Université des Sciences de Montpellier
Master 2 Semestre 1
Unité d'Enseignement FMIN306

Raja
Bases de données distribuées
Livraison

2 janvier 2011

Audrey Novak
Romain Maneschi
Jonathan Fhal
Aloys Urbain

Table des matières

1 Introduction

Ce livrable fait suite au livrable de conception et tant à expliquer toutes les différences de conceptions. De plus, nous vous expliquerons les problèmes rencontrés et les solutions choisis. Enfin nous inscrirons dans ce livrable ce que nous n'avons pu faire par manque de temps ou encore ce qui serait possible de faire pour une future version.

2 Différences

2.1 getMetaInfo() déplacée

Lorque nous initialisons le système, nous avons décidé de récupérer les méta-informations des bases de données par le biais des traducteurs (Mysql, Oracle et Postgres) plutôt que par les traducteurs N3. En effet aucune requête en HQL ne nous a parmis de récupérer les informations importantes telles que le nom de la base de données, les tables et les colonnes. Alors qu'en sql normal nous avons l'habitude de le faire, ainsi nous sommes passés par le sql "traditionnel" pour récupérer ces informations. Nous créons ensuite un modèle ontholo-gique, contenant les classes nécessaires sur les méta-informations, ces classes sont préfixées de métaInfo pour pouvoir les récupérer facilement par la suite. La navigation dans les différents adapters, pour arriver dans les différentes bases de données, en est donc grandement simplifiée.

2.2 Suppression isQueryMatching

La fonction isQueryMatching devait nous permettre de pouvoir trouver les sous-adaptateurs dans lesquels la query doit se propager. Mais nous nous sommes rendu compte que cette fonction effectuait exactement le même travail que execute(query). Du coup, il nous a paru évident de faire le travail qu'une fois et de vérifier. Si le retour est null alors on était dans un mauvais sous-adaptateur.

2.3 Model plûtot que ResultSet

Un des points techniques de Jena nous a obligé à retourner des Modèles plutôt que des ResultSet. En effet, lorsque nous retournons des resultSet, il devient vite très difficile de pouvoir les ajouter dans un modèle. Hors D2RQ permet d'effectuer des requêtes DESCRIBE permettant de récupérer un modèle contenant beacoup d'informations sur les noeud RDF. Ainsi nous n'avions qu'à transformer nos requêtes SELECT en requête DESCRIBE. Puis ajouter le modèle du sous-adaptateur dans le modèle de l'adaptateur pour former un modèle global comprenant toutes les informations. Mais cette technique a un inconvénient, il faut que tous les modèles connaissent les préfixes des sous-modèles sans quoi les noeuds RDF sont mal interprétés.

3 Problèmes

3.1 EquivalentClass

Ce point fût le plus dur à résoudre de tout le projet. En effet, tout notre système repose sur le fait que les adaptateurs représentent des ontologies et que les adaptateurs terminaux représentent des graphes RDF (par le biais d'une base de données et d'un fichier N3). Les ontologies de plus haut niveaux ont la responsabilités de faire correspondre les classes des différentes sous-adaptateurs.

Ce mécanisme devait être implémenté très facilement par des équivalences entre classes. Mais ce mécanisme n'a jamais voulu fonctionner. Il faut en fait récupérer tous les individus des classes équivalentes pour les déclarer sameAs, autrement dit pour les lier explicitement. Dans notre cas, les individus se trouvent dans les bases de données donc on ne peut pas le faire directement dans les ontologies.

Première solution, réimplémenter un système d'équivalence. Pour nous deux classes équivalentes signifient qu'une recherche sur l'une des classes doit aussi être éfffectuée sur l'autre. Nous avons donc créée une fonction qui pour un élément de triplet d'une recherche HQL vérifie si elle a une classe équivalente et si oui elle ajoute le résultat au modèle ontologique que l'on retournera. Mais cette solution ne suffit pas puisqu'il faut aussi lier tous les individus entre eux.

Deuxième solution, aprés avoir chargé tous les résultats dans notre modèle ontologique nous devons donc récupérer tous les individus de toutes les classes équivalentes pour les positionner à same. Nous avons fait le choix de dire que pour que ces individus puissent être à same, il faut absolument que leur valeur soit identique (indépendemment de leur préfixe).

3.2 Meta-infos en variable plûtot qu'à chaque fois

Un autre problème technique qui a apparu très rapidement, fut le fait qu'oracle n'accepte pas beaucoup de pointeur en lecture. Hors lorsqu'on lance une requête aussi simple que les nom des maladies, le nombre de requête dépasse vite ce fameux pointeur puisqu'à chaque fois qu'on appel getMetaInfo() d'un sous-adaptateur appel à son tour celui de son sous-adaptateur et fini par ouvrir un pointeur sur la base. Pour régler ce problème, il existe différentes solutions :

- Fermer les requêtes, cette solution ne peut pas être utilisée avec Jena car le fait de fermer une requête libère la mémoire de tout le modèle, hors nous avons besoins de le garder.
- Créer un pool de connexion pour réutiliser les pointeurs déjà placés. Cette solution n'a pas été possible car nous n'avons pas accés au connexions de D2RQ.
- Garder le modèle des bases de données une fois celui-ci chargé. Cette solution est bien entendue celle retenue. Nous ne voulions pas faire cela au début car nous voulions que notre système s'adapte tout seul au changement de base de données, mais cela n'a pas été possible.

4 Futur

4.1 Update

Par manque de temps, nous n'avons pas pu implémenter le fait de mettre à jour des données. Pour pouvoir le faire, il nous aurait fallu modifier notre parsage de requête (transformation de la requête sous forme de String en Query) pour que l'on puisse rajouter le nom des colonnes à mettre à jour. En effet, en SQL, il y a 2 syntaxes pour un UPDATE :

- UPDATE FROM table VALUES(valeur...)
- UPDATE FROM table (colonne...) VALUES(valeur...)

Avec ces informations, nous aurions pu savoir vers quelle table il faut diriger la requête.

4.2 Raisonneur

Un raisonneur aurait permis de diminuer la complexité algorithmique de notre système. En effet, lorsque vous entrez une requête, nous ne pouvons pas connaitre la granularité de votre recherche. Par exemple si vous parlez de maladie nous ne pouvons pas savoir si vous parlez de la classe ou des entités de la maladie. Nous sommes donc obligés d'éffectuer la recherche dans tous les cas possibles. Nous devons donc lancer 3 fois (conformément au triplet RDF) toute la recherche : à gauche, au centre et à droite pour ne pas éléminer de réponses possibles. Par la suite pour éliminer les réponses non pertinentes nous avons fait le choix de relancer la requête d'entrée sur le schéma ontologique global.

4.3 InterfaceGraphique

Une interface graphique pour visualiser les ontologies aurait pû être facilement créée grâce au logiciel Protégé qui permet d'ajouter à son propre logiciel un visualisateur d'ontologie. Nous avons effectué pas mal de recherche dans ce sens mais le manque de temps ne nous a pas permis d'implémenter cette solution.

5 Conclusion

Malgrès quelques fonctions manquantes notre système permet de lancer des recherches parmis autant de base de donnée que l'on souhaite. Pour cela il suffit de configurer un seul fichier xml et de le renseigner au serveur. Sans oublier de créer ses ontologies pour donner la hiérarchie des données au système. Par la suite on peut insérer, supprimer et rechercher tout ce que l'on veut sur son système de données réparti.

Vous pouvez retrouver ce logiciel sur http ://code.google.com/p/raja. Pour le télécharger il vous suffit d'entrer "svn checkout http ://raja.googlecode.com/svn/trunk/ raja-read-only" dans un client SVN.