Le tutoriel SPARQL

Par - Julien Plu (traducteur) - Thibaut Cuvelier (traducteur)

Date de publication : 27 avril 2011

Dernière mise à jour : 7 janvier 2016

DÉBUTANT

Durée : 1 heure

L'objectif de ce tutoriel SPARQL est de donner un cours rapide en SPARQL. Le tutoriel couvre les fonctionnalités majeures du langage de requête au travers d'exemples, mais ne vise pas à être complet.

Si vous cherchez une courte introduction à SPARQL et Jena, essayez **Recherche de données RDF avec SPARQL**.

SPARQL est un **langage de requête** et un **protocole** pour accéder au RDF conçu par le **groupe de travail du W3C RDF Data Access**.

Comme un langage de requête, SPARQL est « orienté données » en ce sens qu'il interroge uniquement les informations détenues dans des modèles ; il n'y a pas d'inférence dans le langage de requête lui-même. Évidemment, le modèle Jena peut être « intelligent » en ce sens qu'il fournit l'impression que certains triplets existent en les créant à la demande, y compris le raisonnement OWL. SPARQL ne fait rien d'autre que prendre la description de ce que l'application veut, sous la forme d'une requête, et retourne cette information, sous la forme d'un ensemble de liaisons ou d'un graphe RDF.

Commentez

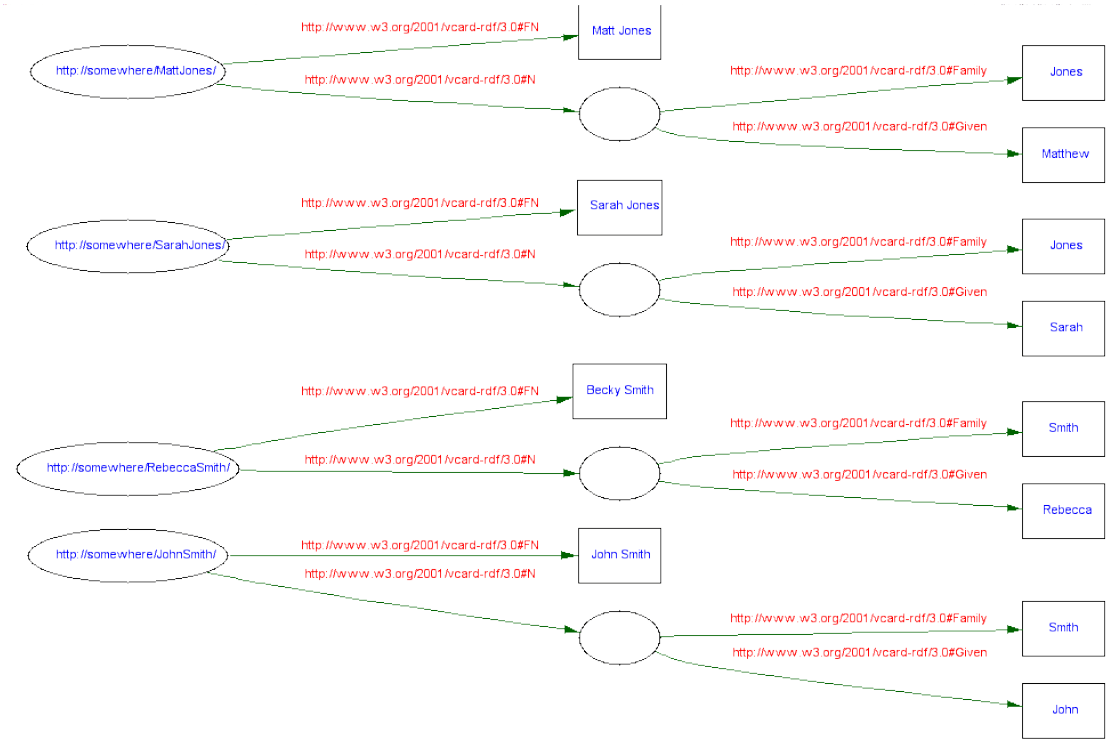
I - les formats de données

Tout d'abord, il faut être clair sur quelles données sont interrogées. SPARQL requête des graphes RDF. Un graphe RDF est un ensemble de triplets (Jena appelle les graphes RDF « modèles » et les triplets « déclarations », car c'est comme ça qu'on les appelait au moment où l'API Jena a été conçue).

Il est important de réaliser que ce sont les triplets qui importent, pas la sérialisation. La sérialisation est juste une manière de coucher par écrit les triplets. Le RDF/XML est la recommandation du W3C, mais il peut être difficile de voir les triplets dans cette forme, car il y a plusieurs façons d'encoder le même graphe. Dans ce tutoriel, on utilisera une sérialisation plus proche des triplets, appelée **Turtle** (voir aussi le langage N3 décrit dans l'**amorce Web sémantique du W3C**).

On va commencer avec des données simples dans **vc-db-1.rdf** : ce fichier contient du RDF pour un certain nombre de descriptions vCard de personnes. Les vCard sont décrits dans la **RFC2426** et la traduction RDF est décrite dans la note du W3C **Représenter des objets vCard en RDF/XML**. Notre base de données d'exemple contient juste quelques informations sur les noms.

Graphiquement, les données ressemblent à :



En triplets, cela pourrait ressembler à :

```
@prefix vCard:   <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix :        <#> .

<http://somewhere/MattJones/>
    vCard:FN     "Matt Jones" ;
    vCard:N      [ vCard:Family
                           "Jones" ;
                   vCard:Given
                           "Matthew"
                 ] .

<http://somewhere/RebeccaSmith/>
```

```
    vCard:FN    "Becky Smith" ;
    vCard:N     [ vCard:Family
                          "Smith" ;
                  vCard:Given
                          "Rebecca"
                ] .

<http://somewhere/JohnSmith/>
    vCard:FN    "John Smith" ;
    vCard:N     [ vCard:Family
                          "Smith" ;
                  vCard:Given
                          "John"
                ] .

<http://somewhere/SarahJones/>
    vCard:FN    "Sarah Jones" ;
    vCard:N     [ vCard:Family
                          "Jones" ;
                  vCard:Given
                          "Sarah"
                ] .
```

Ou même plus explicitement comme triplets :

```
@prefix vCard:   <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://somewhere/MattJones/>  vCard:FN   "Matt Jones" .
<http://somewhere/MattJones/>  vCard:N    _:b0 .
_:b0  vCard:Family "Jones" .
_:b0  vCard:Given  "Matthew" .

<http://somewhere/RebeccaSmith/> vCard:FN    "Becky Smith" .
<http://somewhere/RebeccaSmith/> vCard:N     _:b1 .
_:b1 vCard:Family "Smith" .
_:b1 vCard:Given  "Rebecca" .

<http://somewhere/JohnSmith/>    vCard:FN    "John Smith" .
<http://somewhere/JohnSmith/>    vCard:N     _:b2 .
_:b2 vCard:Family "Smith" .
_:b2 vCard:Given  "John"  .

<http://somewhere/SarahJones/>   vCard:FN    "Sarah Jones" .
<http://somewhere/SarahJones/>   vCard:N     _:b3 .
_:b3 vCard:Family  "Jones" .
_:b3 vCard:Given   "Sarah" .
```

Il est important de remarquer que ce sont les mêmes graphes RDF et que les triplets dans le graphe n'ont pas d'ordre particulier. Ils sont juste écrits en groupes liés ci-dessus pour la lecture par un humain - la machine ne s'en soucie pas.

II - Une première requête SPARQL

Dans cette section, on va jeter un œil à une simple première requête et montrer comment l'exécuter avec Jena.

II-A - Le « hello world » des requêtes

Le fichier « **q1.rq** » contient la requête suivante :

```
SELECT ?x
WHERE { ?x  <http://www.w3.org/2001/vcard-rdf/3.0#FN>  "John Smith" }
```

En exécutant cette requête avec l'application en ligne de commande :

```
---------------------------------
| x                             |
=================================
| <http://somewhere/JohnSmith/> |
---------------------------------
```

Ceci fonctionne par correspondance du modèle de triplets dans la clause WHERE sur les triplets du graphe RDF. Le prédicat et l'objet du triplet sont des valeurs fixées de sorte que le modèle va correspondre seulement aux triplets avec ces valeurs. Le sujet est une variable, et il n'y a pas d'autre restriction sur la variable. Le modèle correspond à n'importe quels triplets avec ces valeurs d'objet et de prédicat, et les fait correspondre avec les solutions pour x.

L'item inclus dans <> est une URI (précisément, c'est une IRI) et l'item inclus dans "" est littéral ordinaire. Juste comme Turtle, N3 ou N-triples, les littéraux typés sont écrits avec ^^ et les étiquettes de langues peuvent être ajoutées avec @.

?x est une variable appelée x. Le ? ne forme pas une partie du nom c'est pourquoi il n'apparaît pas dans le tableau de sortie.

Il n'y a qu'une correspondance. La requête renvoie la correspondance dans la variable de requête x. La sortie montrée a été obtenue en utilisant une des applications en ligne de commande d'ARQ.

II-B - Exécuter la requête

Il y a des **scripts d'assistance** dans les répertoires /bat et /bin de la distribution ARQ. Ils pourraient ne pas être présents dans la distribution de Jena. Vous devez vérifier l'existence de ces scripts avant de l'utiliser.

II-B-1 - Installation sous Windows

Définissez la variable d'environnement ARQROOT à l'emplacement de la distribution ARQ.

```
set ARQROOT=c:\MyProjects\ARQ
```

La distribution habituellement a le numéro de version dans le nom du répertoire.

Dans le répertoire d'ARQ, exécutez :

```
bat\sparql.bat --data=doc\Tutorial\vc-db-1.rdf --query=doc\Tutorial\q1.rq
```

Vous pouvez simplement mettre le répertoire /bat dans votre CLASSPATH ou les déplacer à l'extérieur du répertoire. Ils dépendent tous de ARQROOT.

II-B-2 - Scripts Bash pour Linux/Cygwin/Unix

Définissez la variable d'environnement ARQROOT à l'emplacement de la distribution ARQ.

```
export ARQROOT=$HOME/MyProjects/ARQ
```

La distribution habituellement a le numéro de version dans le nom du répertoire.

Dans le répertoire d'ARQ, exécutez :

```
bin/sparql --data=doc/Tutorial/vc-db-1.rdf --query=doc/Tutorial/q1.rq
```

Vous pouvez simplement mettre le répertoire /bat dans votre CLASSPATH ou les déplacer à l'extérieur du répertoire. Ils dépendent tous de ARQROOT.

Cygwin est un environnement de type Linux pour Windows.

II-B-3 - Utiliser directement les applications Java en ligne de commande

Vous aurez besoin de définir le CLASSPATH pour inclure *tous* les fichiers jar du dossier /lib d'ARQ.

Par exemple, sous Windows :

```
ARQdir\lib\antlr-2.7.5.jar;ARQdir\lib\arq-extra.jar;ARQdir\lib\arq.jar;
ARQdir\lib\commons-logging-1.1.jar;ARQdir\lib\concurrent.jar;ARQdir\lib\icu4j_3_4.jar;
ARQdir\lib\iri.jar;ARQdir\lib\jena.jar;ARQdir\lib\jenatest.jar;
ARQdir\lib\json.jar;ARQdir\lib\junit.jar;ARQdir\lib\log4j-1.2.12.jar;
ARQdir\lib\lucene-core-2.2.0.jar;ARQdir\lib\stax-api-1.0.jar;
ARQdir\lib\wstx-asl-3.0.0.jar;ARQdir\lib\xercesImpl.jar;ARQdir\lib\xml-apis.jar
```

où ARQdir est l'endroit où vous avez décompressé ARQ. Tout ceci doit être sur une seule ligne.

Le nom des fichiers jar quelquefois change et de nouveaux fichiers jar sont ajoutés - vérifiez cette liste avec votre version de ARQ.

Les commandes elles-mêmes sont dans le paquet arq.

III - Modèles de base

Cette section couvre les bases des motifs et les solutions, les principaux blocs constitutifs de requêtes SPARQL.

III-A - Les solutions

Les solutions d'une requête sont un ensemble de paires d'un nom de variable avec une valeur. Une requête SELECT expose directement les solutions (après un tri, une limite ou un décalage) comme le jeu de résultats, d'autres formes d'une requête utilisent les solutions pour faire un graphe. La solution est la manière de faire correspondre le modèle qui valorise les variables devant le prendre pour un modèle de correspondances.

Le premier exemple de requête n'avait qu'une seule solution. On change le modèle dans cette seconde requête : (**q-bp1.rq**) :

```
SELECT ?x ?fname
WHERE {?x  <http://www.w3.org/2001/vcard-rdf/3.0#FN>  ?fname}
```

Ceci a quatre solutions, une pour chaque triplet propriété nom de VCARD dans les données source :

```
-----------------------------------------------------
| x                                 | name          |
=====================================================
| <http://somewhere/RebeccaSmith/> | "Becky Smith" |
| <http://somewhere/SarahJones/>   | "Sarah Jones" |
| <http://somewhere/JohnSmith/>    | "John Smith"  |
| <http://somewhere/MattJones/>    | "Matt Jones"  |
-----------------------------------------------------
```

Jusqu'ici, avec des modèles de triplets et des modèles de base, chaque variable sera définie dans chaque solution. Les solutions pour une requête peuvent être considérées comme un tableau, mais dans le cas général, c'est un tableau où toutes les lignes n'auront pas une valeur pour chaque colonne. Toutes les solutions pour une requête SPARQL donnée ne doivent pas posséder de valeurs pour toutes les variables dans chaque solution comme nous le verrons plus tard.

III-B - Modèles de base

Un modèle de base est un ensemble de modèles de triplets. Il correspond lorsque les modèles de triplets correspondent à toutes les mêmes valeurs utilisées chaque fois que la variable avec le même nom est utilisée.

```
SELECT ?givenName
WHERE
  { ?y  <http://www.w3.org/2001/vcard-rdf/3.0#Family>  "Smith" .
    ?y  <http://www.w3.org/2001/vcard-rdf/3.0#Given>  ?givenName .
  }
```

Cette requête (**q-bp2.rq**) utilise deux modèles de triplets, chaque triplet finit par un point . (mais le point après le dernier peut être omis comme il l'était dans l'exemple de modèle d'un triplet). La variable y doit être la même pour chaque modèle de triplets correspondant. Les solutions sont :

```
-------------
| givenName |
=============
| "John"    |
| "Rebecca" |
-------------
```

III-B-1 - Qnames

Il y a un mécanisme raccourci pour écrire de longues URI utilisant des préfixes. La requête précédente est plus claire écrite comme la requête (**q-bp3.rq**) :

```
PREFIX vcard:      <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?givenName
WHERE
 { ?y vcard:Family "Smith" .
   ?y vcard:Given  ?givenName .
 }
```

Il y a un mécanisme de préfixation, les deux parties des URI, venant de la déclaration du préfixe et de la partie après le : dans le qname, sont concaténées. Ce n'est pas strictement ce qu'un qname XML est, mais il utilise la règle du RDF pour convertir un qname en URI par concaténation des parties.

III-B-2 - Nœuds anonymes

On change la requête juste un peu pour aussi retourner y (**q-bp4.rq**) :

```
PREFIX vcard:      <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?y ?givenName
WHERE
 { ?y vcard:Family "Smith" .
   ?y vcard:Given  ?givenName .
 }
```

et les nœuds anonymes apparaissent :

```
--------------------
| y    | givenName |
====================
| _:b0 | "John"    |
| _:b1 | "Rebecca" |
--------------------
```

ils apparaissent comme des qnames étranges commençant par _:. Ce n'est pas le label interne pour le nœud anonyme, c'est l'affichage de ARQ qui leur a affecté les _:b0 et _:b1 pour montrer quand deux nœuds anonymes sont identiques. Ici ils sont différents. Cela ne révèle pas le label interne utilisé pour le nœud anonyme bien qu'il soit disponible lorsque vous utilisez l'API Java.

IV - Les filtres

La correspondance de graphes permet aux modèles dans le graphe d'être trouvés. Cette section décrit comment les valeurs dans une solution peuvent être restreintes. Il y a de nombreuses comparaisons possibles, on couvrira juste deux cas ici.

IV-A - Correspondance de chaînes de caractères

SPARQL fournit une opération pour tester les chaînes de caractères, basée sur les expressions régulières. Cela inclut la capacité de demander des tests du style SQL LIKE, bien que la syntaxe de l'expression régulière soit différente du SQL :

```
FILTER regex(?x, "pattern" [, "flags"])
```

L'argument flags est optionnel. Le drapeau i signifie qu'une correspondance insensible à la casse est effectuée.

L'exemple de requête (**q-f1.rq**) trouve les noms donnés avec un r ou un R à l'intérieur.

```
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?g
WHERE
{ ?y vcard:Given ?g .
  FILTER regex(?g, "r", "i") }
```

avec les résultats :

```
-------------
| g     |
=============
| "Rebecca" |
| "Sarah"   |
-------------
```

Le langage d'expressions régulières est le même que le **langage d'expression régulière de XQuery**, qui est une version codifiée de celle trouvée dans Perl.

IV-B - Tester les valeurs

Il y a des cas où l'application veut filtrer sur la valeur d'une variable. Dans le fichier de données **vc-db-2.rdf**, nous avons ajouté un champ supplémentaire pour l'âge. L'âge n'est pas défini par le schéma vCard, donc nous avons créé une nouvelle propriété dans le but de ce tutoriel. RDF autorise un tel mélange de différentes définitions d'information, parce que les URI sont uniques. Notez aussi que la valeur de la propriété info:age est typée.

Dans cet extrait de données, nous montrons la valeur typée. Elle peut être écrite en dur, 23.

```
<http://somewhere/RebeccaSmith/>
    info:age "23"^^xsd:integer ;
    vCard:FN "Becky Smith" ;
    vCard:N [ vCard:Family "Smith" ;
              vCard:Given  "Rebecca" ] .
```

Ainsi, une requête (**q-f2.rq**) pour trouver le nom des gens qui sont plus vieux que vingt-quatre ans :

```
PREFIX info: <http://somewhere/peopleInfo#>

SELECT ?resource
WHERE
  {
    ?resource info:age ?age .
    FILTER (?age >= 24)
  }
```

L'expression arithmétique doit être entre parenthèses. La seule solution est :

```
---------------------------------
| resource                      |
=================================
| <http://somewhere/JohnSmith/> |
---------------------------------
```

Juste un correspond, résultant en l'URI de John Smith. En transformant cette requête pour demander que ceux de moins de vingt-quatre ans donnent également une correspondance pour Rebecca Smith. Rien à propos des Jones.

La base de données ne contient aucune information sur l'âge des Jones : il n'y a pas de propriétés info:age sur ces vCard donc la variable age n'a pas reçu de valeur et n'a pas été testée par le filtre.

V - Informations optionnelles

Le RDF est représenté sous forme de données semi-structurées ainsi SPARQL à la capacité de requêter des données, mais de ne pas faire échouer la requête quand ces données n'existent pas. La requête utilise une partie optionnelle pour étendre l'information trouvée dans une solution de requête, mais pour retourner les informations non optionnelles de toute façon.

V-A - OPTIONAL

Cette requête (**q-opt1.rq**) retourne le nom d'une personne et aussi son âge si cette portion d'information est disponible.

```
PREFIX info:    <http://somewhere/peopleInfo#>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name ?age
WHERE
{
    ?person vcard:FN  ?name .
    OPTIONAL { ?person info:age ?age }
}
```

Deux des quatre personnes dans les données (**vc-db-2.rdf**) possèdent la propriété age ainsi deux des solutions de la requête ont cette information. Cependant, parce que le modèle de triplets pour l'âge est optionnel, il y a un modèle de solutions pour les personnes qui n'ont pas l'information age.

```
------------------------
| name          | age  |
========================
| "Becky Smith" | 23   |
| "Sarah Jones" |      |
| "John Smith"  | 25   |
| "Matt Jones"  |      |
------------------------
```

Si la clause optionnelle n'avait pas été là, aucune information sur l'âge n'aurait été récupérée. Si le modèle de triplets avait été inclus, mais pas optionnel alors nous aurions eu la requête (**q-opt2.rq**) :

```
PREFIX info:   <http://somewhere/peopleInfo#>
PREFIX vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name ?age
WHERE
{
    ?person vcard:FN  ?name .
    ?person info:age ?age .
}
```

avec seulement deux solutions :

```
-----------------------
| name          | age |
=======================
| "Becky Smith" | 23  |
| "John Smith"  | 25  |
-----------------------
```

parce que la propriété info:age doit maintenant être présente dans la solution.

V-B - OPTIONAL et FILTER

OPTIONAL est un opérateur binaire qui combine deux modèles de graphes. Le modèle optionnel est n'importe quel modèle du groupe et peut impliquer tous les types de modèles SPARQL. Si le groupe correspond, la solution est étendue, sinon, la solution d'origine est donnée (**q-opt3.rq**).

```
PREFIX info:        <http://somewhere/peopleInfo#>
PREFIX vcard:      <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name ?age
WHERE
{
    ?person vcard:FN  ?name .
    OPTIONAL { ?person info:age ?age . FILTER ( ?age > 24 ) }
}
```

Ainsi, si nous filtrons sur les âges supérieurs à vingt-quatre ans dans la partie optionnelle, nous obtiendrons encore quatre solutions (à partir du modèle vcard:FN), mais seulement pour obtenir les âges s'ils passent le test :

```
-----------------------
| name          | age |
=======================
| "Becky Smith" |     |
| "Sarah Jones" |     |
| "John Smith"  | 25  |
| "Matt Jones"  |     |
-----------------------
```

Pas d'âge inclus pour « Becky Smith », parce qu'il est inférieur à vingt-quatre.

Si la condition de filtre est déplacée hors de la partie optionnelle, alors elle peut influer sur le nombre de solutions, mais il peut être nécessaire de faire le filtre plus compliqué pour permettre à la variable age de ne pas y être liée (**q-opt4.rq**).

```
PREFIX info:        <http://somewhere/peopleInfo#>
PREFIX vcard:      <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name ?age
```

```
WHERE
{
    ?person vcard:FN  ?name .
    OPTIONAL { ?person info:age ?age . }
    FILTER ( !bound(?age) || ?age > 24 )
}
```

si une solution possède une variable age, alors elle doit être plus grande que vingt-quatre. Elle peut aussi être non liée. Il y a maintenant trois solutions :

```
----------------------
| name          | age |
======================
| "Sarah Jones" |     |
| "John Smith"  | 25  |
| "Matt Jones"  |     |
----------------------
```

Évaluer une expression qui possède des variables non liées où une liée était attendue provoque une exception d'évaluation et l'expression entière échoue.

V-C - OPTIONAL et requêtes dépendant de l'ordre

Une chose à laquelle faire attention est de se servir de la même variable dans deux clauses optionnelles ou plus (et ainsi pas dans quelques modèles de base) :

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name
WHERE
{
  ?x a foaf:Person .
  OPTIONAL { ?x foaf:name ?name }
  OPTIONAL { ?x vCard:FN  ?name }
}
```

Si le premier OPTIONAL lie ?name et ?x à quelques valeurs, le second OPTIONAL est une tentative de faire correspondre les autres triplets (?x et ?name possèdent des valeurs). Si le premier OPTIONAL n'a pas fait correspondre la partie optionnelle, alors la seconde est une tentative de faire correspondre son triplet avec deux variables.

VI - Alternatives dans un modèle

Une autre manière de traiter avec les données semi-structurées est d'effectuer une requête pour une des possibilités. Cette section couvre le modèle UNION, où l'une des possibilités est essayée.

VI-A - UNION - deux façons pour les mêmes données

Les vocabulaires vCard et FOAF ont des propriétés pour le nom des personnes. Dans vCard, c'est vCard:FN, le « nom formaté », et dans FOAF, c'est foaf:name. Dans cette section, nous nous pencherons sur un petit ensemble de données où les noms des personnes peuvent être donnés soit par le vocabulaire FOAF soit par le vocabulaire vCard.

Supposons que nous ayons **un graphe RDF** qui contient des informations de nom en utilisant les vocabulaires vCard et FOAF.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
```

```
_:a foaf:name   "Matt Jones" .

_:b foaf:name   "Sarah Jones" .

_:c vcard:FN    "Becky Smith" .

_:d vcard:FN    "John Smith" .
```

Une requête pour accéder aux informations du nom, quand il peut être dans les deux formes, pourrait être (**q-union1.rq**) :

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name
WHERE
{
   { [] foaf:name ?name } UNION { [] vCard:FN ?name }
}
```

Cela renvoie les résultats :

```
-----------------
| name          |
=================
| "Matt Jones"  |
| "Sarah Jones" |
| "Becky Smith" |
| "John Smith"  |
-----------------
```

Ça n'a pas d'importance quelle forme de l'expression était utilisée pour le nom, la variable ?name est définie. Ceci peut être réalisé en utilisant un FILTER comme cette requête (**q-union-1alt.rq**) le montre :

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name
WHERE
{
  [] ?p ?name
  FILTER ( ?p = foaf:name || ?p = vCard:FN )
}
```

on teste si la propriété est une URI ou autre chose. Les solutions ne peuvent pas sortir dans le même ordre. La première forme est susceptible d'être plus rapide, en fonction des données et du stockage utilisé, parce que la seconde forme peut obtenir tous les triplets du graphe qui correspondent au modèle des triplets avec des variables non liées (ou des nœuds anonymes) dans chaque emplacement, alors on teste chaque ?p pour voir s'il correspond à l'une des valeurs. Cela dépendra de la sophistication de l'optimiseur de requêtes de savoir s'il repère les requêtes qu'il peut effectuer de manière plus efficace et est capable de passer par les contraintes autant que par la couche de stockage.

VI-B - Union - se rappeler où les données ont été trouvées

L'exemple ci-dessous utilise la même variable dans chaque branche. Si différentes variables sont utilisées, l'application peut découvrir quel sous-modèle a causé la correspondance (**q-union2.rq**) :

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name1 ?name2
WHERE
{
```

```
   { [] foaf:name ?name1 } UNION { [] vCard:FN ?name2 }
}

---------------------------------
| name1         | name2         |
=================================
| "Matt Jones"  |               |
| "Sarah Jones" |               |
|               | "Becky Smith" |
|               | "John Smith"  |
---------------------------------
```

Cette deuxième requête a conservé des informations d'où le nom de la personne provenait en attribuant le nom à différentes variables.

VI-C - OPTIONAL et UNION

Dans la pratique, OPTIONAL est plus commun que UNION, mais ils ont tous deux leur utilisation. OPTIONAL est utile pour augmenter les solutions trouvées, UNION est utile pour concaténer les solutions à partir de deux possibilités. Ils ne retournent pas nécessairement les informations de la même manière :

Requête (**q-union3.rq**) :

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name1 ?name2
WHERE
{
  ?x a foaf:Person
  OPTIONAL { ?x  foaf:name  ?name1 }
  OPTIONAL { ?x  vCard:FN   ?name2 }
}

---------------------------------
| name1         | name2         |
=================================
| "Matt Jones"  |               |
| "Sarah Jones" |               |
|               | "Becky Smith" |
|               | "John Smith"  |
---------------------------------
```

Mais méfiez-vous d'utiliser ?name dans chaque OPTIONAL parce que c'est une requête dépendant de l'ordre.

VII - Les jeux de données

Cette section couvre les jeux de données RDF, un jeu de données RDF est l'unité qui est interrogée par une requête SPARQL. Il s'agit d'un graphe par défaut, et un nombre de graphes nommés.

VII-A - Requêter les jeux de données

L'opération de correspondance du graphe (**modèles de base**, **OPTIONAL** et **UNION**) fonctionne sur un graphe RDF. Cela commence en étant le graphe par défaut du jeu de données, mais il peut être modifié par le mot-clé GRAPH.

```
GRAPH uri { ... pattern ... }
GRAPH var { ... pattern ... }
```

Si une URI est donnée, le modèle sera identifié sur le graphe dans le jeu de données avec ce nom, s'il n'y en a pas, la clause GRAPH ne correspond à rien.

Si une variable est donnée, tous les graphes nommés (pas le graphe par défaut) sont essayés. La variable peut être utilisée ailleurs de sorte que si, lors de l'exécution, sa valeur est déjà connue pour une solution, seul le graphe nommé spécifique est essayé.

VII-A-1 - Exemple de données

Un jeu de données RDF peut prendre une variété de formes. Deux configurations sont communes pour avoir le graphe par défaut comme étant l'union (la fusion RDF) de tous les graphes nommés et pour avoir le graphe par défaut comme un inventaire des graphes nommés (d'où ils venaient, où ils ont été lus, etc.) Il n'y a pas de limites : un graphe peut être inclus deux fois sous différents noms, ou quelques graphes peuvent partager les triplets avec les autres.

Dans les exemples ci-dessous, nous utiliserons le jeu de données suivant qui pourrait se produire pour un agrégateur RDF des détails du livre :

Le graphe par défaut (**ds-dft.ttl**) :

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<ds-ng-1.ttl> dc:date "2005-07-14T03:18:56+0100"^^xsd:dateTime .
<ds-ng-2.ttl> dc:date "2005-09-22T05:53:05+0100"^^xsd:dateTime .
```

Le graphe nommé (**ds-ng-1.ttl**) :

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .

[] dc:title "Harry Potter and the Philospher's Stone" .
[] dc:title "Harry Potter and the Chamber of Secrets" .
```

Le graphe nommé (**ds-ng-2.ttl**) :

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .

[] dc:title "Harry Potter and the Sorcerer's Stone" .
[] dc:title "Harry Potter and the Chamber of Secrets" .
```

Maintenant, nous avons deux petits graphes décrivant quelques livres, et nous avons un graphe par défaut qui enregistre quand ces graphes ont été lus pour la dernière fois.

Les requêtes peuvent être exécutées via la ligne de commande de l'application (ce sera sur une seule ligne) :

```
java -cp ... arq.sparql
    --graph ds-dft.ttl --namedgraph ds-ng-1.ttl --namedgraph ds-ng-2.ttl
    --query query file
```

Les jeux de données ne doivent pas être créés juste pour la durée de vie de la requête. Ils peuvent être créés et stockés dans une base de données, comme ce serait plus habituel pour une application agrégateur.

VII-A-2 - Accéder au jeu de données

Le premier exemple accède juste au graphe par défaut (**q-ds-1.rq**) :

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <.>

SELECT *
{ ?s ?p ?o }
```

(Le PREFIX : <.> permet juste d'aider à formater la sortie.

```
-------------------------------------------------------------------
| s            | p       | o                                      |
===================================================================
| :ds-ng-2.ttl | dc:date | "2005-09-22T05:53:05+01:00"^^xsd:dateTime |
| :ds-ng-1.ttl | dc:date | "2005-07-14T03:18:56+01:00"^^xsd:dateTime |
-------------------------------------------------------------------
```

Voici seulement le graphe par défaut, rien venant des graphes nommés, parce qu'ils ne sont pas interrogés sauf si explicitement indiqués par GRAPH.

Nous pouvons interroger tous les triplets en interrogeant le graphe par défaut et les graphes nommés (**q-ds-2.rq**) :

```
PREFIX  xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX  dc:     <http://purl.org/dc/elements/1.1/>
PREFIX  :       <.>

SELECT *
{
    { ?s ?p ?o } UNION { GRAPH ?g { ?s ?p ?o } }
}
```

donne :

```
-------------------------------------------------------------------------------------
| s            | p        | o                                          | g           |
=====================================================================================
| :ds-ng-2.ttl | dc:date  | "2005-09-22T05:53:05+01:00"^^xsd:dateTime  |             |
| :ds-ng-1.ttl | dc:date  | "2005-07-14T03:18:56+01:00"^^xsd:dateTime  |             |
| _:b0         | dc:title | "Harry Potter and the Sorcerer's Stone"    | :ds-ng-2.ttl |
| _:b1         | dc:title | "Harry Potter and the Chamber of Secrets"  | :ds-ng-2.ttl |
| _:b2         | dc:title | "Harry Potter and the Chamber of Secrets"  | :ds-ng-1.ttl |
| _:b3         | dc:title | "Harry Potter and the Philospher's Stone"  | :ds-ng-1.ttl |
-------------------------------------------------------------------------------------
```

VII-A-3 - Requêter un graphe spécifique

Si l'application connaît le nom de graphe, elle peut directement demander une requête comme trouver tous les titres dans un graphe donné (**q-ds-3.rq**) :

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <.>

SELECT ?title
{
  GRAPH :ds-ng-2.ttl
    { ?b dc:title ?title }
}
```

Résultats :

```
-----------------------------------------
| title                                 |
=========================================
| "Harry Potter and the Sorcerer's Stone"   |
| "Harry Potter and the Chamber of Secrets" |
-----------------------------------------
```

VII-A-4 - Requêter pour trouver les données à partir de graphes qui correspondent à un modèle

Les noms des graphes requêtables peuvent être déterminés avec la requête elle-même. Le même processus s'applique pour les variables si elles font partie d'un modèle de graphes ou d'une forme GRAPH. La requête ci-dessous (**q-ds-4.rq**) fixe une condition sur la variable utilisée pour sélectionner les graphes nommés, basée sur des informations dans le graphe par défaut.

```
PREFIX  xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX  dc:     <http://purl.org/dc/elements/1.1/>
PREFIX  :       <.>

SELECT ?date ?title
{
  ?g dc:date ?date . FILTER (?date > "2005-08-01T00:00:00Z"^^xsd:dateTime )
  GRAPH ?g
      { ?b dc:title ?title }
}
```

Les résultats de l'exécution de cette requête sur le jeu de données de l'exemple sont les titres dans l'un des graphes, le premier avec la date au plus tard le 1er août 2005.

```
-------------------------------------------------------------------------------------
| date                                      | title                                   |
=====================================================================================
| "2005-09-22T05:53:05+01:00"^^xsd:dateTime | "Harry Potter and the Sorcerer's Stone"   |
| "2005-09-22T05:53:05+01:00"^^xsd:dateTime | "Harry Potter and the Chamber of Secrets" |
-------------------------------------------------------------------------------------
```

VII-B - Décrire les jeux de données RDF - FROM et FROM NAMED

Une exécution de la requête peut donner le jeu de données lorsque l'objet d'exécution est construit ou il peut être décrit dans la requête elle-même. Lorsque les détails sont sur la ligne de commande, un jeu de données temporaires est créé, mais une application peut créer des jeux de données et puis les utiliser dans de nombreuses requêtes.

Lorsqu'il est décrit dans la requête, FROM url est utilisé pour identifier le contenu dans le graphe par défaut. Il peut y avoir plus d'une clause FROM et le graphe par défaut est le résultat de la lecture de chaque fichier dans le graphe par défaut. C'est la fusion RDF des graphes individuels.

Ne soyez pas troublés par le fait que le graphe par défaut soit décrit par une ou plusieurs URL dans les clauses FROM. C'est là où les données sont lues, pas le nom du graphe. Comme plusieurs clauses FROM peuvent être données, les données peuvent être lues à partir de plusieurs endroits, mais aucun d'entre eux ne devient le nom du graphe.

FROM NAMED url est utilisé pour identifier un graphe nommé. Le graphe est donné par le nom de l'url et les données sont lues à partir de cet emplacement. De multiples clauses FROM NAMED provoquent l'ajout de plusieurs graphes au jeu de données.

Notez que les graphes sont chargés avec le FileManager de Jena qui peut inclure la capacité de fournir des endroits différents pour les fichiers. Par exemple, la requête peut avoir FROM NAMED <http://example/data>, et les données sont réellement lues à partir de file:local.rdf. Le nom du graphe sera <http://example/data>, comme dans la requête.

Par exemple, la requête pour trouver tous les triplets dans les graphes par défaut et dans les graphes nommés peut être écrite comme ceci (**q-ds-5.rq**) :

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
PREFIX :    <.>
```

```
SELECT *
FROM       <ds-dft.ttl>
FROM NAMED <ds-ng-1.ttl>
FROM NAMED <ds-ng-2.ttl>
{
   { ?s ?p ?o } UNION { GRAPH ?g { ?s ?p ?o } }
}
```

VIII - Produire des ensembles de résultats

SPARQL possède quatre formes de résultat :

- SELECT : retourne un tableau de résultats ;
- CONSTRUCT : retourne un graphe RDF, basé sur un modèle dans la requête ;
- DESCRIBE : retourne un graphe RDF, basé sur ce que le processeur de requêtes est configuré pour renvoyer ;
- ASK : pose une requête booléenne.

La forme SELECT retourne directement un tableau de solutions comme un ensemble de résultats, tandis que DESCRIBE et CONSTRUCT utilisent les résultats de l'appariement pour construire des graphes RDF.

VIII-A - Modificateurs de solutions

La correspondance de modèles produit un ensemble de solutions. Cet ensemble peut être modifié de diverses manières :

- Projection : ne garde que les variables sélectionnées ;
- OFFSET / LIMIT : couper le nombre de solutions (mieux utiliser avec ORDER BY) ;
- ORDER BY : trier les résultats ;
- DISTINCT : ne rendre qu'une seule ligne pour une combinaison de variable et de valeur.

Les modificateurs de solutions OFFSET / LIMIT et ORDER BY s'appliquent toujours à toutes les formes de résultats.

VIII-A-1 - OFFSET et LIMIT

Un ensemble de solutions peut être abrégé en spécifiant le décalage (index de début) et la limite (le nombre de solutions à retourner). Utiliser LIMIT seul peut être utile pour assurer que de trop nombreuses solutions ne soient retournées, pour restreindre l'effet de quelques situations inattendues. LIMIT et OFFSET peuvent être utilisés en addition avec le tri pour prendre un intervalle défini par les solutions trouvées.

VIII-A-2 - ORDER BY

Les solutions SPARQL sont triées par l'expression, incluant les fonctions personnalisées.

```
ORDER BY ?x ?y

ORDER BY DESC(?x)

ORDER BY x:func(?x)     # condition de tri personnalisé
```

VIII-A-3 - DISTINCT

La forme du résultat de SELECT peut prendre le modificateur DISTINCT qui assure qu'il n'y a pas deux solutions retournées qui sont les mêmes, cela a lieu après la projection pour les variables interrogées.

VIII-B - SELECT

La forme du résultat de SELECT est une projection, avec DISTINCT appliqué, de l'ensemble des solutions. SELECT identifie quelles variables nommées sont dans l'ensemble de résultats. « * » signifiant « toutes les variables nommées » (les nœuds anonymes dans la requête agissent comme des variables d'appariement, mais ne sont jamais retournés).

VIII-C - CONSTRUCT

CONSTRUCT construit un graphe RDF basé sur un modèle de graphes. Le modèle de graphes peut avoir des variables qui sont liées par une clause WHERE. L'effet est de calculer le fragment de graphe, donné par le modèle, pour chaque solution de la clause WHERE, après la prise en compte des modificateurs de solution. Les fragments de graphe, un par solution, sont fusionnés en un seul graphe RDF qui est le résultat.

Tous les nœuds anonymes explicitement mentionnés dans le modèle de graphes sont créés de nouveau pour chaque fois que le modèle est utilisé pour une solution.

VIII-D - DESCRIBE

La forme CONSTRUCT prend un modèle d'application pour les résultats de graphe. La forme DESCRIBE crée également un graphe, mais la forme de ce graphe est fournie par le processeur de requêtes, pas par l'application. Pour chaque URI trouvée, ou explicitement mentionnée dans la clause DESCRIBE, le processeur de requêtes devrait fournir un fragment utile de RDF, comme tous les détails connus d'un livre. ARQ permet aux gestionnaires de descriptions spécifiques au domaine d'être écrits.

VIII-E - ASK

Le formulaire de résultat ASK retourne un booléen, vrai pour un motif qui correspond, faux sinon.

IX - L'article original

Cet article est la traduction de **SPARQL Tutorial**.

X - Remerciements

Merci à **Claude Leloup**, **jpcheck** et **jacques_jean** pour leur relecture orthographique.