dans `dom`, mettons à jour `map` en conséquence et diminuons `size` de un.

Pour restreindre le domaine à un ensemble de valeurs, nous *marquons* chacune des valeurs à conserver, c.à.d. nous échangeons la valeur avec la valeur non-marquée la plus à gauche dans `dom` et incrémentons le nombre de valeurs marquées. Nous fixons ensuite la taille du domaine au nombre de valeurs marquées. L'opération complète a une complexité en temps linéaire au nombre de valeurs conservées.

Représentation bornée. Le domaine est représenté par ses bornes, c'est à dire ses valeurs minimales et maximales. Contrairement à la représentation discrète, cette représentation est une approximation du domaine exact. Nous supposons que toutes les valeurs entre les bornes sont présentes dans le domaine.

Dans cette représentation, nous ne pouvons pas supprimer une valeur au milieu du domaine, car nous ne pouvons pas représenter un trou à l'intérieur des limites. Cependant, l'augmentation de la borne inférieure ou la diminution de la borne supérieure se fait en temps constant.

La structure de données pour cette représentation étant de petite taille (seulement deux nombres), nous copions la structure entière à chaque point de choix. Restaurer le domaine cohére à restaurer les deux bornes.

Approche Duale. Les propagateurs réalisant du *forward checking* ou de la cohérence de domaine retirent des valeurs du domaine et ont donc besoin d'une représentation discrète. Cependant, les propagateurs réalisant de la cohérence aux bornes ne mettent à jour que les bornes des domaines. Pour que ceux-ci soient efficaces, nous avons besoin d'une représentation bornée. C'est pourquoi Castor crée deux variables x_D et x_B (resp. avec une représentation discrète et bornée) pour chaque variable SPARQL x. Les contraintes sont postées en utilisant seulement l'une des deux variables, en fonction de la représentation la plus efficace pour le propagateur associé. En particulier, les contraintes monotones sont postées sur des variables bornées, tandis que les contraintes de triplets sont postées sur des variables discrètes.

Une contrainte supplémentaire $x_D = x_B$ assure le lien entre les deux vues. Assurer la cohérence de domaine pour

cette contrainte est trop coûteux, car cela équivaut à effectuer toutes les opérations sur les bornes sur la représentation discrète. Au lieu de cela, le propagateur de Castor fait du *forward checking*, c'est à dire qu'une fois une variable affectée, l'autre sera affectée à la même valeur. Comme optimisation, lors de la restriction d'un domaine à son intersection avec un ensemble S, nous filtrons les valeurs de S qui sont en dehors des bornes et mettons à jour les bornes de x_B. Cette optimisation ne change pas la complexité car S doit être entièrement parcouru de toute façon.

5.2 Contraintes et propagateurs

Il existe deux types de contraintes dans les requêtes SPARQL : les motifs de triplets et les filtres. Les filtres sur des requêtes composées ne sont vérifiés qu'après affectation de toutes leurs variables. Les filtres sur les requêtes basiques et les triplets sont postés et exploités pendant la recherche. Comme pour les domaines, l'objectif est de minimiser les structures devant être restaurées lors d'un retour en arrière. Dans la version actuelle de Castor, les propagateurs n'utilisent pas ce genre de structures.

Une contrainte dans Castor est un objet qui implémente deux méthodes : `propagate` et `restore`. Quand la contrainte est créée, elle s'enregistre aux événements des variables. La méthode `propagate` est appelée lorsque l'un de ces événements se produit. La méthode `restore` est appelée lorsque la recherche fait marche arrière. Actuellement, chaque variable a quatre événements : `bind`, qui se produit lorsque le domaine devient un singleton, `change`, qui se produit lorsque le domaine a changé, `updateMin` et `updateMax`, qui se produisent lorsque la borne inférieure (resp. supérieure) a changé. Pour savoir quelles valeurs ont été retirées d'une variable depuis la dernière exécution du propagateur, on enregistre (localement à la contrainte) la taille du domaine à la fin de la méthode `propagate`. Les valeurs retirées se situent entre l'ancienne et la nouvelle taille dans le tableau `dom` au prochain appel de la méthode. La méthode `restore` est utilisée pour réinitialiser les tailles enregistrées après un retour en arrière. Les propagateurs sont appelés jusqu'à ce que le point fixe soit atteint.

Motifs de triplets. Un motif de triplet est une contrainte table. Elle réagit à l'événement `bind` des variables. Quand une variable est liée, nous cherchons tous les triplets cohérents de la base de données et nous restreignons les domaines des variables non liées restantes. Les triplets sont stockés sur disque en utilisant le schéma introduit par RDF-3x [14], permettant une recherche efficace par index.

Filtres. La vérification des filtres sur les requêtes composées est effectuée par un évaluateur d'expression conforme aux spécifications SPARQL. L'évaluateur considère toutes les variables avec une taille de domaine supérieure à 1 comme non liées. Les filtres sur les requêtes basiques sont postés avec les motifs de triplets. Le propagateur fait du *forward checking* : dès que toutes les variables sauf une sont affectées, nous enlevons du domaine de la variable non affectée les valeurs qui rendent l'expression fausse.

Certains filtres peuvent être propagés plus efficacement avec des algorithmes spécialisés. Le propagateur pour $x \neq y$ attend qu'une valeur soit attribuée à l'une des deux variables et la supprime du domaine de l'autre variable. Il n'est pas besoin d'itérer sur toutes les valeurs du domaine. La contrainte $x = y$ assure la cohérence d'arc en retirant de $D(y)$ les valeurs qui ont été retirées de $D(x)$ et vice-versa, en réagissant à l'événement `change`. De même, le propagateur pour $x < y$ assure la cohérence aux bornes.

5.3 Recherche

L'arbre de recherche est défini en utilisant une stratégie d'étiquetage. À chaque nœud, une variable est choisie et un nœud enfant est créé pour chacune des valeurs de son domaine. L'heuristique standard du plus petit domaine est utilisée pour choisir la variable. L'ordre des valeurs est défini par leur ordre actuel dans le tableau `dom`.

L'arbre de recherche est exploré en profondeur d'abord afin de restaurer efficacement les domaines (section 5.1) et de contrôler efficacement l'incohérence de sous-requêtes optionnelles (section 4).

Pour poster des contraintes lors de la recherche, nous introduisons des *sous-arbres*. Un sous-arbre contient un ensemble de contraintes et un ensemble de variables à étiqueter. Il parcourt toutes les affectations des variables satisfaisant les contraintes. À chaque affectation, Castor peut créer un nouveau sous-arbre ou afficher la solution, en fonction de la requête. Quand un sous-arbre a été complètement exploré, les domaines des variables sont restaurés à leur état lorsque le sous-arbre a été créé et les contraintes sont enlevées, et la recherche continue dans le sous-arbre précédent.

6 Résultats expérimentaux

Pour évaluer la faisabilité et les performances de notre approche, nous avons exécuté des requêtes du SPARQL Performance Benchmark (SP^{2}Bench) [17]. Le SP^{2}Bench cohére en un générateur déterministe de base de données RDF de taille configurable, et 12 requêtes représentatives. Les bases de données représentent les relations entre des articles académiques fictifs et leurs auteurs, selon le modèle de publications académiques dans la base de données DBLP. Le benchmark comporte à la fois des requêtes basiques et composées, mais ne fait usage que de simples filtres de comparaison. Nous avons retiré des requêtes les modificateurs de solution non pris en charge comme `DISTINCT` et `ORDER BY`. Nous nous concentrons sur les requêtes identifiées comme difficiles par les auteurs du

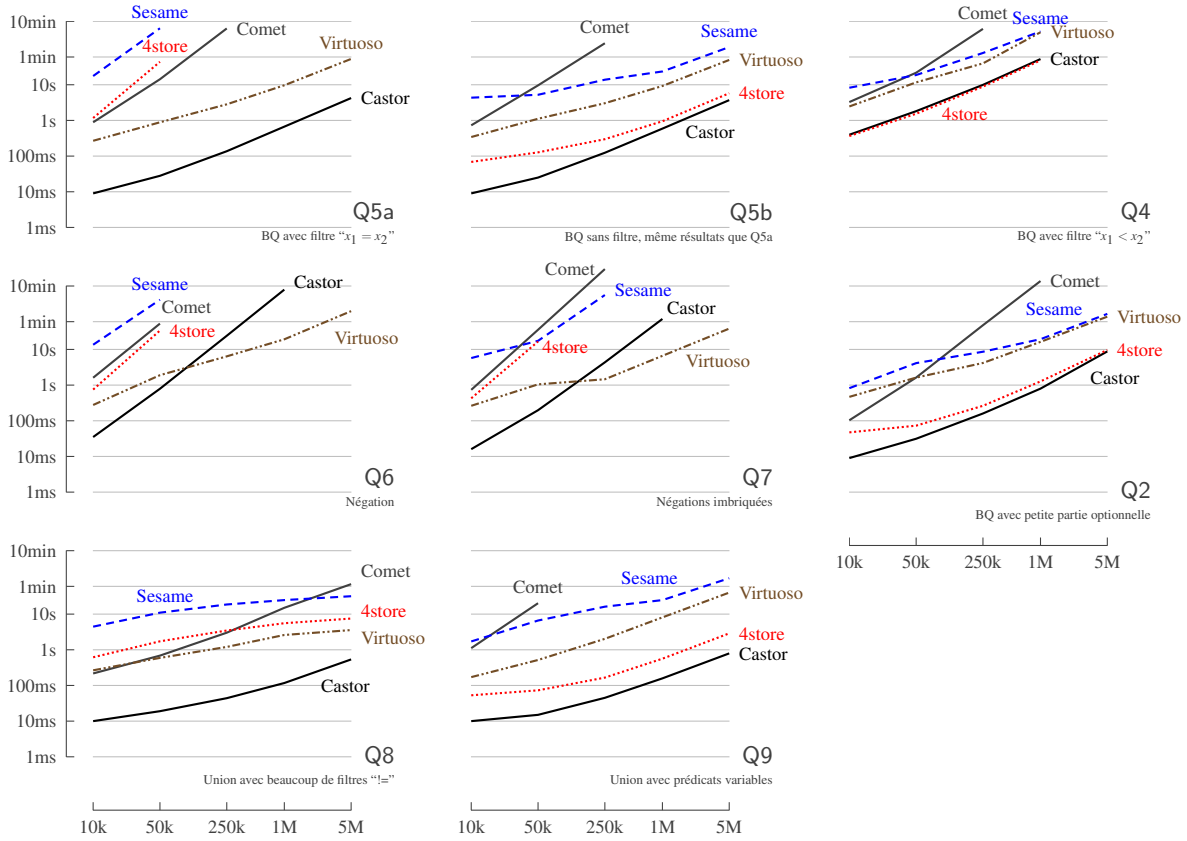


FIGURE 6 – Comparaison sur 8 requêtes. Les abscisses représentent les tailles des bases en nombre de triplets. Les ordonnées donnent les temps d'exécutions des requêtes. Les deux axes ont une échelle logarithmique.

SP^{2}Bench (Q4, Q5, Q6 et Q7) ainsi qu'une requête plus simple (Q2) et deux requêtes impliquant l'opérateur `UNION` (Q8 et Q9). Nous considérons donc 8 requêtes puisqu'il y a deux variantes de Q5.

Nous comparons les performances des moteurs SPARQL de l'état de l'art 4store [9], Sesame [4] et Virtuoso [7], avec le solveur Castor décrit en section 5 et une implémentation directe de la sémantique opérationnelle dans Comet [6]. L'implémentation Comet charge toute la base de données en mémoire. Elle utilise la contrainte table du système pour les motifs de triplets et les expressions du système pour les filtres. Sesame a été exécuté avec le store natif sur disque et trois index (spoc, posc, ospc).

Nous avons généré 6 bases de données de 10k, 25k, 250k, 1M et 5M triplets. Nous avons effectué trois mesures à froid de chaque requête sur toutes les bases de données générées, c.à.d. que les moteurs ont été redémarrés et la cache du système effacé entre deux exécutions. Ces paramètres correspondent à ceux utilisés par les auteurs du SP^{2}Bench. Toutes les expériences ont été effectuées sur un ordinateur Intel Pentium 4 3.2 GHz sous ArchLinux 64bits avec un noyau 3.2.6, 3 Go de RAM DDR-400 et un disque Samsung SP0411C SATA/150 de 40 Go avec un système de fichiers ext4. Nous reprenons le temps écoulé à résoudre les requêtes, sans compter le temps nécessaire pour charger les bases de données. Nous avons vérifié que tous les moteurs trouvent le même ensemble de solutions.

La figure 6 montre le temps d'exécution des requêtes considérées. Notez que les deux axes ont des échelles logarithmiques. Nous discutons maintenant des résultats pour chaque requête.

Requête simple. La requête Q2 est de la forme BQ_1 `OPTIONAL` BQ_2. BQ_1 est une requête basique avec 9 variables et 9 motifs de triplets. La partie optionnelle BQ_2 a un triplet unique avec une seule variable n'apparaissant dans BQ_1. L'exécution de la sous-requête BQ_2 peut donc être faite par un accès à la base de données. 4store et Castor sont de performance égale, surpassant Sesame et Virtuoso. Comet souffre des structures de données lourdes.

Filtres. Les requêtes Q4 et Q5a sont similaires. Les deux sont des requêtes basiques avec un filtre. La requête Q4 a 7 variables, 8 motifs de triplets et un filtre $x_1 < x_2$ sur deux variables x_1 et x_2. La requête Q5a a 6 variables, 6 motifs de triplets et un filtre $x_1 = x_2$. Les moteurs CP sont capables

de dépasser l'état de l'art sur Q5a grâce à leur propagation efficace de la contrainte d'égalité. Nous soupçonnons que cette contrainte soit traitée en post-processing dans Sesame et 4store. Virtuoso obtient de meilleures performances en optimisant la requête, mais cela ne préserve pas correctement la sémantique. La requête Q4 est plus difficile car elle a beaucoup plus de solutions que Q5a ($2.65 \cdot 10^6$ contre $1.01 \cdot 10^5$ pour la base de données avec 1M triplets). Ainsi, le filtrage est moins contraignant. Castor est toujours compétitif avec l'état de l'art.

Les deux variantes de Q5, Q5a et Q5b, calculent exactement le même ensemble de solutions. La dernière encode la contrainte d'égalité dans ses 5 motifs de triplets en utilisant 4 variables. Sans surprise, les moteurs CP fonctionnent de manière semblable sur les deux requêtes comme ils exploitent les filtres dès le début pendant la recherche. Sesame et 4store gèrent bien mieux la requête sans filtre que Q5a. Ceci montre la pertinence de notre approche, particulièrement compte tenu du fait que les filtres sont présents dans environ la moitié des requêtes dans le monde réel [2].

Négations. Une négation en SPARQL est une requête composée qui a la forme $(Q_1 \texttt{ OPTIONAL } Q_2)$ `FILTER` $(!\text{bound}(x))$, où x est une variable n'apparaissant que dans Q_2. Le filtre élimine toutes les solutions affectant une valeur à x, c'est à dire que nous ne gardons que les solutions de Q_1 qui ne peuvent pas être étendues à des solutions de $Q_1 \,.\, Q_2$. La requête Q6 est une négation avec des filtres additionnels à l'intérieur de Q_2. La requête Q7 n'a pas de filtres supplémentaires, mais Q_2 est elle-même une négation imbriquée. Sur les deux requêtes, Castor surpasse Sesame et 4store, mais n'est pas meilleur que Virtuoso.

Unions. Les requêtes composées Q8 et Q9 utilisent l'opérateur `UNION`. La première ajoute des filtres d'inégalité dans ses deux sous-requêtes. Les sous-requêtes de la dernière ne contiennent que deux motifs de triplets chacun. Pourtant, Q9 génère de nombreuses solutions. Comet est incapable d'aller au delà de 50k triplets en raison de ses structures lourdes. Castor surpasse toutefois l'état de l'art. Dans la requête Q8, les deux sous-requêtes alternatives ont certains motifs de triplets dupliqués. L'exploitation de cette propriété peut expliquer l'aspect plat des courbes de l'état de l'art par rapport à Castor.

Conclusion. La table 1 montre la vitesse relative de Castor par rapport à 4 store et Virtuoso. Le but de Castor est d'utiliser la CP pour résoudre des requêtes très contraintes, c.à.d. des requêtes où les filtres éliminent beaucoup de solutions. Ces requêtes (par exemple Q5a) sont traitées beaucoup plus efficacement par Castor. Sur les requêtes reposant plus sur l'accès base de données (comme Q2 et Q9), la CP reste compétitive.

7 Discussion

Nous avons proposé une modélisation déclarative et une sémantique opérationnelle pour résoudre les requêtes SPARQL en utilisant la programmation par contraintes. Nous avons introduit un solveur léger spécialisé implémentant cette sémantique. Nous avons montré que cette approche surpasse l'état de l'art sur des requêtes contraintes, et est compétitive sur la plupart des autres requêtes.

Travaux apparentés. Mamoulis et Stergiou ont utilisé des CSPs pour résoudre des requêtes XPath complexes sur des documents XML [12]. Les documents XML sont des graphes, tout comme les données RDF, mais avec une structure d'arbre sous-jacente. Cette structure est utilisée par les auteurs pour concevoir des propagateurs spécifiques. Cependant, ils ne peuvent pas être utilisés pour les requêtes SPARQL. Mouhoub et Feng ont appliqué la programmation par contraintes à la résolution de requêtes combinatoires dans des bases de données relationnelles [13]. Ces requêtes impliquent de joindre plusieurs tables soumises à des contraintes arithmétiques complexes. Le problème est similaire à SPARQL. Toutefois, les auteurs ne traitent pas les bases volumineuses. Leurs expériences sont limitées aux tables avec 800 lignes. Cette taille n'est pas réaliste pour des données RDF. D'autres travaux visent à étendre le langage de requête SQL standard pour soutenir des expressions de satisfaction de contraintes explicites [11, 19]. Cela permet de résoudre des CSPs dans des bases de données relationnelles.

Travaux futures. Castor est un jeune prototype qui a vocation à être amélioré et étendu. Par exemple, l'heuristique de sélection de variables est le MinDom de base. Une autre solution pourrait être de choisir en premier les variables centrales des requêtes en forme d'étoile. En outre, aucune recherche n'a encore été faite pour trouver un ordonnancement optimal des propagateurs : ils sont simplement appelés successivement jusqu'au point fixe. Pour atteindre le point fixe plus rapidement, il serait préférable d'appeler d'abord les propagateurs élaguant le plus. Les estimations de sélectivité, un outil utilisé dans les bases de données relationnelles [21], pourraient peut-être être utilisés pour ordonner les propagateurs. Cela soulève la question plus générale de savoir si et comment les techniques d'optimisations utilisées dans les bases de données relationnelles peuvent être combinées avec l'approche CP.

Remerciements. Le premier auteur est financé comme assistant de recherche par le FNRS belge (fonds national de la recherche scientifique). Cette recherche est aussi financée par le programme de pôles d'attraction interuniversitaire (état belge, police scientifique belge) et le projet FRFC 2.4504.10 du FNRS belge.

TABLE 1 – Speedup de Castor par rapport à 4store (à gauche) et Virtusoso (à droite). La lettre 'C' (resp. 'V') signifie que seul Castor (resp. Virtuoso) était capable de résoudre l'instance dans la limite de temps.

	10k	50k	250k	1M	5M
Q2	5.39	2.34	1.64	1.59	1.10
Q4	0.92	0.85	0.89	0.92	—
Q5a	133.39	1574.62	C	C	C
Q5b	7.87	5.11	2.40	1.61	1.58
Q6	21.45	42.86	C	C	—
Q7	26.87	84.14	C	C	—
Q8	61.67	91.40	78.80	47.49	13.99
Q9	5.30	4.91	3.69	3.56	3.64

	10k	50k	250k	1M	5M
Q2	53.90	52.02	26.08	20.47	9.42
Q4	6.20	6.41	4.01	5.64	—
Q5a	31.01	31.71	20.66	14.45	12.48
Q5b	39.12	44.63	24.50	15.68	13.37
Q6	7.94	2.40	0.26	0.04	V
Q7	16.92	5.30	0.34	0.09	V
Q8	26.57	31.36	27.36	22.18	6.65
Q9	17.13	34.86	45.09	51.23	50.62

Références

[1] Renzo Angles and Claudio Gutierrez. The expressive power of SPARQL. In *The Semantic Web – ISWC 2008*, volume 5318 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2008.

[2] M. Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. 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] Jean-François Baget. RDF entailment as a graph homomorphism. In *The Semantic Web – ISWC 2005*, volume 3729 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2005.

[4] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame : A generic architecture for storing and querying RDF and RDF Schema. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, pages 54–68. Springer, 2002.

[5] Michael J. Cafarella, Alon Halevy, and Jayant Madhavan. Structured data on the web. *Commun. ACM*, 54 :72–79, February 2011.

[6] Dynamic Decision Technologies Inc. *Comet*, 2010.

[7] Orri Erling and Ivan Mikhailov. RDF support in the Virtuoso DBMS. In *Networked Knowledge – Networked Media*, volume 221 of *Studies in Computational Intelligence*, pages 7–24. Springer, 2009.

[8] Gecode Team. Gecode : Generic constraint development environment, 2006.

[9] Steve Harris, Nick Lamb, and Nigel Shadbolt. 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.

[10] G. Klyne, J. J. Carroll, and B. McBride. Resource description framework (RDF) : Concepts and abstract syntax, 2004.

[11] Robin Lohfert, James Lu, and Dongfang Zhao. Solving SQL constraints by incremental translation to SAT. In *New Frontiers in Applied Artificial Intelligence*, volume 5027 of *Lecture Notes in Computer Science*, pages 669–676. Springer, 2008.

[12] Nikos Mamoulis and Kostas Stergiou. Constraint satisfaction in semi-structured data graphs. In *Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 393–407. Springer, 2004.

[13] Malek Mouhoub and Chang Feng. CSP techniques for solving combinatorial queries within relational databases. In *Intelligent Systems for Knowledge Management*, volume 252 of *Studies in Computational Intelligence*, pages 131–151. Springer, 2009.

[14] Thomas Neumann and Gerhard Weikum. RDF-3X : a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1 :647–659, August 2008.

[15] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34 :16 :1–16 :45, September 2009.

[16] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF, January 2008.

[17] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^2Bench : A SPARQL performance benchmark. In *Proc. IEEE 25th Int. Conf. Data Engineering ICDE '09*, pages 222–233, 2009.

[18] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33. ACM, 2010.

[19] Sebastien Siva and Lesi Wang. A SQL database system for solving constraints. In *Proceeding of the 2nd PhD workshop on Information and knowledge management*, PIKM '08, pages 1–8. ACM, 2008.

[20] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, 174(12–13) :850–864, 2010.

[21] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM.