OntoQL, un langage d'exploitation des bases de données à base ontologique

Stéphane Jean

▶ To cite this version:

Stéphane Jean. OntoQL, un langage d'exploitation des bases de données à base ontologique. Interface homme-machine [cs.HC]. Université de Poitiers, 2007. Français. NNT : . tel-00201777

HAL Id: tel-00201777

https://theses.hal.science/tel-00201777

Submitted on 2 Jan 2008

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.

ENSMA : Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
LISI : Laboratoire d'Informatique Scientifique et Industrielle

THESE

pour l'obtention du Grade de

DOCTEUR DE L'UNIVERSITE DE POITIERS

(Faculté des Sciences Fondamentales et Appliquées)

(Diplôme National — Arrêté du 7 août 2006)

Ecole Doctorale : Science Pour l'Ingénieur
Secteur de Recherche : INFORMATIQUE ET APPLICATION

Présentée par :

Stéphane JEAN

****

OntoQL, un langage d'exploitation des bases de données à base ontologique

****

Directeurs de Thèse : **Yamine AIT-AMEUR et Guy PIERRA**

****

Soutenue le 5 décembre 2007
devant la Comission d'Examen

****

JURY

Rapporteurs :	**Witold LITWIN**	Professeur, Université Paris Dauphine, Paris
	Aris M. OUKSEL	Professeur, Université de l'Illinois, Chicago, USA
	Michel SCHNEIDER	Professeur, Université Blaise Pascal, Clermont-Ferrand
Examinateurs :	**Yamine AIT-AMEUR**	Professeur, ENSMA, Futuroscope
	Olivier CORBY	Chargé de recherche, INRIA, Sophia Antipolis
	Guy PIERRA	Professeur, ENSMA, Futuroscope

Merci à

Guy Pierra, à la fois directeur du LISI et mon directeur de thèse, pour m'avoir accueilli au sein de son laboratoire et pour m'avoir apporté ses précieuses lumières tout au long de cette thèse. Je le remercie pour la partie importante des contributions présentées dans ce mémoire qui proviennent de sa perspicacité.

Yamine Aït-Ameur, mon directeur de thèse, pour m'avoir guidé jour après jour pendant ces trois années de thèse et surtout pour m'avoir appris à mener par moi-même un travail de recherche. Je le remercie d'avoir eu la gentillesse de me transmettre un peu de son savoir-faire et de son savoir-être.

Witold Litwin, **Aris M. Ouksel** et **Michel Schneider** pour m'avoir fait l'honneur d'être rapporteurs de cette thèse. Je leur suis très reconnaissant d'avoir accepté cette lourde tâche.

Olivier Corby pour avoir accepté d'être membre du jury en tant qu'examinateur. Je suis très honoré de l'intérêt qu'il a porté à mes travaux.

Hondjack Dehainsala et **Dung Xuan Nguyen** pour l'excellent travail qu'ils ont réalisé pendant leur thèse. Sans leurs efforts, cette thèse n'aurait tout simplement pas existée.

Ladjel Bellatreche pour son amitié, son enthousiasme et sa passion de la recherche qui ont été pour moi une source de motivation et d'inspiration pendant cette thèse.

Frédéric Carreau et **Claudine Rault** pour toutes les tâches ingrates qu'ils ont réalisées pour moi. Leur aide a grandement facilité mon travail pendant ces trois années.

Tous les membres de l'équipe de football du LISI et plus particulièrement **Jean-Claude**, **Loé**, **Ahmed**, **Idir**, **Nabil**, **Jean-Marc**, **Jérome**, **Mickaël** et **Frédéric** pour m'avoir permis de m'aérer l'esprit dans une très bonne ambiance. J'ai été très heureux d'avoir pu être le capitaine d'une aussi brillante équipe.

Tous les autres membres du LISI (**Dago**, **Eric**, **Tex**, **Pascal**, **Manu**, **Stéphane**, **Nicolas**, **Chimène**, **Karim**, **Sybille**, **Kamel**, **Youcef**, **Nadjet**, **Michaël**, **David**, **Michel**, **Laurent**, **Annie**, **Dominique**, **Patrick**, etc.) pour tous les bons moments passés en leur compagnie.

Ma future femme **Sarah**, **mon père**, **ma mère** et **mon frère** ainsi que mes amis d'enfance **Bertrand**, **Cédric** et **Christophe** qui sont des rochers sur lesquels j'ai pu m'accrocher dans les moments difficiles de cette thèse. Je leur dédie cette thèse.

A mes rochers,
Sarah,
mon père, ma mère et mon frère,
mes amis d’enfance Bertrand, Cédric et Christophe.

Table des matières

Introduction **1**

Partie I De la nécessité d'un nouveau langage d'exploitation des bases de données à base ontologique

Chapitre 1 Ontologies et bases de données à base ontologique **7**

1 Introduction 9

2 Les ontologies de domaine dans une perspective d'exploitation de bases de données 10

2.1 Spécificité d'une ontologie de domaine comme modèle d'un domaine 10

2.2 Utilisation des ontologies dans le domaine des bases de données 14

2.3 Utilisation des ontologies dans d'autres domaines 15

2.4 Une taxonomie des ontologies de domaine 17

2.5 Relation entre les différentes catégories d'ontologies : le modèle en oignon . 21

2.6 Liens entre le modèle en oignon et les bases de données 24

2.7 Synthèse de notre interprétation des ontologies de domaine 25

3 Les modèles d'ontologies 25

3.1 Modèles d'ontologies orientés vers la définition de OCC 25
3.2 Modèles d'ontologies orientés vers la définition de OCNC 32
3.3 Traitements pour les OL 37
3.4 Noyau commun aux modèles d'ontologies 38
3.5 Synthèse sur les modèles d'ontologies 41
4 Les Bases de Données à Base Ontologique (BDBO) 42
4.1 Définition du concept de BDBO 42
4.2 Représentation des ontologies 43
4.3 Représentation des données à base ontologique (instances) 45
4.4 Synthèse sur les bases de données à base ontologique 48
5 Conclusion : application du concept d'ontologie aux bases de données 48

Chapitre 2 Exigences pour un langage d'exploitation de BDBO 51

1 Introduction 53
2 Exigences liées à l'architecture de BDBO proposée 53
2.1 Exigences liées au modèle en oignon 54
2.2 Exigences liées à la compatibilité avec l'architecture traditionnelle des bases de données 56
2.3 Exigences sur le pouvoir d'expression du langage 57
2.4 Exigences sur l'implantation du langage 59
3 Analyse des langages de BDBO conçus pour le Web Sémantique 60
3.1 Les langages conçus pour le formalisme RDF 60
3.2 Les langages conçus pour le modèle d'ontologies RDF-Schema 66
4 Analyse des autres langages de BDBO 77
4.1 Le langage CQL associé au modèle d'ontologies PLIB 77
4.2 Le langage SOQA-QL indépendant d'un modèle d'ontologies particulier . . 82
5 Conclusion 86

Partie II Notre proposition : le langage OntoQL

Chapitre 3 Traitements des données à base ontologique d'une BDBO **91**

1 Introduction 93

2 Exploitation des données à base ontologique au niveau logique 94

2.1 Modèle de données du niveau logique 94

2.2 Langage de définition, de manipulation et d'interrogation de données 96

2.3 Utilisation 99

3 Exploitation des données à base ontologique au niveau ontologique. Application à la couche OCC 100

3.1 Modèle de données du niveau ontologique, couche OCC 100

3.2 Aspects syntaxiques 104

3.3 Langage de Définition de Données (LDD) 106

3.4 Langage de Manipulation de Données (LMD) 111

3.5 Langage d'Interrogation de Données (LID) 112

4 Exploitation des données à base ontologique au niveau ontologique. Application à la couche OCNC 119

4.1 Problématique 119

4.2 Langage de Définition de Vues (LDV) 120

4.3 Utilisation 123

5 Exploitation des données à base ontologique au niveau ontologique. Application à la couche OL 127

5.1 Modèle de données du niveau ontologique, couche OL 127

5.2 Aspects syntaxiques 128

5.3 Langage de définition, de manipulation et d'interrogation de données 129

6 Conclusion 131

Chapitre 4 Traitements des ontologies et simultanément des ontologies et des données d'une BDBO **133**

1 Introduction 135

2 Exploitation des ontologies d'une BDBO 135

2.1 Modèle de données permettant de représenter les ontologies 136

2.2 Aspects syntaxiques 139

2.3 Langage de Définition des Ontologies (LDO) 140

2.4 Langage de Manipulation des Ontologies (LMO) 142

2.5 Langage d'Interrogation des Ontologies (LIO) 145

3 Interrogation conjointe des ontologies et des données 149

3.1 Des ontologies vers les données à base ontologique 149

3.2 Des données à base ontologique vers les ontologies 152

4 Analyse critique du langage OntoQL et perspectives d'évolution 153

4.1 Analyse du langage OntoQL par rapport aux exigences définies 153

4.2 Perspectives d'évolution du langage OntoQL 157

5 Conclusion 159

Partie III Validation théorique et opérationnelle du langage OntoQL

Chapitre 5 Sémantique formelle du langage OntoQL **163**

1 Introduction 165

2 Définition formelle du modèle de données d'une BDBO 165

2.1 Modèle de données *Encore* d'une BDOO 165

2.2 Modèle de données d'accès aux données d'une BDBO 166

2.3 Modèle de données d'accès aux ontologies d'une BDBO 169

2.4 Modèle de données d'accès aux ontologies et aux données d'une BDBO . . 170
3 *OntoAlgebra* : adaptation de l'algèbre *Encore* au modèle de données d'une BDBO . 170
3.1 Algèbre *Encore* permettant d'interroger les données d'une BDOO 171
3.2 Algèbre permettant d'interroger les données d'une BDBO 172
3.3 Algèbre permettant d'interroger les ontologies d'une BDBO 179
3.4 Algèbre permettant d'interroger les ontologies et les données d'une BDBO . 181
4 Etude du langage OntoQL en utilisant l'algèbre *OntoAlgebra* 183
4.1 Propriétés du langage OntoQL 183
4.2 Optimisation logique de requêtes OntoQL 184
5 Conclusion 188

Chapitre 6 Implantation du langage OntoQL sur le prototype OntoDB 189

1 Introduction 191
2 Le prototype de BDBO OntoDB 191
2.1 L'architecture OntoDB 191
2.2 Représentation des données 192
2.3 Représentation des ontologies 194
3 Implantation du modèle de données de OntoQL sur OntoDB 196
3.1 Représentation des données 196
3.2 Représentation des ontologies 198
3.3 Représentation du modèle d'ontologies 200
4 Implantation d'un interpréteur de requêtes OntoQL sur OntoDB 202
4.1 Traitement d'une requête OntoQL 202
4.2 Traduction des requêtes OntoQL sur les données 204
4.3 Traduction des requêtes OntoQL sur les ontologies 209
4.4 Traduction des requêtes OntoQL sur les ontologies et sur les données 211
5 Interfaces usuelles d'un langage d'interrogation de bases de données 214
5.1 Interface interactive OntoQLPlus 214
5.2 Interface graphique OntoQBE 215

5.3 Interface JAVA JOBDBC 216

5.4 Interface JAVA OntoAPI 218

6 Interfaces spécifiquement conçues pour les BDBO 221

6.1 Interface par mots clés 221

6.2 Interface avec le langage SPARQL 223

7 Conclusion 230

Conclusion et perspectives **233**

Bibliographie **241**

Annexes **251**

Annexe A Syntaxe complète du langage OntoQL **251**

1 notations 251

2 Les tokens 251

2.1 Les mot-clés 251

2.2 Les éléments lexicaux 253

2.3 Les identifiants interprétés 254

3 Les ressources 255

3.1 Les types de données 255

3.2 Les valeurs 255

4 Langage de définition de données : LDD et LDO 259

5 Langage de manipulation de données : LMD et LMO 261

6 Langage d'interrogation de données : LID et LIO 262

7 Langage de définition de vues : LDV 263

8 Paramétrage du langage 263

Annexe B Comparaison du pouvoir d'expression de OntoQL avec des langages conçus pour RDF/RDF-Schema **265**

1 L'exemple utilisé 265

2 Cas d'utilisation 266

3 Synthèse 269

Annexe C Détails sur la sémantique du langage OntoQL **271**

1 Les opérateurs de *OntoAlgebra* permettant d'interroger les données d'une BDBO . . 271

2 Expression algébrique d'une requête OntoQL 275

3 Règles d'équivalence sur l'algèbre *OntoAlgebra* 276

Annexe D Détails sur l'implantation du langage OntoQL **279**

1 Traduction d'une expression algébrique d'une requête OntoQL en une expression de l'algèbre relationnelle étendue 279

2 Correspondances entre le modèle d'ontologies noyau de OntoQL et le modèle PLIB 281

3 Correspondances entre le modèle RDF-Schema et le modèle d'ontologies noyau de OntoQL 281

Table des figures **283**

Liste des tableaux **285**

Glossaire **287**

Introduction

Contexte

Avec le développement d'Internet et des Intranets, l'environnement dans lequel se trouvent les bases de données est de plus en plus ouvert, dynamique, hétérogène et distribué. Dans ce contexte, l'interrogation, l'échange et l'intégration de données contenues dans les bases de données sont devenus des problèmes cruciaux. Au coeur de ces problèmes se situe la nécessité d'expliciter la sémantique dans les bases de données.

Dans les années 90, les travaux menés dans le domaine de la Représentation de Connaissances ont mis en évidence la faisabilité de modéliser explicitement et de façon consensuelle les caractéristiques structurelles et descriptives du domaine sur lequel porte une connaissance pour rendre celle-ci plus facilement partageable. Ces travaux ont abouti à la définition de modèles permettant d'expliciter la sémantique des données appelées *ontologies* [Gruber, 1993]. Des travaux utilisant des ontologies ont été menés par différentes communautés pour proposer des solutions à des problèmes très variés. Ces travaux ont vu naître :

- différentes catégories d'ontologies. Les deux catégories d'ontologies principalement utilisées sont les *ontologies conceptuelles* qui visent à représenter les catégories de concepts et les propriétés de ces concepts présents dans un domaine d'étude et les *ontologies linguistiques* qui visent à définir le sens des mots et les relations entre ces mots ;
- différents formalismes ou modèles d'ontologies permettant de définir des ontologies. Il existe aujourd'hui des modèles d'ontologies stables dans différents domaines tel que OWL dans le domaine du Web Sémantique ou PLIB dans le domaine technique.

Les ontologies sont alors apparues comme une solution possible à l'explicitation de la sémantique dans les bases de données. Pour exploiter la notion d'ontologie au sein de bases de données, deux problèmes restaient à résoudre :

- où et comment représenter les dites ontologies ;
- quels outils et quelles méthodes définir pour exploiter de façon opérationnelle de telles ontologies.

Représenter les ontologies : les BDBO

Pour résoudre le premier problème, la solution a consisté à embarquer les ontologies au sein des bases de données. Cette démarche permet alors d'augmenter les bases de données par les ontologies qui décrivent la sémantique des concepts qu'elles représentent. Cet enrichissement des modèles classiques de bases de données a donné naissance aux *Bases de Données à Base Ontologique* que nous notons BDBO. La sémantique des données contenues dans ces bases de données est fournie par les ontologies qu'elles conservent. De telles données, associées à une ontologie qui en définit le sens, sont dites *données à base ontologique*.

Un premier domaine d'application a été celui du Web. En effet, depuis les années 2000, dans le contexte du Web Sémantique, les ontologies ont été utilisées pour annoter les données du Web afin de faciliter leur traitement par les machines [Berners-Lee et al., 2001]. La quantité des données du Web décrites par des ontologies devenant de plus en plus importante, plusieurs propositions, pour stocker ces données à base ontologique et les ontologies qui les décrivent dans des bases de données à base ontologique, ont vu le jour [Alexaki et al., 2001, Broekstra et al., 2002]. Ces BDBO, ainsi que les langages de requêtes qui leur ont été associés, permettent de manipuler les ontologies et les données à base ontologique dans un environnement uniforme. Cependant, ces BDBO n'ont pas été conçues pour expliciter la sémantique de données contenues dans une base de données au sens usuel, mais plus, pour fournir une solution de persistance à des données issues du Web et décrites par des ontologies conceptuelles représentées selon un modèle d'ontologies particulier. En conséquence, la plupart des architectures de BDBO proposées s'éloignent de l'architecture traditionnelle des bases de données. Ces BDBO sont en général basées sur des schémas de représentation des données très éclatés, de type binaire ou ternaire. Elles sont influencées par la structuration des données sur le Web (structure RDF tout particulièrement). Ce type de base de données à base ontologique n'exploite pas les informations de structure et de typage dont on dispose traditionnellement dans les modèles logiques des bases de données.

Exploiter les ontologies : les langages pour les BDBO

S'agissant des langages, le même constat peut être effectué [Prud'hommeaux and Seaborne, 2006, Karvounarakis et al., 2002]. En effet, ces langages ne permettent ni d'exploiter, ni de gérer la structure des données et ils ne préservent aucune compatibilité avec les langages traditionnels d'exploitation de bases de données. En particulier, ils ne permettent pas d'accéder aux données à partir du modèle logique, ni de modifier la structure de ce modèle logique.

Récemment, plusieurs architectures de BDBO ont été proposées [Dehainsala et al., 2007a, Pierra et al., 2005, Park et al., 2007] où l'implantation des données à base ontologique se rapproche de la structure des bases de données traditionnelles. Sous certaines hypothèses de typage et de structuration, ces BDBO permettent, d'une part, d'obtenir de meilleures performances dans le traitement des requêtes et, d'autre part, elles permettent d'indexer des bases de données existantes par des ontologies.

Néanmoins, aucun langage de BDBO n'a été défini spécifiquement pour ce type d'architecture. C'est l'objectif principal des travaux présentés dans cette thèse.

Plus précisément, ce travail de thèse vise à concevoir un langage d'exploitation de BDBO qui :

– s'appuie sur une architecture de BDBO compatible avec les modèles de base de données traditionnels, c'est-à-dire résultant de l'architecture ANSI/SPARC, pour permettre d'expliciter la sémantique des données qu'elle contient ;
– n'est pas spécifique d'un modèle d'ontologies particulier, permettant ainsi de prendre en compte plusieurs modèles d'ontologies ;
– permet d'exploiter les caractéristiques des différentes catégories d'ontologies ;
– reste compatible avec les langages de SQL, qui font office de standard dans les bases de données, pour continuer à autoriser une manipulation des données à partir du modèle logique lorsque, par exemple, l'ontologie est inconnue.

Ces capacités constituent les principaux objectifs assignés à ce langage.

Notre proposition

Le langage *OntoQL*, que nous proposons, répond effectivement aux objectifs précédents. Il est fondé sur une architecture de BDBO qui étend l'architecture ANSI/SPARC. Ce langage permet de définir des ontologies selon un modèle en couches qui caractérise différentes catégories d'ontologies. L'approche proposée permet alors d'utiliser conjointement des ontologies issues des différentes catégories identifiées. Au coeur de ce modèle en couches se trouve un modèle d'ontologies noyau qui représente la sémantique commune des différents modèles d'ontologies. Ce noyau peut être étendu pour prendre en compte les spécificités de chaque modèle. Enfin, ce langage est compatible avec SQL et propose une syntaxe qui en est proche pour permettre des traitements sur les données aussi bien au niveau ontologique qu'au niveau logique. Il combine ainsi les capacités des langages traditionnels de bases de données qui disposent d'opérateurs algébriques pour manipuler les données à partir de leur structure et les capacités des langages proposés dans le contexte du Web Sémantique pour interroger des données à partir de leur sémantique, c'est-à-dire de l'ontologie qui en définit le sens. Il permet, en plus, d'interroger à la fois les ontologies et les données d'une BDBO offrant ainsi des capacités tout à fait nouvelles au sein des langages de type SQL.

Structure du mémoire

Ce mémoire est organisé en trois parties de deux chapitres.

La première partie décrit les motivations pour la définition d'un nouveau langage d'exploitation des ontologies et des données à base ontologique.

Le chapitre 1 présente une analyse du concept d'ontologie dans une perspective d'utilisation pour les bases de données. Après avoir précisé notre interprétation du concept d'ontologie, nous présentons un modèle en couches, nommé *modèle en oignon*, qui permet de combiner les différentes catégories d'ontologies. Nous présentons ensuite différents modèles d'ontologies existants et montrons qu'ils possèdent un noyau commun. Enfin, nous présentons les architectures de BDBO de la littérature et les comparons à l'architecture traditionnelle des bases de données. Les résultats obtenus au cours de cette analyse sont utilisés dans la conclusion de ce chapitre pour proposer l'extension de l'architecture ANSI/SPARC avec

un niveau supplémentaire destiné à permettre la description, ou l'annotation sémantique, des données contenues dans une base de données par des références à des concepts conservés dans des ontologies.

Le chapitre 2 propose d'abord une analyse en termes d'exigences de l'architecture de BDBO proposée au chapitre 1. Ces exigences sont établies en suivant les principes fondamentaux introduits avec l'architecture ANSI/SPARC. En nous basant sur les exigences définies, nous montrons ensuite les avantages et insuffisances des langages d'exploitation de BDBO proposés dans la littérature.

A partir des exigences définies précédemment, la deuxième partie expose le langage OntoQL.

Le chapitre 3 présente les traitements proposés par le langage OntoQL pour exploiter les données à base ontologique. Nous montrons que ce langage reste compatible avec le langage SQL, permettant ainsi d'exploiter les données d'une BDBO au niveau logique. Puis, nous expliquons comment nous avons conçu ce langage pour qu'il reste proche du langage SQL tout en permettant d'exploiter les données au niveau ontologique, indépendamment du niveau logique, selon les trois couches du modèle en oignon.

Le chapitre 4 présente les traitements proposés par le langage OntoQL pour exploiter les ontologies d'une BDBO. Nous montrons que ces traitements sont basés sur un modèle d'ontologies noyau afin de ne pas être spécifiques d'un modèle d'ontologies donné. Ce modèle d'ontologies noyau peut-être étendu par des instructions du langage. Puis, nous présentons les traitements permettant d'interroger à la fois les ontologies et les données d'une BDBO. Le langage OntoQL est en effet équipé de mécanismes, issus des langages conçus pour les bases de données fédérées, qui permettent de naviguer des éléments des ontologies vers leurs instances ou dans le sens inverse. Ce chapitre se termine par une analyse critique du langage OntoQL par rapport aux exigences établies dans le chapitre 2.

La troisième partie présente la validation formelle et opérationnelle du langage OntoQL.

En vue de préciser les fondements de OntoQL, le chapitre 5 présente la sémantique formelle sur laquelle le langage OntoQL est fondé. Cette sémantique est constituée d'une algèbre d'opérateurs nommée *OntoAlgebra*. Nous montrons comment nous avons construit cette algèbre à partir de celle de *Encore* proposée pour les bases de données orientées-objets. Puis, en utilisant l'algèbre *OntoAlgebra*, nous discutons les propriétés de fermeture et de complétude relationnelle du langage OntoQL et étudions l'optimisation de requêtes écrites dans ce langage.

Le chapitre 6 est consacré aux développements menés autour du langage OntoQL. Nous présentons d'abord son implantation sur la BDBO OntoDB en montrant qu'elle a été conçue pour pouvoir être portée sur d'autres BDBO et pour permettre l'optimisation des requêtes OntoQL à différents niveaux. Puis, nous montrons les différents outils que nous avons développés pour compléter la suite logicielle construite autour de la BDBO OntoDB.

Nous terminons cette thèse par une conclusion générale et en exposant les perspectives ouvertes par les travaux réalisés.

Cette thèse comporte quatre annexes.

L'annexe A donne la syntaxe complète du langage OntoQL. L'annexe B présente une comparaison du pouvoir d'expression du langage OntoQL avec ceux conçus pour RDF/RDF-Schema en se basant sur une étude proposée dans la littérature. Enfin, les annexes C et D présentent respectivement des détails sur la sémantique et sur l'implantation du langage OntoQL.

Première partie

De la nécessité d'un nouveau langage d'exploitation des bases de données à base ontologique

Chapitre 1

Ontologies et bases de données à base ontologique

Sommaire

1 **Introduction** **9**
2 **Les ontologies de domaine dans une perspective d'exploitation de bases de données** **10**
2.1 Spécificité d'une ontologie de domaine comme modèle d'un domaine . . 10
2.2 Utilisation des ontologies dans le domaine des bases de données 14
2.3 Utilisation des ontologies dans d'autres domaines 15
2.4 Une taxonomie des ontologies de domaine 17
2.5 Relation entre les différentes catégories d'ontologies : le modèle en oignon 21
2.6 Liens entre le modèle en oignon et les bases de données 24
2.7 Synthèse de notre interprétation des ontologies de domaine 25
3 **Les modèles d'ontologies** **25**
3.1 Modèles d'ontologies orientés vers la définition de OCC 25
3.2 Modèles d'ontologies orientés vers la définition de OCNC 32
3.3 Traitements pour les OL 37
3.4 Noyau commun aux modèles d'ontologies 38
3.5 Synthèse sur les modèles d'ontologies 41
4 **Les Bases de Données à Base Ontologique (BDBO)** **42**
4.1 Définition du concept de BDBO 42
4.2 Représentation des ontologies 43
4.3 Représentation des données à base ontologique (instances) 45
4.4 Synthèse sur les bases de données à base ontologique 48
5 **Conclusion : application du concept d'ontologie aux bases de données** **48**

Résumé. L'objectif de ce chapitre est de présenter les bases du langage d'exploitation de bases de données à base ontologique proposé dans ce mémoire de thèse [Jean et al., 2007d]. Ces bases s'appuient, d'une part, sur l'analyse des multiples notions d'ontologie que l'on rencontre actuellement dans la littérature et, d'autre part, sur les conséquences que l'on peut en tirer dans une perspective d'exploitation de bases de données.

1 Introduction

La conception d'une base de données est réalisée en plusieurs étapes. En premier, un modèle conceptuel est conçu à partir des besoins utilisateurs. Ce modèle prescrit les informations qui doivent être représentées dans la base de données cible. Ensuite, ce modèle conceptuel est transformé en un modèle logique qui fournit les structures de représentation concrètes pour les concepts du modèle conceptuel en fonction du type de base de données considéré. Ce processus de conception pose les deux problèmes suivants :

- l'échange de données entre différentes bases de données issues de modèles conceptuels différents, bien que portant sur le même domaine d'application, est difficile car ces modèles présentent à la fois des différences structurelles et sémantiques ;
- une part importante de la sémantique des données est perdue pendant la transformation du modèle conceptuel en un modèle logique.

Ces deux problèmes ont été traités indépendamment l'un de l'autre par la communauté des bases de données. Pour le premier problème, des approches permettant l'intégration de bases de données ont été proposées [Chawathe et al., 1994, Arens et al., 1993, Levy et al., 1996]. Pour le second problème, de nouveaux modèles de bases de données ont été conçus dans le but de réduire le fossé entre les modèles conceptuels et logiques [Bancilhon et al., 1992, Stonebraker and Moore, 1996].

Au cours de la même période, les travaux menés dans le domaine de la Représentation de Connaissances ont mis en évidence la faisabilité de modéliser explicitement les aspects structurels et descriptifs des concepts d'un domaine à travers des modèles consensuels appelés ontologies. Permettant ainsi une représentation consensuelle et explicite de la sémantique des données, les ontologies, appliquées aux bases de données, apparaissent comme une solution possible aux deux problèmes évoqués précédemment.

De nos jours, la notion d'ontologie a été utilisée dans de nombreux domaines de recherche incluant le traitement du langage naturel, la fouille de données, le commerce électronique, le Web Sémantique, la spécification de composants logiciels et matériels, l'ingénierie et l'intégration de systèmes d'information. Cette diversité de secteurs d'utilisation a vu naître :

- plusieurs définitions de la notion d'ontologie ;
- de nombreux formalismes de représentation d'ontologies souvent appelés *modèles d'ontologies* ;
- plusieurs catégories d'ontologies se focalisant soit sur la définition des mots utilisés dans un domaine d'étude, soit sur les concepts présents dans ce domaine ;
- différentes architectures de bases de données permettant de stocker ontologies et données dans un même environnement.

Dans ce chapitre, nous présentons une analyse de ces différentes propositions [Jean et al., 2007d]. L'objectif est (1) de préciser notre interprétation de la notion d'ontologie, (2) d'identifier la catégorie d'ontologies et le modèle d'ontologies considérés dans nos travaux et (3) de tirer les conséquences de cette analyse dans une perspective d'exploitation de bases de données.

Ce chapitre est composé de quatre sections. Dans la section suivante, nous présentons d'abord les multiples notions d'ontologies que l'on rencontre dans la littérature et proposons trois caractéristiques

qui distinguent une ontologie des autres modèles informatiques. Nous présentons ensuite un modèle en couches qui permet de combiner les différentes catégories d'ontologies. Dans la section 3, nous analysons les modèles d'ontologies proposés et montrons leur complémentarité pour la conception d'une ontologie. Dans la section 4, nous analysons les bases de données permettant de stocker à la fois des ontologies et des données en comparant leur architecture à celle rencontrée dans les bases de données traditionnelles. Enfin, nous concluons ce chapitre dans la section 5 en présentant notre proposition d'application des ontologies aux bases de données.

2 Les ontologies de domaine dans une perspective d'exploitation de bases de données

La notion d'ontologie trouve son origine dans une branche de la philosophie traitant de la science de l'être. Cette discipline philosophique, initiée par Aristote, essaie de définir l'être à travers ce qui le caractérise de façon essentielle. Le terme lui-même apparaît tardivement en 1692, emprunté au latin scientifique *ontologia*. Ce terme a été introduit en informatique dans les années 70 par McCarthy dans le domaine de l'Intelligence Artificielle [Psyché et al., 2003]. Il a ensuite été repris dans le domaine de la Représentation des Connaissances dans les années 1990. De nos jours, le nombre de domaines dans lesquels le concept d'ontologie est utilisé a encore augmenté, incluant le traitement du langage naturel, la fouille d'information, le Web Sémantique, l'intégration d'information, etc. Cette diversification de domaines d'utilisation a vu naître de nombreuses interprétations de ce terme. Notre but dans cette section est de présenter notre interprétation du concept d'ontologie et de préciser ce qui, pour nous, distingue une ontologie des autres notions de modèles mis en oeuvre en informatique.

2.1 Spécificité d'une ontologie de domaine comme modèle d'un domaine

Nous présentons d'abord quelques définitions usuelles de la notion d'ontologie, puis, nous décrivons les trois caractéristiques que nous avons définies à partir de ces définitions, pour distinguer une ontologie d'un modèle en informatique. Ces caractéristiques engendrent la proposition d'une nouvelle définition de la notion d'ontologie.

2.1.1 Définitions usuelles

De nombreuses définitions pour la notion d'ontologie ont été proposées [Gruber, 1993, Borst, 1997, Guarino, 1998]. D'après le Free Online Dictionary Of Computing[1] (FOLDOC) une ontologie est « an explicit formal specification of how to represent the objects, concepts and other entities that are assumed to exist in some area of interest and the relationships that hold among them ». Le Dico du Net[2] définit une ontologie comme une « organisation hiérarchique de la connaissance sur un ensemble d'objets par leur regroupement en sous catégories suivant leurs caractéristiques essentielles ». Dans la littérature scientifique, la définition admise et la plus couramment citée est celle de Gruber : « an explicit specification

[1] http://foldoc.org

[2] http://www.dicodunet.com

of a conceptualization » [Gruber, 1993]. Borst en a proposé la modification en 1997 : « an ontology is a formal specification of a shared conceptualization » [Borst, 1997].

Ces définitions présentent une ontologie comme une conceptualisation d'une partie du monde. Cependant, pour nous, ces définitions recouvrent la plupart des types de modèles déjà utilisés en informatique tels que les modèles conceptuels, les modèles de connaissances ou les formats d'échange. Pour comprendre la distinction qui existe entre une ontologie et les autres modèles informatiques, il est nécessaire d'étudier plus finement les descriptions données autour de ces définitions.

Partant de ce constat, nous proposons trois caractéristiques permettant de distinguer une ontologie d'un autre modèle informatique. Ces caractéristiques identifient pour nous ce qu'est une ontologie. Mais, avant de présenter ces caractéristiques, nous précisons d'abord les termes utilisés dans ce mémoire.

2.1.2 Ontologie, classe, propriété, instance, entité et attribut

Dans le contexte de l'informatique, deux types d'ontologies sont fréquemment distingués. Les *ontologies de haut niveau* fournissent des définitions pour des concepts généraux tels que les notions de processus, d'objet ou d'événement afin de servir de fondement pour des ontologies plus spécifiques dites de domaine [Niles and Pease, 2001, Gangemi et al., 2003]. Ces *ontologies de domaine* sont liées à un univers du discours particulier. Elles décrivent et représentent la connaissance existant dans le domaine correspondant à cet univers. Dans ce travail de thèse, nous nous sommes focalisés sur les ontologies de domaine qui permettent de décrire la sémantique des objets d'un domaine d'étude. Dans ce mémoire, le mot ontologie sera utilisé pour désigner une ontologie de domaine, ce qui n'exclut pas que cette ontologie de domaine soit elle-même fondée sur une ontologie de haut niveau.

Les ontologies de domaine représentent la sémantique des concepts d'un domaine en termes de *classes* et de *propriétés*. Une classe, aussi appelée *concept*, regroupe et abstrait les objets du domaine présentant des caractéristiques communes. Une propriété, aussi appelée *attribut* ou *rôle*, permet de caractériser les objets du domaine par une ou plusieurs valeurs. Elle peut être définie sur un *domaine* indiquant la classe regroupant les objets qu'elle permet de décrire et associée à un *codomaine* indiquant le type de données dans lequel elle peut prendre ses valeurs. Les classes ont une *extension* constituée d'un ensemble d'*instances*. Une instance, aussi appelée *individu*, désigne un objet du domaine. Elle est décrite par son appartenance à une ou plusieurs classes et par un ensemble de valeurs de propriétés.

Pour définir des ontologies, un *modèle d'ontologies* est nécessaire. Un modèle d'ontologies est un formalisme permettant de représenter des ontologies. Généralement représenté sous la forme d'un modèle orienté-objet, il est composé d'un ensemble d'*entités* et d'*attributs* permettant de décrire les éléments d'une ontologie tels que ses classes et ses propriétés.

Convention de rédaction.

Pour les besoins de cette thèse, nous avons choisi de distinguer les termes associés aux ontologies et aux modèles d'ontologies. Ainsi, nous réservons les mots :
– *classes*, *propriétés* et *instances* pour caractériser des éléments d'une ontologie ;
– *entités* et *attributs* pour caractériser des éléments d'un modèle d'ontologies.

Ayant précisé les termes utilisés dans ce mémoire, nous présentons dans la section suivante les trois caractéristiques qui, pour nous, distinguent une ontologie d'un autre modèle en informatique.

2.1.3 Caractéristiques d'une ontologie de domaine

De notre point de vue, une ontologie est une conceptualisation qui doit présenter les trois caractéristiques suivantes.

1. *Formelle*. Une ontologie est une conceptualisation basée sur une sémantique formelle (par exemple dénotationnelle) qui permet d'en vérifier la consistance et/ou de réaliser des raisonnements et déductions à partir de ses concepts et de ses instances. Nous notons que cette caractéristique exclut les approches qui proposent des méta-modèles sans leur associer une sémantique explicite.
2. *Consensuelle*. Une ontologie est une conceptualisation acceptée par une communauté qui peut être plus ou moins large. Elle n'est ainsi pas conçue spécifiquement pour un système particulier. Au contraire, elle décrit les concepts d'un domaine d'étude de manière à permettre de satisfaire les besoins techniques et métiers partagés par l'ensemble des membres de la communauté. Ainsi, deux systèmes développés au sein de la même communauté et portant sur le même domaine d'application peuvent être basés sur la même ontologie. L'ontologie fournit alors une interface d'accès commune à ces deux systèmes, indépendante de leurs implantations particulières, qui permet de réaliser certains traitements automatiques comme l'échange d'information entre ces deux systèmes ou l'intégration de ces deux systèmes. Précisons que la communauté dans laquelle une ontologie est consensuelle doit être plus large que celle impliquée dans le développement d'un système particulier. Par exemple, l'ontologie Gene Ontology (GO)[3] est un effort de collaboration entre plus de 10 institutions permettant de décrire des produits de gènes (protéines ou ARN) dans un organisme donné. De même, les ontologies de produit conformes à la norme ISO 13584 (PLIB) font l'objet d'un consensus international résultant d'un processus rigoureux de standardisation et sont publiées comme des standards internationaux ISO ou IEC. Nous notons que cette caractéristique exclut la plupart des modèles conceptuels qui ne sont généralement définis que pour un système particulier.
3. *Référençable.* Chaque concept d'une ontologie est associé à un identifiant permettant de le référencer à partir de n'importe quel environnement, indépendamment de l'ontologie dans laquelle il a été défini (par exemple, avec une URI). Nous notons que cette caractéristique est très originale et exclut, en particulier, tous les formats d'échange de données tels que la norme STEP (Standard for the Exchange of Product Model Data) [ISO10303, 1994], où les éléments représentés ne peuvent être référencés qu'à partir de la structure d'échange spécifiée.

Nous montrons dans la section suivante que ces caractéristiques permettent de distinguer une ontologie des autres modèles informatiques et en particulier des modèles conceptuels des bases de données.

2.1.4 Différences entre les modèles informatiques usuels et une ontologie

Nous étudions ces différences dans les cas des modèles conceptuels des bases de données et des modèles de connaissances introduits dans le domaine de la Représentation de Connaissances ou de l'Intelligence Artificielle.

Un modèle conceptuel d'une base de données, tout comme une ontologie, définit une conceptualisation d'une partie du monde. Aussi, un modèle conceptuel semble similaire à une ontologie. Pour les

[3] `http://www.geneontology.org/`

distinguer nous identifions, pour le cas des bases de données qui constituent notre domaine d'intérêt, les différences entre un modèle conceptuel et une ontologie par rapport aux trois caractéristiques définies.

– Formelle. Le caractère formel d'un modèle conceptuel dépend du formalisme de modélisation utilisé. Si certains sont particulièrement ambigus, d'autres, comme par exemple EXPRESS [Schenk and Wilson, 1994], disposent d'une sémantique bien définie. Dans le meilleur des cas, les modèles conceptuels peuvent donc être considérés comme formels.
– Consensuelle. Un modèle conceptuel est conçu en fonction des besoins d'une application : il prescrit et impose les informations qui seront représentées dans une application particulière. Deux systèmes répondant, en général, à des besoins au moins légèrement différents, les modèles conceptuels sont toujours différents d'application en application. Ainsi, les modèles conceptuels ne respectent pas cette caractéristique.
– Référençable. Un identifiant d'un élément défini dans un modèle conceptuel est un nom qui ne peut être référencé sans ambiguïté que dans le contexte du système d'information basé sur ce modèle conceptuel particulier. Ainsi, un modèle conceptuel ne satisfait pas cette caractéristique.

Cette analyse montre qu'un modèle conceptuel d'une base de données n'est pas une ontologie.

Un modèle de connaissances n'est pas une ontologie pour des raisons similaires à celles évoquées pour les modèles conceptuels des bases de données. Ces modèles, utilisant des constructeurs issus de la logique sont, en général, formels. En effet, les modèles de connaissances se focalisent sur les capacités d'inférence permises. Par contre, avant que la notion d'ontologie n'ait émergée, aucun mécanisme permettant de référencer chaque concept d'un modèle de connaissances n'était disponible. De même, aucun processus n'est, en général, défini pour permettre d'assurer un consensus sur les concepts définis au delà de la communauté des utilisateurs de ce système. C'est pourquoi, comme les modèles conceptuels, les modèles de connaissances ne présentent pas les caractéristiques de consensualité et de référençabilité.

Les caractéristiques définies permettent ainsi de distinguer une ontologie d'un autre modèle informatique. Pour mettre en avant ces distinctions, nous avons proposé une nouvelle définition d'une ontologie de domaine fondée sur ces trois caractéristiques.

2.1.5 Ontologie de domaine : définition

Dans nos travaux, une ontologie de domaine est un *dictionnaire formel et consensuel des catégories et propriétés d'entités existant dans un domaine d'étude et des relations qui les lient*. Par entité, nous signifions quoi que ce soit qui puisse être pensé sur le domaine d'étude. Le terme dictionnaire fait référence au fait que toute entité ou relation décrite dans l'ontologie peut être directement référencée par un symbole, dans n'importe quel but et à partir de n'importe quel contexte, indépendamment des autres entités et relations. Ce symbole d'identification peut être soit un identifiant indépendant d'une langue naturelle, soit un *terme*, c'est-à-dire un ensemble de mots dans une langue naturelle donnée. Mais, quel que soit le symbole utilisé, et à la différence des dictionnaires linguistiques, ce symbole référence directement un concept d'un domaine d'étude dont la description est donnée formellement permettant ainsi des raisonnements (automatiques ou non) et de la vérification de consistance.

Dans cette première partie visant à préciser notre analyse des ontologies de domaine dans une perspective d'exploitation de bases de données, nous avons précisé la distinction que nous faisons entre une

ontologie et d'autres modèles informatiques. Les ontologies possèdent des caractéristiques qui les distinguent de ces autres modèles. Dans les prochaines sections, nous montrons l'intérêt de ces particularités en présentant différentes utilisations des ontologies.

2.2 Utilisation des ontologies dans le domaine des bases de données

Les spécificités d'une ontologie comme modèle d'un domaine permettent de résoudre différents problèmes rencontrés dans le contexte des bases de données.

2.2.1 Conception/Indexation de bases de données

Puisqu'une ontologie est une conceptualisation sur un domaine d'étude, elle peut être utilisée comme base pour la conception d'une base de données. Cette approche est suivie dans [del Mar Roldán García et al., 2005], [Sugumaran and Storey, 2006] et [Dehainsala et al., 2007b]. De manière générale, la démarche proposée dans ces approches est la suivante :

1. choisir une ontologie couvrant le domaine d'étude sur lequel porte l'application pour laquelle la base de données est conçue ;
2. éventuellement, étendre cette ontologie pour ajouter les concepts nécessaires qui ne sont pas représentés ;
3. choisir le sous-ensemble de cette ontologie qui couvre les besoins de l'application pour laquelle la base de données est conçue ;
4. implanter ce sous-ensemble dans la base de données.

Une ontologie peut également être utilisée pour enrichir la sémantique du modèle logique d'une base de données en l'annotant. Cette *indexation sémantique* des bases de données consiste à associer les divers éléments d'un modèle logique (tables, colonnes, contraintes, etc.) à une ontologie. La définition de ces correspondances nécessite un langage de correspondance (*mapping*) tel que R_2O [Barrasa et al., 2004]. Elle peut être assistée par des outils qui proposent des correspondances en s'appuyant sur les noms utilisés [del Mar Roldán García et al., 2005, Sugumaran and Storey, 2006] ou sur la structure du modèle logique traité comme par exemple, les clés étrangères [An et al., 2006].

2.2.2 Échange de données

Une conceptualisation consensuelle d'un domaine, dont chaque élément est référençable, peut être utilisée comme un format d'échange de données sur ce domaine [ISO13584-42, 1998, Chawathe et al., 1994]. Contrairement au format d'échange usuel qui spécifie la structure complète des données échangées et où la signification de chaque élément de données résulte de sa position dans la structure globale, les échanges basés sur des ontologies peuvent être très flexibles. En effet, dans ce type d'échange, la signification de chaque élément d'information peut être définie localement en référençant des identifiants d'éléments d'une ontologie. Cette capacité fait que des structures d'échanges très différentes peuvent être interprétées de façon non ambiguë par un même système receveur.

2.2.3 Intégration de données

Un système d'intégration fournit une interface d'accès unique à des données stockées dans plusieurs sources de données (par exemple, des bases de données). Généralement, ces sources de données sont conçues indépendamment l'une de l'autre par des concepteurs différents. En conséquence, des données relatives à un même sujet peuvent être représentées différemment dans ces différentes sources. C'est le problème de l'hétérogénéité des données. Goh [Goh, 1997] a identifié trois principales causes à l'hétérogénéité sémantique des données.

- *Les conflits de nom* ont lieu lorsque des noms différents sont utilisés pour décrire le même concept (synonyme) ou lorsque le même nom est utilisé pour des concepts différents (homonyme).
- *Les conflits de mesure de valeur* ont lieu lorsque différents systèmes de référence sont utilisés pour évaluer une valeur. C'est le cas, par exemple, lorsque différentes unités de mesure sont utilisées par les différentes sources de données.
- *Les conflits de contexte* ont lieu lorsque des concepts semblent avoir la même signification mais diffèrent en réalité dû à différents contextes de définition ou d'évaluation.

Parce qu'une ontologie peut servir de pivot pour définir la sémantique des données des différentes sources à l'aide de concepts communs, formalisés et référençables, leur utilisation est une solution pour résoudre les problèmes d'hétérogénéité des données. Différentes propositions d'intégration basées sur des ontologies ont ainsi été faites. Un état de l'art de ces approches est présenté dans [Wache et al., 2001] et plus récemment dans [Noy, 2004].

Ces exemples montrent l'intérêt des ontologies dans le contexte des bases de données. Les ontologies permettent de décrire la sémantique des données structurées par un schéma dans une base de données. Dans la section suivante, nous montrons que ce n'est pas le seul domaine où l'utilisation d'ontologies présente un intérêt.

2.3 Utilisation des ontologies dans d'autres domaines

Les particularités d'une ontologie de domaine comme modèle informatique ont été exploitées pour des problèmes aussi variés que le traitement du langage naturel, l'interopérabilité des logiciels ou le Web Sémantique.

2.3.1 Traitement du langage naturel

Le traitement du langage naturel aborde entre autres le problème de la compréhension du langage humain par un ordinateur. L'analyse syntaxique et sémantique du langage naturel est une étape clé pour la résolution de ce problème. Les ontologies peuvent être utilisées dans ces étapes pour, d'une part, construire le lexique utilisé lors de l'analyse syntaxique d'un texte et, d'autre part, pour effectuer des traitements complexes lors de l'analyse sémantique du texte tels que la résolution des problèmes de polysémie Cette approche est par exemple suivie dans [Estival et al., 2004].

Les techniques développées dans le cadre du traitement du langage naturel sont notamment utilisées pour la recherche d'information. En effet, la reconnaissance de similitudes conceptuelles entre des

mots permet d'améliorer la recherche documentaire. Dans les moteurs de recherche actuels, une requête est composée d'un ensemble de mots éventuellement connectés par les opérateurs logiques OU, ET et NON. Le moteur produit essentiellement sa réponse en fonction des mots contenus dans les documents parcourus. De nombreuses propositions ont été faites (cf. [Haav and Lubi, 2001] pour un état de l'art) consistant à utiliser des ontologies dans des moteurs de recherche pour permettre de retourner, en plus, les documents pertinents par rapport à la sémantique des mots de la requête.

2.3.2 Interopérabilité des logiciels

L'utilisation d'un modèle comme une spécification d'un logiciel est à la base de l'approche *MDA* (Model-Driven Archictecture). Dans cette approche, un modèle est utilisé pour générer le code de l'application. Le lien formel entre le modèle et le code permet alors de faire évoluer ce dernier lorsque la spécification du logiciel évolue. Actuellement, plusieurs logiciels abordant des problèmes similaires sur le même domaine sont généralement définis en utilisant différents modèles. En conséquence, un problème d'interopérabilité entre ces logiciels peut apparaître. L'utilisation d'ontologies est une approche possible pour résoudre ce problème. Puisque les ontologies sont consensuelles, les différents modèles d'un même domaine définis pour plusieurs logiciels peuvent être liés à une ontologie de domaine. Ces logiciels peuvent alors interagir en utilisant les accesseurs fournis par cette ontologie. Cette approche est appelée : « ingénierie logicielle dirigée par les ontologies » [Tetlow et al., 2005].

2.3.3 Web Sémantique

La généralisation des accès haut débit au réseau Internet provoque un fort engouement pour ce média de communication. Autrefois réservé aux professionnels de l'informatique avec un contenu essentiellement scientifique, celui-ci est aujourd'hui accessible à tous pour des utilisations très variées. Que ce soit pour faire ses démarches administratives, faire ses courses en ligne, rechercher un emploi ou bien d'autres choses, tout le monde a perçu les bénéfices de ce vecteur de communication.

Cette formidable source d'information souffre pourtant d'un défaut majeur qui décourage bien des débutants. Alors que l'on parle d'une toile pour décrire ce réseau, l'ensemble des services qui y sont offerts sont complètement isolés. En conséquence, pour arriver au résultat escompté, une personne doit, soit avoir une connaissance approfondie du Web, soit passer par une longue période fastidieuse d'errance sur différents sites. L'évolution très rapide et incontrôlée des services conduit souvent à la seconde solution même pour des personnes expérimentées.

Cette difficulté vient du fait que les noms utilisés pour décrire un même service sont très différents et dépendant de son créateur. L'idée du Web Sémantique [Berners-Lee et al., 2001] est de développer des ontologies de domaine puis d'indexer les services par leur description en termes de ces ontologies, rendant ainsi la recherche automatisable et réalisable par des agents logiques.

Nous venons de voir qu'une ontologie est une solution possible à différents problèmes rencontrés dans des domaines aussi variés que celui des bases de données, de l'ingénierie logicielle ou du Web Sémantique. Cette diversité d'objectifs se ressent sur les ontologies conçues. Afin de définir plus précisément les ontologies considérées dans nos travaux, nous avons cherché à définir une taxonomie des ontologies de domaine.

2.4 Une taxonomie des ontologies de domaine

Toutes les ontologies de domaine ne sont pas identiques. Pour nous, le premier critère permettant de classifier les ontologies est l'objet précis que l'ontologie vise à conceptualiser : s'agit-il de conceptualiser l'ensemble des termes qui décrivent le domaine dans un certain langage, ou s'agit-il de conceptualiser les objets du domaine. Ces deux manières de conceptualiser conduisent à la distinction entre les *ontologies linguistiques* et les *ontologies conceptuelles* [Cullot et al., 2003, Pierra, 2003]. En suivant la proposition de Pierra [Pierra, 2003], nous appelons *Ontologies Linguistiques* (OL), les ontologies dont l'objectif est la représentation de la signification des termes utilisés, dans un univers du discours particulier et dans une langue naturelle donnée. Et, nous appelons *Ontologies Conceptuelles* (OC), les ontologies dont le but est la représentation des catégories d'objets et des propriétés des objets présents dans une partie du monde.

Cette distinction permet de faire apparaître deux catégories d'ontologies. Cependant, au sein même de la catégorie des OC, les ontologies présentent des caractéristiques très différentes selon le domaine d'application où elles sont utilisées et selon le modèle d'ontologies avec lequel elles sont décrites. Nous avons donc cherché à raffiner cette catégorie. Ceci nous a amené à mettre l'accent sur un deuxième critère de distinction des ontologies selon que celles-ci définissent un langage canonique ou qu'elles introduisent une redondance. Ceci va en effet jouer un rôle important lorsqu'il s'agira d'exploiter les ontologies dans une perspective d'exploitation de bases de données.

2.4.1 Canonicité ou redondance des OC

Les concepts qui apparaissent dans une OC peuvent être organisés selon les deux catégories suivantes [Gruber, 1995] :

- les *concepts primitifs* sont les concepts « pour lesquels nous ne sommes pas capables de donner une définition axiomatique complète » [Gruber, 1995]. La définition de ces concepts s'appuie sur une documentation textuelle et un savoir partagé avec les utilisateurs. L'ensemble des concepts primitifs suffit pour définir les frontières du domaine conceptualisé par une ontologie. Les concepts primitifs sont la fondation sur laquelle d'autres concepts de l'ontologie pourront ensuite être définis. La définition de concepts primitifs étant toujours, au moins partiellement, informelle, le seul critère de qualité pour une telle définition, est qu'elle représente un consensus parmi une communauté. Sans un tel consensus, personne ne peut certifier la complétude de l'ensemble des concepts consensuels décrits et la consistance (il n'y a pas de contradiction entre les définitions informelles et les relations formelles entre concepts) des définitions fournies par une ontologie ;
- les *concepts définis* sont les concepts « pour lesquels une ontologie fournit une définition axiomatique complète au moyen de conditions nécessaires et suffisantes exprimées en termes d'autres concepts primitifs ou eux-mêmes définis ». Les définitions de tels concepts sont *conservatives* car elles associent un nouveau concept à quelque chose qui est déjà défini par un autre moyen dans l'ontologie en cours de conception [Gruber, 1995]. Elles n'introduisent donc pas de nouveaux concepts mais des alternatives de désignation pour des concepts que l'on pouvait déjà désigner. Elles n'enrichissent donc pas les connaissances sur ce domaine, mais le vocabulaire qui permet de les représenter. C'est le modèle d'ontologies ou le langage associé qui définit les différents

opérateurs permettant de composer les concepts primitifs et/ou définis pour construire d'autres concepts définis. Cette caractéristique est la base des mécanismes d'inférence. Par exemple, un système d'inférence peut effectuer une *classification automatique*, c'est-à-dire déduire automatiquement des relations de subsomptions entre concepts et l'appartenance d'instance à certaines classes, à partir de la définition axiomatique complète des concepts définis.

L'introduction de concepts définis amène donc une notion d'*équivalence conceptuelle* qui permet d'exprimer de plusieurs manières différentes les mêmes concepts. Différents constructeurs ont été proposés pour définir des équivalences conceptuelles. Ces constructeurs peuvent s'appliquer sur les classes d'une ontologie. Par exemple, dans le projet PICSEL [Rousset et al., 2002], des constructeurs des logiques de description sont utilisés pour définir le concept `Hotel` comme une *spécialisation par restriction* du concept primitif `HousingPlace`. Une `HousingPlace` est définie comme un endroit ayant des bâtiments associés, des salles de restauration et des chambres à coucher. Un `Hotel` est alors complètement défini comme une `HousingPlace` qui a plus de 5 chambres à coucher et qui est seulement associé à des bâtiments collectifs.

Les équivalences conceptuelles peuvent également être définies sur les propriétés d'une ontologie. Par exemple, les *fonctions de dérivation* introduites dans le contexte de la logique des frames permettent de dériver les valeurs d'une propriété à partir d'autres propriétés. Ainsi, une propriété `chef`, liant un employé à un autre employé peut être dérivée à partir des propriétés `appartient_à` qui lie un employé à un département et `dirige` qui lie un département à un employé. Cet exemple montre bien que l'instanciation de concepts définis au sein d'une base de données introduit une redondance dans celle-ci.

Grâce à la distinction entre les concepts primitifs et les concepts définis, nous sommes maintenant en mesure de proposer une taxonomie des ontologies de domaine.

2.4.2 Les Ontologies Conceptuelles Canoniques (OCC)

Dans la conception d'une base de données, lorsqu'un modèle conceptuel est défini, la représentation multiple du même fait est à proscrire. La même approche est suivie pour réaliser un modèle d'échange entre deux bases de données différentes. Définir un modèle conceptuel ou un modèle d'échange consiste donc à définir un vocabulaire *canonique* dans lequel chaque information dans le domaine cible est capturée de manière unique, sans définir le moindre constructeur de synonymie. Par exemple, dans le projet STEP, des modèles d'échanges sont définis dans le langage EXPRESS [Schenk and Wilson, 1994] pour différentes catégories d'objets techniques. Ces modèles d'échanges sont toujours canoniques. Ils peuvent alors être utilisés par des utilisateurs industriels pour échanger de façon unique et non ambiguë des descriptions de différents produits du domaine entre différentes organisations.

Nous appelons *Ontologies Conceptuelles Canoniques* (OCC), les ontologies dont les définitions ne contiennent aucune redondance. Dans les OCC, chaque concept du domaine est décrit d'une seule façon, en utilisant une description qui ne peut inclure que des conditions nécessaires. En conséquence, les OCC ne comportent que des concepts primitifs. Un exemple d'une OCC normalisée, définie pour le commerce électronique peut être consulté dans [IEC61360-4, 1999]. Nous verrons ultérieurement que les OCC sont particulièrement importantes dans le contexte de la gestion ou de l'échange de données.

2.4.3 Les Ontologies Conceptuelles Non Canoniques (OCNC)

Dans la conception de bases de données, l'équivalence conceptuelle joue néanmoins un rôle important même s'il est de second ordre. Chaque base de données est conçue pour un domaine particulier et représente de manière canonique n'importe quel fait sur ce domaine. Ensuite, dans le but d'offrir une plus grande souplesse d'accès aux utilisateurs, un concepteur d'une base de données peut définir des vues. Ces concepts définis sont spécifiés en utilisant l'opérateur `CREATE VIEW` sur la base des concepts primitifs qui constituent le schéma de la base de données. Dans les bases de données déductives, cette fonctionnalité existe également en permettant de définir des relations dont les tuples sont dérivés par des règles logiques.

Quels que soient les constructeurs offerts pour exprimer des équivalences conceptuelles, nous appelons *Ontologies Conceptuelles Non Canoniques* (OCNC), les ontologies qui contiennent non seulement des concepts primitifs, mais aussi des concepts définis. Les OCNC sont particulièrement utiles lorsqu'elles sont utilisées comme schéma global de requête. En effet, la redondance qu'elles introduisent permet d'augmenter le nombre de concepts en termes desquels il est possible d'exprimer une requête donnée.

Les constructeurs OCNC sont également très utiles pour définir des mappings entre différentes ontologies. Par exemple, la figure 1.1 présente deux OCC sur le domaine des vins. Ces OCC définissent le concept de vin, sachant qu'un vin est décrit par une couleur. La OCC (A) est *orientée propriétés* ce qui signifie qu'une classe n'est créée dans cette ontologie que lorsqu'elle est nécessaire pour définir le domaine d'une propriété. Ainsi, cette ontologie ne définit qu'une seule classe nommée (`A_WINE`) pour être le domaine de la propriété `color`. A l'opposé, la OCC (B) est *orientée classes*, ce qui signifie qu'elle utilise des classes pour caractériser ses instances plutôt que des propriétés. Ainsi, dans cette ontologie différentes classes sont créées pour les différentes couleurs de vins possibles. Nous avons représenté dans cet exemple la classe des vins rouges nommée `B_REDWINE` ainsi que celle des vins blancs nommée `B_WHITEWINE`. L'appartenance d'une instance à l'une de ces deux classes permet de connaître sa couleur puisque cette information est portée par cette classe.

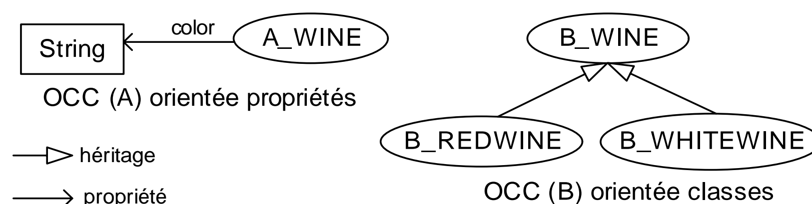


FIG. 1.1 – OCC locales sur le domaine des vins

En appliquant, des opérateurs permettant de créer des OCNC, avec une syntaxe issue des logiques de description, nous pouvons écrire l'équivalence conceptuelle suivante.

$\texttt{B_REDWINE} \equiv \texttt{A_WINE} \sqcap \texttt{color} : \texttt{red}$

Cette définition indique que la classe `B_REDWINE` définie dans la OCC (B) est équivalente à la classe `A_WINE` définie dans la OCC (A) restreinte par la valeur de la propriété `color`. Ces primitives ont une sémantique formelle qui permet d'implanter des mécanismes de raisonnements dans des outils nommés *raisonneurs* comme par exemple, RACER [Haarslev and Möller, 2001]. Ces outils supportent les méca-

nismes de classification automatique des classes et des instances. En conséquence, les deux OCC locales peuvent être automatiquement fusionnées dans la OCNC de la figure 1.2. Cette OCNC fournit un accès global aux données des deux OCC. Et, si on laisse les données dans chacune des bases de données, le raisonneur pourra inférer que toutes les instances de vin rouge sont à la fois les instances de `B_REDWINE` et les instances de `A_WINE` ayant la valeur `red` pour la propriété `color`. Quand ces faits sont matérialisés, comme c'est le cas dans OntoMerge [Dou et al., 2003], il devient possible de diviser l'ontologie fusionnée en la OCC (A) et la OCC (B) et de générer dans chacune l'ensemble des instances figurant initialement soit dans une base de données soit dans l'autre. Cet exemple montre que les constructeurs de OCNC sont utiles pour les tâches d'intégration.

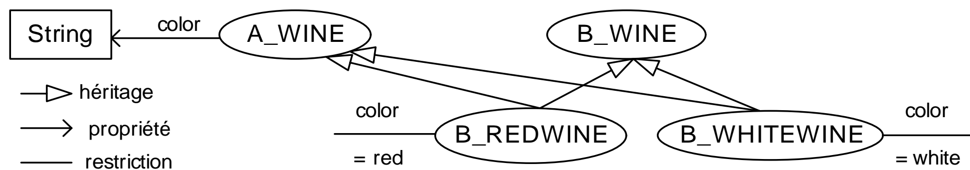


Fig. 1.2 – OCNC intégrée sur le domaine des vins

2.4.4 Ontologies Linguistiques (OL)

Dans d'autres domaines d'application comme la fouille d'information ou le traitement du langage naturel, le langage humain joue un rôle clé. Même dans les bases de données, le langage naturel est utilisé à différents endroits. En effet, les noms des tables et des attributs sont choisis pour refléter leur signification. De plus, la documentation d'un modèle conceptuel est largement, et dans de nombreux cas totalement, exprimée dans une langue naturelle.

Nous appelons *Ontologies Linguistiques* (OL), les ontologies qui définissent l'ensemble des termes qui apparaissent dans la description langagière d'un domaine. Dans cette catégorie d'ontologies, outre les relations entre concepts représentées par des termes (par exemple, `subsumée_par` pour la relation de subsomption), des relations entre les termes (par exemple, la synonymie ou l'homonymie) doivent être également définies. Les relations entre les termes étant souvent fortement contextuelles, les inférences automatiques basées sur ces ontologies nécessitent, en général, la supervision d'un expert.

Les OL aident à reconnaître des similarités conceptuelles entre des phrases même si différents termes sont utilisés. Néanmoins, puisque la signification des termes est contextuelle et que les relations entre termes sont approximatives, de fausses similarités peuvent être produites et les résultats ne peuvent jamais être considérés comme fiables. Un exemple d'utilisation de OL est donné dans [Das et al., 2004]. Une OL sur les types de cuisine est utilisée pour retourner le plus de résultats significatifs possibles aux requêtes recherchant des restaurants servant un type de nourriture donné. Par exemple, une requête recherchant les restaurants servant de la nourriture `latin/american`, retournera les restaurants correspondant à des tuples ayant `american` ou `mexican` comme valeur pour l'attribut `type_de_cuisine`. Les OL sont également largement utilisées dans les approches semi-automatiques d'intégration de sources de données. Par exemple, dans le projet MOMIS [Beneventano et al., 2000], une OL est utilisée pour construire semi-automatiquement le schéma global du système intégré à partir des différentes sources de données. Ce processus doit néanmoins être entièrement validé par un expert humain.

2.4.5 Une vision des ontologies de domaine spécifique à chaque discipline

Actuellement, les trois catégories d'ontologies que nous venons de définir sont principalement utilisées dans trois différentes disciplines informatiques.

La définition et l'exploitation d'OCC sont principalement réalisées dans les domaines de l'échange de données et des bases de données. Dans les OCC, les définitions se focalisent sur l'identification et la caractérisation des concepts primitifs. Les OCC incluent des descriptions complexes et précises de ces concepts. Ces descriptions sont fournies par des constructeurs orientés OCC permettant, par exemple, de représenter le contexte dans lequel chaque élément d'une ontologie est défini [Pierra, 2003]. Par contre, elles n'incluent pas d'opérateurs permettant de définir des équivalences conceptuelles utiles pour définir des mappings entre des ontologies. En conséquence, la représentation de ces mappings est codée dans des programmes.

Le problème de la définition et de l'exploitation de OCNC est principalement traité par la communauté Intelligence Artificielle. En effet, la définition d'équivalences conceptuelles sert de support à de nombreux mécanismes d'inférence. Les OCNC contiennent des définitions conservatives de concepts définis utilisant des opérateurs de la logique tels que les opérateurs booléens (par exemple, l'union ou intersection de classes). Par contre, en règle générale, elles incluent des descriptions moins précises des concepts primitifs se limitant généralement à un libellé et un commentaire pour décrire les classes et les propriétés.

Le problème de la définition et de l'exploitation de OL est principalement traité par les communautés linguistique informatique et recherche d'information. Dans les OL, dont WordNet[4] constitue un cas limite car elle n'est pas associée à un domaine précis, chaque terme est associé à plusieurs *synsets* (ensemble de synonymes) qui reflètent ses différentes significations. L'imprécision de cette conceptualisation est due au fait que la signification des termes dépend du contexte et que les relations entre les mots (par exemple, la similarité) n'ont pas de définition formelle alors que les relations entre les concepts (par exemple, la subsomption) en ont une.

Le constat que chaque catégorie d'ontologies correspond à une vision spécifique à chaque discipline, nous a amené à étudier les liens possibles entre ces différentes catégories.

2.5 Relation entre les différentes catégories d'ontologies : le modèle en oignon

A partir de notre description des différentes catégories d'ontologies, nous avons identifié les relations suivantes entre les OCC, OCNC, et OL :

- les mappings entre OCC peuvent être également définis avec des opérateurs d'équivalences conceptuelles des OCNC ;
- les OCNC peuvent utiliser les constructeurs orientés OCC pour définir leurs concepts primitifs ;
- les OL peuvent définir les différentes significations de chaque mot d'un langage naturel particulier en référençant une OCNC. Cette référence fournirait une base formelle pour effectuer des inférences et des traductions automatiques de termes spécifiques à un contexte donné.

[4] `http://www.cogsci.princeton.edu/wn`

Pour aller plus loin dans cette observation, nous avons proposé un modèle en couches pour la conception d'ontologies qui permet d'intégrer ces différentes catégories d'ontologies [Jean et al., 2007d].

2.5.1 Un modèle en couches pour la conception d'ontologies

Un guide souvent utilisé pour le développement de OCNC [Noy and McGuinness, 2001] propose une approche de conception en sept étapes suivantes.

1. Déterminer le domaine et l'étendue de l'ontologie à développer.
2. Considérer la réutilisation d'une ontologie existante sur le même domaine.
3. Lister les termes importants dans cette ontologie sans se préoccuper du possible chevauchement des concepts qu'ils peuvent induire.
4. Définir les classes et la hiérarchie de classes à partir de la liste créée dans l'étape 3 en sélectionnant les termes qui décrivent des objets ayant une existence indépendante plutôt que ceux qui décrivent ces objets.
5. Définir les propriétés associées aux classes avec les termes restants.
6. Définir les contraintes (cardinalité, domaine et codomaine, restriction de valeurs) qui s'appliquent sur les propriétés.
7. Créer les instances des classes de la hiérarchie.

Cette approche exploite la capacité des OCNC de définir des équivalences conceptuelles et ainsi d'intégrer plusieurs ontologies traitant le même domaine. Puisque nous pensons que les OCNC peuvent bénéficier du fait d'être articulées avec une OCC, nous proposons une approche alternative pour le développement d'une OCNC en commençant par la conception d'une OCC.

1. La première étape dans la conception d'une ontologie peut être de se mettre d'accord au sein d'une communauté sur une OCC. Pour atteindre cet accord, les étapes suivantes sont requises.
   – Identifier clairement quel est le domaine couvert par cette ontologie en s'appuyant sur le savoir-faire des experts du domaine.
   – Choisir un modèle permettant de définir précisément les concepts primitifs (classes et propriétés) existant dans le domaine et permettant d'indiquer le contexte d'évaluation des valeurs des propriétés.
   – Fournir des descriptions partagées de l'ensemble de ces concepts primitifs couvrant le domaine considéré. Cette conceptualisation doit permettre de satisfaire des besoins techniques et métiers larges et diversifiés partagés par les membres de cette communauté. Elle doit atteindre une large acceptation et reconnaissance.
2. Au sein de la communauté d'utilisateurs et/ou de développeurs, sur la base de l'OCC définie, une OCNC peut être construite afin d'être utilisée par les membres de cette communauté, soit pour construire leur propre point de vue du domaine, soit pour modéliser formellement sous forme de classes certains concepts existants dans le domaine cible et qui s'expriment sous forme de combinaisons de concepts canoniques. Procéder de cette façon, assure de conserver la capacité d'échanger et de partager les informations exprimées en termes de l'OCC.

3. Afin de permettre l'utilisation de l'OCNC définie pour les inférences linguistiques et/ou pour fournir une interface utilisateur conviviale dans différents langages naturels, associer à chaque concept de l'OCNC une liste de termes spécifiques pour une ou plusieurs langues naturelles données. Ces relations entre termes et concepts peuvent éventuellement être qualifiées (facteur de vraisemblance, etc.).

La figure 1.3 illustre un modèle en couches que nous nommons le *modèle en oignon* d'une ontologie de domaine résultant de cette approche alternative. Une OCC fournit une base formelle pour modéliser et échanger efficacement la connaissance d'un domaine. Une OCNC fournit les mécanismes pour lier différentes conceptualisations faites sur ce domaine. Finalement, les OL fournissent une représentation en langage naturel des concepts de ce domaine, éventuellement dans les différents langages où ces concepts sont significatifs.

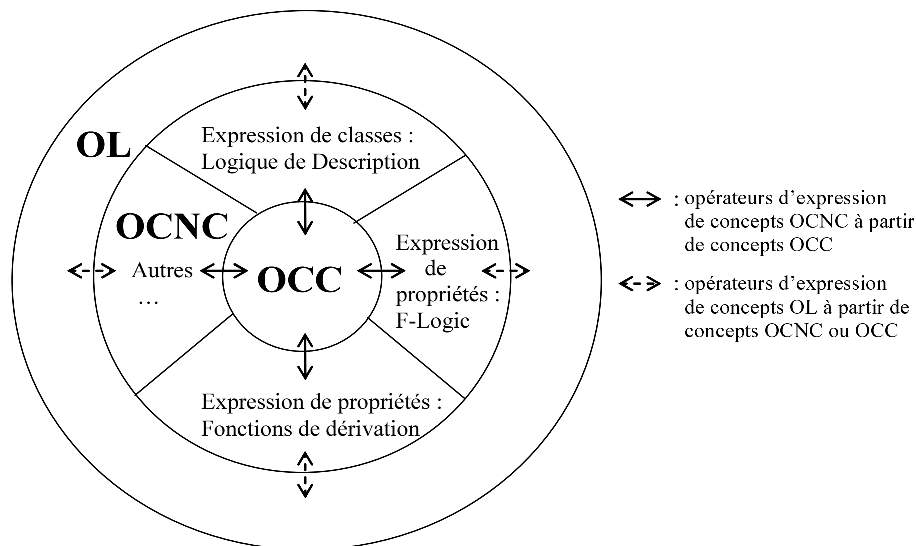


Fig. 1.3 – Le modèle en oignon pour les ontologies de domaine

Pour montrer l'intérêt du modèle en oignon, nous présentons, dans la section suivante, un exemple d'un scénario d'échange basé sur une ontologie conçue selon ce modèle.

2.5.2 Un scénario d'échange basé sur une ontologie en couches

Dans l'univers des bases de données, chaque base de données utilise un vocabulaire canonique. Généralement, chaque base de données utilise un vocabulaire canonique différent. Notre scénario consiste à échanger des données entre différentes sources de données utilisant un vocabulaire canonique différent.

Plutôt que de définir une OCNC couvrant tous les termes de toutes les sources (approche illustrée sur la figure 1.4 (A)), le modèle en oignon suggère que tous les échanges soient effectués en utilisant une OCC consensuelle. Chaque source contient simplement les descriptions de ses propres concepts en termes des concepts primitifs de la OCC (approche illustrée sur la figure 1.4 (B)). Cette approche a été mise en application pour l'intégration de bases de données dans [Bellatreche et al., 2004]. Notons que si chaque concept est représenté différemment dans les *n* sources participant à l'échange, et, s'il existe

dans chaque source un moyenne de *p* concepts, la solution (A) requiert d'implanter *n*p* mappings dans chaque source, alors que la solution (B) requiert seulement *p* mappings dans chaque source.

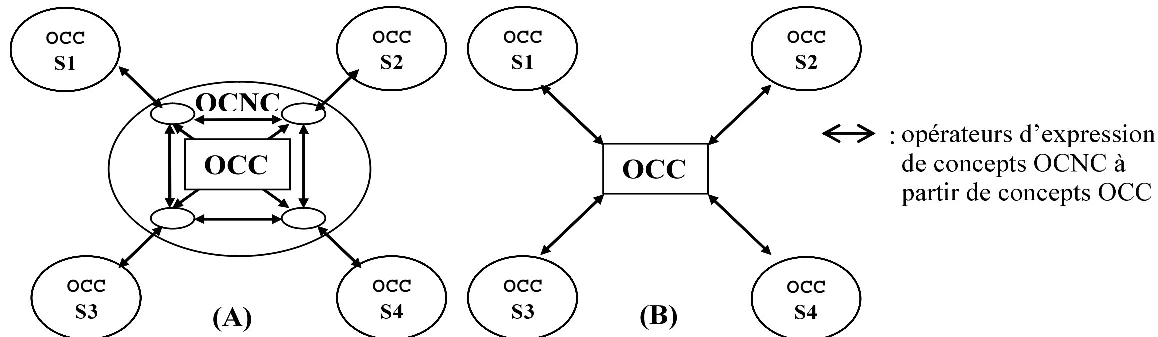


Fig. 1.4 – Utilisation d'ontologies pour l'échange canonique de données

Ce scénario et la démarche de conception proposés dans la section précédente montrent l'intérêt d'articuler les trois catégories d'ontologies selon le modèle en oignon tout au long du cycle de vie des ontologies de domaine. En effet, chaque catégorie d'ontologies offre des capacités particulières :

– les OCC fournissent une description canonique et précise de chaque concept d'un domaine donné. Elles fournissent une base solide pour l'échange entre différentes sources d'information ;

– les opérateurs des OCNC sont utilisés pour interagir avec ou intégrer d'autres applications ou sources ayant déjà leur propre ontologie ;

– les OL offrent des capacités linguistiques sur l'ensemble des concepts (primitifs et définis) du domaine.

Nous montrons dans la section suivante en quoi ces caractéristiques sont intéressantes pour les bases de données.

2.6 Liens entre le modèle en oignon et les bases de données

Chaque couche du modèle en oignon peut être utilisé pour résoudre différents problèmes des bases de données.

– Les OCC sont des modèles conceptuels formels et partageables. Ils peuvent être utilisés comme base pour la conception d'un modèle logique de base de données ou comme schéma global dans un scénario d'intégration de bases de données.

– Les OCNC proposent des mécanismes similaires aux vues des bases de données, avec une théorie formelle offrant des capacités d'inférence. Ces mécanismes peuvent être utilisés pour réaliser le mapping entre différents schémas de bases de données.

– Les OL peuvent être utilisées pour localiser les similitudes existantes entre plusieurs schémas de bases de données [Beneventano et al., 2000], pour documenter les bases de données gérées dans les Systèmes de Gestion de Bases de Données (SGBD) ou pour enrichir le langage de dialogue personne/SGBD.

2.7 Synthèse de notre interprétation des ontologies de domaine

Dans cette section nous avons analysé le concept d'ontologie de domaine dans une perspective d'exploitation de bases de données. Le terme ontologie étant souvent utilisé comme un nouveau terme pour des modèles existants, nous avons d'abord cherché à caractériser la spécificité de la notion d'ontologie et nous avons proposé pour cela de retenir trois critères. Une ontologie de domaine doit être *formelle*, c'est-à-dire permettant d'effectuer des raisonnements automatiques et d'en vérifier la consistance, *consensuelle* dans une communauté et *référençable* à partir de n'importe quel environnement. Ces trois éléments caractérisent les ontologies de domaine comme un nouveau modèle en informatique. Ceci nous a conduit à en proposer une nouvelle définition qui les distingue plus clairement des autres modèles existants : « *une ontologie de domaine est un dictionnaire formel et consensuel des catégories et propriétés d'entités existant dans un domaine d'étude et des relations qui les lient* ».

Les travaux sur les ontologies de domaine ayant été principalement développés par trois communautés, nous avons également proposé une taxonomie des ontologies de domaine en OCC, OCNC, et OL. Après avoir revu la couverture partielle de ces différents modèles, nous avons proposé un modèle en couches, nommé le modèle en oignon, d'une ontologie de domaine qui permet de concevoir et d'utiliser les capacités de chaque catégorie d'ontologies dans un environnement intégré.

Ayant décrit notre interprétation du concept d'ontologie, nous y ferons fréquemment référence dans ce mémoire, et nous caractériserons les ontologies manipulées par référence au modèle en oignon. Dans la section suivante, nous discutons les différents formalismes ou modèles proposés pour représenter des ontologies et nous les situons par rapport au modèle en oignon.

3 Les modèles d'ontologies

La représentation d'une ontologie requiert un modèle d'ontologies qui offre les primitives nécessaires pour exprimer les entités et relations qu'elle contient et les opérateurs pour les traiter. De nombreux modèles ont été proposés dans la littérature. Ils sont issus de différents champs disciplinaires dont les principaux sont les logiques de description, la logique des frames et les bases de données. Pour définir le modèle d'ontologies sur lequel baser notre travail, nous avons choisi d'analyser les constructeurs offerts par des modèles d'ontologies issus de ces différents champs disciplinaires. Nous avons choisi d'étudier le modèle PLIB [ISO13584-42, 1998, ISO13584-25, 2004] issu du domaine des bases de données, les modèles OWL [Dean and Schreiber, 2004] et RDF-Schema [Brickley and Guha, 2004] issus respectivement des logiques de description et des réseaux sémantiques et le modèle F-Logic [Kifer et al., 1995] issu de la logique des frames. Nous présentons dans les sections suivantes ces modèles d'ontologies en classant les ontologies qu'ils permettent de définir suivant les différentes couches du modèle en oignon.

3.1 Modèles d'ontologies orientés vers la définition de OCC

Dans cette section, nous présentons les modèles d'ontologies RDF-Schema et PLIB qui permettent essentiellement de définir des OCC.

3.1.1 RDF / RDF-Schema

RDF [Manola and Miller, 2004] est le premier langage apparu pour définir la sémantique de sources WEB. Sa structure est issue des réseaux sémantiques introduits dans le domaine de la représentation de connaissances. RDF permet d'exprimer des assertions dont la structure est un triplet (*sujet*, *prédicat*, *objet*). Les éléments de ce triplet sont définis comme suit :

- le *sujet* est un URI (Uniform Resource Identifier) identifiant un élément nommé *ressource*. Une ressource peut être une page Web, une partie d'une page Web référencée par une ancre ou même un objet quelconque auquel un URI a été attribuée ;
- le *prédicat* est une propriété permettant de caractériser la ressource. Par exemple une page Web peut être caractérisée par le prédicat `créé_par` afin d'en définir l'auteur ;
- l'*objet* est la valeur prise par la propriété. Cette valeur peut être un URI ou une valeur *littérale* qui peut être non typée ou typée par l'un des types primitifs définis dans XML Schema.

Puisque l'objet d'un triplet peut être le sujet d'un autre, les triplets peuvent être reliés entre eux. En représentant les sujets et objets par des noeuds et les prédicats par des arrêtes, des données RDF se représentent sous la forme d'un graphe. Au niveau sémantique, chaque triplet exprime une assertion. En conséquence, la signification d'un tel graphe est l'ensemble de ces assertions.

RDF n'introduit que très peu de prédicats prédéfinis, ceux-ci pouvant être définis librement. En conséquence, il ne propose pas de constructeurs prédéfinis pour la conception d'ontologies. C'est pourquoi il a été rapidement étendu par un ensemble de constructeurs permettant la définition d'ontologies. Cette extension de RDF porte le nom de RDF-Schema [Brickley and Guha, 2004]. Les constructeurs (relations ou prédicats) introduits par RDF-Schema sont les suivants.

- **Constructeur de classes.** `rdfs:Class` et `rdfs:subClassOf` permettent de définir des classes et de les organiser dans une hiérarchie dont les liens sont des relations de subsomption. L'héritage multiple[5] est autorisé. Ces classes sont identifiées par un URI comprenant un espace de noms afin de pouvoir les référencer de manière unique (RDF-Schema respecte donc bien le critère de référençabilité). `rdfs:label` et `rdfs:comment` permettent respectivement de donner des noms aux classes et de les décrire de façon textuelle.
- **Constructeur de propriétés.** `rdfs:domain` et `rdfs:range` permettent respectivement de définir le domaine et le codomaine d'une propriété RDF (`rdf:Property`). Le domaine d'une propriété RDF-Schema est optionnel. Lorsqu'il n'est pas défini, cette propriété peut être utilisée pour décrire n'importe quelle instance appartenant à l'univers du discours conceptualisé par l'ontologie. Un domaine peut être défini par une ou plusieurs classes. Si plusieurs classes sont utilisées, le domaine est composé des instances qui appartiennent à l'intersection de ces classes. La définition du codomaine d'une propriété est similaire à celle de son domaine à part qu'en plus des classes, des types de données sont disponibles pour le définir.

  Comme les classes, les propriétés peuvent être d'une part décrites de façon textuelle en utilisant `rdfs:label` et `rdfs:comment` et sont, d'autre part, organisées dans une hiérarchie par des liens de subsomption en utilisant `rdfs:subPropertyOf`. Si une propriété p_{sub} est une sous-propriété d'une propriété p, alors l'extension de p_{sub} est contenue dans celle de p, l'extension d'une pro-

[5] Par abus de langage, nous utilisons ici le terme héritage pour signifier la subsomption.

priété p étant définie comme l'ensemble des couples (i, v) où l'instance i à la valeur v pour la propriété p. Par exemple, la propriété est_delegue_de est une sous-propriété de est_membre_de puisque tout délégué d'une formation est membre de celle-ci.

– **Constructeur de types de données.** Les valeurs d'une propriété peuvent être des instances de classes ou des littéraux. Ces littéraux peuvent être typés en utilisant les types prédéfinis de XML Schema. Ceci permet par exemple de représenter des valeurs de types chaîne de caractères, numérique ou date. Par ailleurs, RDF-Schema fournit également des types collections (rdfs:Container). RDF-Schema propose des constructeurs pour construire différentes sortes de collections et en particulier les listes (rdf:List) et les sacs (rdf:Bag).
– **Constructeur d'instances.** Les instances sont définies, en RDF, par deux types de triplets. Les triplets de la forme (i, rdf:type, C) indiquent que i est une instance de la classe C. RDF-Schema supporte la *multi-instanciation* qui permet à une instance d'appartenir à plusieurs classes même si ces classes ne sont pas liées par une relation de subsomption. Les autres triplets, de la forme (i, p, v), caractérisent l'instance i par la valeur v pour la propriété p.

L'utilisation conjointe de RDF et RDF-Schema permet donc à la fois de représenter (en RDF-Schema) une ontologie et (en RDF) des instances définies en termes de cette ontologie.

Exemple. Les figures 1.5 et 1.6 illustrent cette capacité. Nous avons représenté sur ces figures un extrait de l'ontologie SIOC[6] conçue, entre autres, par un des auteurs de RDF-Schema. Cet exemple d'ontologie sera utilisé tout au long de ce mémoire.

Explication. Cette ontologie est représentée dans la figure 1.5 sous la forme d'un graphe. Les concepts principaux de cette ontologie sont les suivants. Un forum (Forum) est hébergé sur un site (Site). Il est administré par un modérateur (has_moderator). Des utilisateurs (User) peuvent s'abonner à des forums (subscriber_of) et créer des messages (Post) sur ces forums (has_container). Plusieurs réponses peuvent être données à un message (has_reply).

En dessous de cette ontologie, nous avons représenté des données de cette ontologie. Les URI des instances sont entourées d'un ovale tandis que les valeurs littérales sont représentées dans un rectangle. Dans cette partie, une instance de la classe Post est décrite. L'URI de ce message se termine par post-sioc, son titre et son contenu sont définis par des valeurs littérales tandis que son créateur, le forum sur lequel il est créé ainsi que les réponses qui y ont été faites sont définis en référençant d'autres instances.

Enfin, sur la figure 1.6, nous avons indiqué un cours extrait de cette ontologie en RDF/XML, un format XML pour des ontologies RDF-Schema. Nous voyons notamment grâce à cette syntaxe que les classes et propriétés de cette ontologie sont décrites par un nom (<rdfs:label>) et un commentaire (<rdfs:comment>).

Même si ce n'est pas illustré dans cet exemple, le langage RDF-Schema laisse un grand degré de liberté pour représenter une ontologie. En particulier, RDF-Schema n'impose aucune séparation entre les niveaux instance, ontologie et modèle d'ontologies. Ainsi, il permet par exemple qu'une instance soit également une classe et donc, qu'elle puisse elle-même avoir des instances. Il permet aussi qu'une propriété soit également une classe et donc qu'elle puisse être elle-même le domaine d'autres propriétés.

[6] http://sioc-project.org/

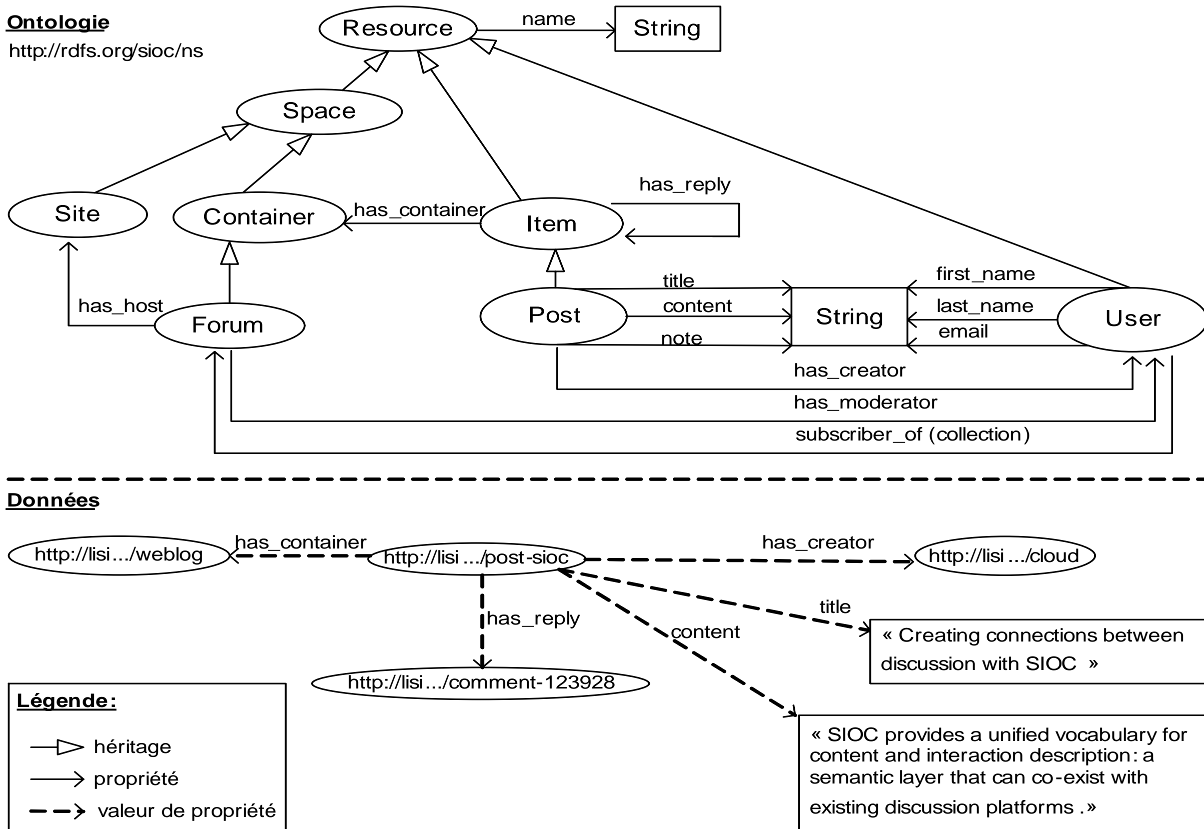
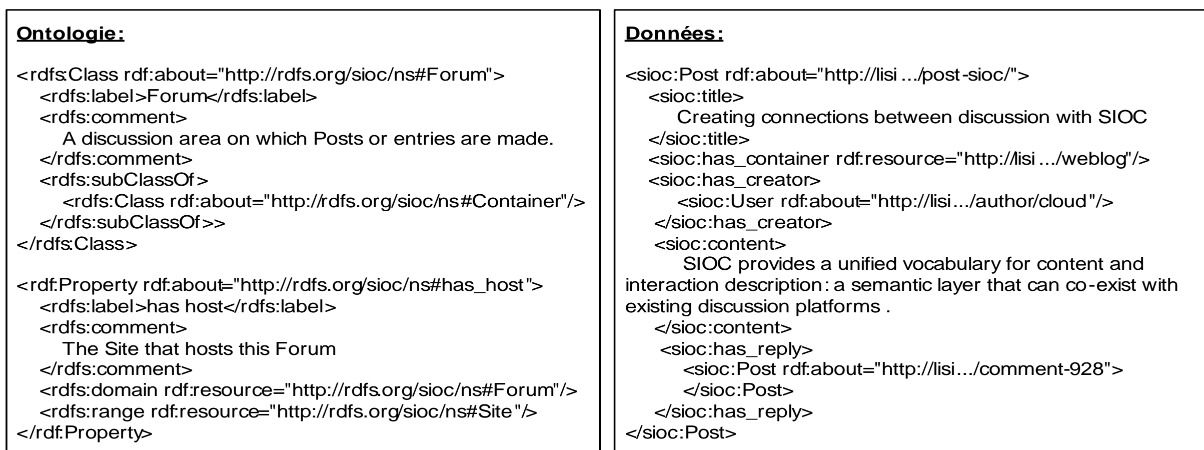


FIG. 1.5 – Un extrait de l'ontologie SIOC représentée sous forme graphique



Ontologie:

```
<rdfs:Class rdf:about="http://rdfs.org/sioc/ns#Forum">
   <rdfs:label>Forum</rdfs:label>
   <rdfs:comment>
      A discussion area on which Posts or entries are made.
   </rdfs:comment>
   <rdfs:subClassOf>
      <rdfs:Class rdf:about="http://rdfs.org/sioc/ns#Container"/>
   </rdfs:subClassOf>>
</rdfs:Class>

<rdf:Property rdf:about="http://rdfs.org/sioc/ns#has_host">
   <rdfs:label>has host</rdfs:label>
   <rdfs:comment>
      The Site that hosts this Forum
   </rdfs:comment>
   <rdfs:domain rdf:resource="http://rdfs.org/sioc/ns#Forum"/>
   <rdfs:range rdf:resource="http://rdfs.org/sioc/ns#Site"/>
</rdf:Property>
```

Données:

```
<sioc:Post rdf:about="http://lisi.../post-sioc/">
   <sioc:title>
      Creating connections between discussion with SIOC
   </sioc:title>
   <sioc:has_container rdf:resource="http://lisi.../weblog"/>
   <sioc:has_creator>
      <sioc:User rdf:about="http://lisi.../author/cloud"/>
   </sioc:has_creator>
   <sioc:content>
      SIOC provides a unified vocabulary for content and
interaction description: a semantic layer that can co-exist with
existing discussion platforms .
   </sioc:content>
   <sioc:has_reply>
      <sioc:Post rdf:about="http://lisi.../comment-928">
      </sioc:Post>
   </sioc:has_reply>
</sioc:Post>
```

FIG. 1.6 – Un extrait de l'ontologie SIOC représentée dans le format RDF/XML

Notons que RDF-Schema ne contient aucune primitive explicite permettant de décrire des équivalences conceptuelles[7]. C'est la raison pour laquelle nous considérons que ce modèle est orienté vers les OCC. Dans la section suivante nous présentons le modèle PLIB également spécialisé dans la définition de OCC.

3.1.2 PLIB

PLIB (Parts LIBrary) est un projet initié en 1987 au niveau européen [ISO13584-42, 1998, ISO13584-25, 2004] et élevé au niveau ISO en 1990. Son objectif était de permettre une modélisation informatique des catalogues de composants industriels afin de les rendre échangeables entre fournisseurs et utilisateurs. Pour atteindre cet objectif, le langage de modélisation EXPRESS a été utilisé pour représenter un catalogue sous forme d'instances d'un modèle. Un tel catalogue propose une classification des classes de composants industriels représentés, ainsi qu'une description de toutes les propriétés s'appliquant à celles-ci. En conséquence, un catalogue contient à la fois une ontologie permettant de représenter une partie des concepts présents dans un domaine inclus dans celui des composants industriels et des instances de cette ontologie. Le modèle publié en 1998 [ISO13584-42, 1998] et complété récemment par des extensions dans les normes ISO 13584-24 et -25 est donc un modèle d'ontologies et un modèle d'instances de classes des ontologies.

Partant du constat que les termes primitifs d'un domaine technique sont très nombreux et difficiles à appréhender, l'objectif du modèle PLIB est de permettre de définir de tels concepts avec la plus grande précision possible. Le modèle PLIB permet ainsi de créer des OCC avec les constructeurs suivants.

- **Constructeur de classes.** PLIB propose le constructeur `item_class` pour définir des classes. Ces classes, tout comme les autres éléments d'une ontologie, sont identifiées par un *BSU* (Basic Semantic Unit). Ce BSU est construit à partir de l'identifiant universel de l'organisation qui est la source de l'ontologie (`supplier`). Les classes PLIB peuvent être décrites par une description textuelle (nom, noms synonymes, définition, note, remarque) éventuellement dans plusieurs langues naturelles. Cette description peut aussi être complétée par des documents. Deux opérateurs sont disponibles pour organiser ces classes dans une hiérarchie. Le premier est l'opérateur `is_a`. Cet opérateur est l'opérateur d'héritage simple qui implique l'héritage des propriétés définies par les classes subsumantes. Le second opérateur, nommé `case_of`, permet la subsomption multiple et n'implique pas implicitement l'héritage de propriétés. Pour hériter d'une propriété d'une de ses classes subsumantes, une classe doit explicitement indiquer l'importation de cette propriété dans sa définition. Une classe peut donc hériter d'une partie des propriétés des classes subsumantes. L'objectif de cet opérateur est de permettre la construction modulaire d'ontologies de domaine.
- **Constructeur de propriétés.** Les propriétés en PLIB sont définies avec le constructeur `non_dependent_pdet`. Comme les classes, elles sont identifiées par un BSU et décrites par une définition textuelle et/ou graphique. Le champ d'application d'une propriété est défini par son domaine (`scope`). En pratique, ce champ d'application est souvent composé de plusieurs classes qui peuvent se trouver dans plusieurs branches de la hiérarchie. Pour résoudre ce problème, PLIB propose de qualifier les propriétés de la façon suivante :

[7] Il est à noter qu'il est possible d'utiliser la relation de subsomption de RDF-Schema pour définir des équivalences conceptuelles. Par exemple, si une classe c_1 est subsumée par c_2 et que c_2 est subsumée par c_1, alors c_1 et c_2 sont équivalentes.

  - une propriété est définie sur une classe `C` au plus haut niveau de la hiérarchie où elle peut être définie sans ambiguïté ; elle est dite *visible* pour toutes les sous-classes de `C` ;
  - une propriété visible sur une classe `C` peut devenir *applicable* sur cette classe, c'est-à-dire que toute instance de `C` doit présenter une caractéristique ou une grandeur qui représente cette propriété.

  PLIB propose un autre constructeur de propriété, nommé `dependent_P_DET`, pour les propriétés dont l'évaluation dépend de paramètres de contexte (`condition_DET`). Par exemple, la durée de vie d'un roulement à bille dépend de la charge qu'il supporte ainsi que de la vitesse à laquelle il est utilisé. De même, la résistance d'un transistor dépend de la température à laquelle elle est mesurée.

- **Constructeur de types de données.** Le modèle PLIB fournit des types de données primitifs tels que les entiers ou les réels. Ces derniers peuvent être associés à une unité de mesure (`measure_type`) ou à une monnaie (`currency_type`). Le type `class_instance_type` permet d'utiliser une classe comme un type de données. PLIB fournit également le type `aggregate_type` pour pouvoir créer des collections. Plusieurs types collections sont disponibles comme par exemple les listes, les tableaux ou les ensembles. La cardinalité de ces collections peut être définie. Enfin, le modèle d'ontologies PLIB permet de créer des types de données utilisateurs comme par exemple des types énumérés (`non_quantitative_code_type`).
- **Constructeur d'instances.** Une instance PLIB est définie par sa classe de base et l'ensemble de ses valeurs de propriétés. PLIB ne permet donc pas la multi-instanciation. A la place, PLIB offre le mécanisme d'agrégation d'instance. Pour cela, PLIB propose de distinguer les propriétés essentielles d'une classe de celles qui dépendent d'un point de vue sur le concept représenté. Cette distinction est mise en place via les trois constructeurs de classes suivants :
  - les *classes de définition* (`item_class`) qui contiennent les propriétés essentielles d'une classe ;
  - les *classes de représentation* (`functional_model_class`) qui contiennent les propriétés qui n'ont de sens que par rapport à un point de vue ;
  - les *classes de vue* (`functional_view_class`) qui définissent la perspective ou le point de vue selon lequel sont définies les propriétés des classes de représentation.

  Ainsi, une instance peut appartenir non seulement à sa classe de base mais également aux classes de représentation attachées à cette classe de base par la relation `is_view_of`. Pour éviter toute redondance au niveau des valeurs des propriétés des instances, les propriétés définies sur la classe de base et sur les classes de représentation sont disjointes.

  Remarque. Il est à noter que, contrairement à la modélisation orientée-objet, une instance n'a pas besoin de fournir une valeur pour l'ensemble des propriétés applicables sur sa classe de base. Ainsi, une propriété applicable peut ou non être *utilisée* dans la représentation d'une instance particulière dans un univers formel particulier (base de données, échange informatisé, etc.).

Exemple. L'exemple suivant présente l'ajout de la classe `Administrator` à l'ontologie SIOC en utilisant le modèle PLIB.

```
Item_class Administrator (
```

```
is_case_of : User imported_properties : first_name, last_name, email
definition : 'a person responsible for the administration of forums'
remark : 'an administrator is allowed to banish an user from a forum'
icon : 'logo_administrator.jpg'
dependent_P_DET :
  salary : real_currency_type currency : .USD. depends_on : date
condition_DET :
  date : string_type )
```

Explication. Les ontologies PLIB sont habituellement représentées sous la forme de fichiers physiques EXPRESS. Dans cet exemple, nous utilisons notre propre syntaxe par soucis de lisibilité. Ainsi, la classe `Administrator` est définie comme étant subsumée par la classe `User` en utilisant l'opérateur `case_of`. L'utilisation de cet opérateur permet de n'importer que les propriétés de la classe `User` pertinentes pour la classe `Administrator`. Ainsi, la propriété `subscriber_of` n'a pas été importée, considérant que l'on ne s'intéresse pas aux forums auxquels l'administrateur d'un forum particulier peut, par ailleurs, être abonné. La classe `Administrator` est décrite par une définition et une remarque. Une icône lui est également associée sous la forme d'une image. Cette icône pourra, par exemple, être affichée lorsqu'un administrateur interviendra sur un forum. La classe `Administrator` est le domaine de définition de la propriété `salary`. Le codomaine de cette propriété est le type réel associé à une unité de monnaie (`real_currency_type`). L'unité de mesure associée à ce type est le dollar (code `.USD.`). Les valeurs de cette propriété dépendent de la date à laquelle le salaire d'un administrateur est considéré. Ceci est représenté en PLIB en définissant `salary` comme une propriété dépendante du paramètre de contexte `date`.

Le modèle d'ontologies PLIB permet donc de définir avec précision des concepts primitifs. Sa spécificité est de permettre de représenter le contexte dans laquelle une ontologie est définie [Pierra, 2007, Pierra, 2003]. Dans une ontologie PLIB, la représentation du contexte est exprimée aux trois niveaux suivants :

– explicitation du contexte de définition des classes et des propriétés. Une propriété doit être définie dans le contexte d'une classe et, afin de minimiser sa sensibilité au contexte, une classe doit définir l'ensemble des propriétés essentielles pour ses instances. Par contre, une instance d'une classe ne fournit pas nécessairement de valeur à toutes les propriétés de la classe ;

– explicitation du point de vue selon lequel un objet du mode réel est défini. Un même objet peut être représenté par plusieurs instances de classes correspondant aux différents points de vue. Chaque instance utilise des propriétés définies sur la classe correspondant à son point de vue ;

– explicitation du contexte d'évaluation des valeurs. Les valeurs permettant de caractériser les instances de classes peuvent être associées à des unités de mesure ainsi qu'à des paramètres influant sur leur évaluation.

Comme RDF-Schema, PLIB ne contient aucun opérateur permettant de définir des équivalences conceptuelles.

3.2 Modèles d'ontologies orientés vers la définition de OCNC

Les modèles d'ontologies permettant la définition de OCNC fournissent des constructeurs pour construire des concepts définis dans une ontologie à partir d'autres concepts primitifs et/ou définis. Nous présentons dans les sections suivantes les modèles d'ontologies OWL et F-Logic qui offrent cette possibilité de différentes manières. Le modèle PLIB est en cours d'extension pour supporter cette possibilité.

3.2.1 OWL

Afin d'étendre le pouvoir d'expression de RDF-Schema, deux initiatives ont été lancées. La première est une initiative américaine lancée par la DARPA (Defense Advanced Research Projects Agency) qui a permis la spécification du langage DAML-ONT [Stein et al., 2000]. La seconde est une initiative sponsorisée par la communauté européenne qui a permis la spécification du langage OIL [Fensel et al., 2001]. La fusion de ces deux langages a donné naissance à DAML+OIL [Connolly et al., 2001] qui a servi de base pour la conception du langage OWL standardisé par le W3C.

Un objectif important de OWL est de pouvoir assurer que les raisonnements qui peuvent être réalisés sur des ontologies conçues avec un sous-ensemble de ce langage soit *décidables*, c'est-à-dire qu'il soit possible de concevoir un algorithme qui puisse garantir de déterminer, dans un temps fini, si oui ou non les définitions fournies par une ontologie OWL peuvent être déduites à partir de celles d'une autre ontologie [Horrocks et al., 2003]. L'impossibilité d'obtenir un langage à la fois compatible avec RDF-Schema et dont les raisonnements associés soit décidables a poussé les concepteurs de OWL à en spécifier trois versions : OWL Lite, OWL DL et OWL Full. Ces trois versions offrent une expressivité croissante (OWL Lite ⊂ OWL DL ⊂ OWL Full). Les raisonnements associés à OWL Lite et OWL DL sont décidables mais ne sont pas, par contre, compatibles avec RDF-Schema car ils restreignent les capacités des constructeurs de ce modèle. Par exemple, OWL Lite et OWL DL ne permettent pas qu'une classe soit une instance d'une autre classe ce qui est possible en RDF-Schema. En conséquence, toutes ontologies RDF-Schema n'est pas forcément une ontologie OWL Lite ou OWL DL. A l'opposé OWL Full est compatible avec RDF-Schema mais les raisonnements associés ne sont pas forcément décidables. Nous décrivons dans ce qui suit les constructeurs offerts par OWL DL. Nous préciserons ensuite les limitations de OWL Lite par rapport à ces constructeurs ainsi que les libertés permises par OWL Full.

- **Constructeur d'ontologies.** OWL permet de construire une ontologie comme étant un ensemble de définitions de classes et de propriétés regroupées dans un ou plusieurs espaces de noms. OWL fournit de nombreux attributs permettant de caractériser une ontologie comme par exemple `owl:versionInfo` qui indique son numéro de version.
- **Constructeur de classes.** `owl:Class` permet de définir des classes. Chaque classe est identifiée par un URI, elle est décrite de façon textuelle comme une classe RDF-Schema (`rdfs:label` et `rdfs:comment`). Ces classes peuvent être organisées dans une hiérarchie en utilisant le constructeur `rdfs:subClassOf` de RDF-Schema. Il existe deux classes prédéfinies nommées `owl:Thing` et `owl:Nothing`. La classe `owl:Thing` est la super-classe de toutes les autres classes. Ainsi, toutes les instances appartiennent à cette classe. A l'opposé, `owl:Nothing` est une sous-classe de toutes les autres classes. Elle ne possède aucune instance.

  OWL permet également de définir des *classes définies* en fournissant des *opérateurs booléens*

sur les classes (`owl:unionOf`, `owl:intersectionOf`, et `owl:complementOf`), des *restrictions* permettant de définir en intention les instances d'une classe (`owl:allValuesFrom`, `owl:someValuesFrom` et `owl:hasValue`) et un constructeur permettant de définir une classe en énumérant ses instances (`owl:oneOf`). Si `C`, `P`, `I` et `V` sont respectivement l'ensemble des classes, des propriétés, des instances et des valeurs de propriétés d'une ontologie, les signatures des constructeurs fournis pour définir des *classes définies* sont les suivantes :

- `owl` : `unionOf` : $C^n \rightarrow C$ permet de définir une classe comme étant l'union de plusieurs classes. Par exemple, la classe `Human` peut être définie comme étant l'union des classes `Man` et `Woman` ;
- `owl` : `intersectionOf` : $C^n \rightarrow C$ permet de définir une classe comme étant l'intersection de plusieurs classes. Par exemple, la classe `Man` peut être définie comme étant l'intersection des classes `Human` et `Male` ;
- `owl` : `complementOf` : $C \rightarrow C$ permet de définir une classe comme étant le complémentaire d'une autre classe. Par exemple, la classe `NonHuman` peut être définie comme étant l'ensemble des instances qui n'appartiennent pas à la classe `Human` ;
- `owl` : `allValuesFrom` : $P \times C \rightarrow C$ permet de définir une classe comme étant l'ensemble des instances qui ne prennent comme valeur d'une propriété que des instances d'une classe donnée. Par exemple, la classe `DriverOfCarOnly` peut être définie comme étant l'ensemble des instances qui ont pour valeur de la propriété `vehicle` des instances de la classe `Car` seulement ;
- `owl` : `someValuesFrom` : $P \times C \rightarrow C$ permet de définir une classe comme étant l'ensemble des instances qui prennent comme valeurs d'une propriété au moins une instance d'une classe donnée. Par exemple, la classe `DriverOfCar` peut être définie comme étant l'ensemble des instances qui ont pour valeur de la propriété `vehicle` au moins une instance de la classe `Car` ;
- `owl` : `hasValue` : $P \times V \rightarrow C$ permet de définir une classe comme étant l'ensemble des instances qui prennent une certaine valeur pour une propriété donnée. Par exemple, la classe `Male` peut être définie comme l'ensemble des instances qui prennent la valeur `M` pour la propriété `gender` ;
- `owl` : `oneOf` : $I^n \rightarrow C$ permet de définir une classe en extension, c'est-à-dire en indiquant la liste de ses instances. Par exemple, la classe `Capital` peut être définie par la liste des instances `Paris`, `London`, etc.

- **Constructeur de propriétés.** `owl:ObjectProperty` et `owl:DatatypeProperty` spécialisent le constructeur RDF-Schema `rdf:Property` pour permettre de définir des propriétés. Ces deux constructeurs sont fournis afin de distinguer les propriétés en fonction de leur codomaine. `owl:ObjectProperty` permet de définir des propriétés dont le codomaine est une classe tandis que `owl:DatatypeProperty` permet de définir une propriété dont le codomaine est un type de données. Quel que soit son type, une propriété est définie comme une propriété RDF-Schema, c'est-à-dire par un domaine (`rdfs:domain`) qui peut être l'intersection de plusieurs classes, par un codomaine (`rdfs:range`) et par sa place dans la hiérarchie des propriétés (`rdfs:subPropertyOf`). OWL permet également de qualifier une propriété. Ainsi, deux propriétés peuvent être définies

comme étant l'inverse l'une de l'autre (owl:inverseOf). Une propriété peut être définie comme étant symétrique (owl:SymmetricProperty), transitive (owl:TransitiveProperty), injective (owl:InverseFunctionalProperty) ou comme étant une fonction (owl:FunctionalProperty). La signification de ces caractéristiques est celle qui en est donnée en mathématique.

- **Constructeur de types de données.** Le type de données d'une propriété owl:DatatypeProperty peut être le type RDF-Schema rdfs:Literal ou un des types simples définis pour les XML Schéma dont la liste est donnée dans [Smith et al., 2004].
- **Constructeur d'instances.** Comme pour RDF-Schema, une instance OWL est définie en utilisant le constructeur RDF rdf:type, elle est identifiée par un URI et peut appartenir à plusieurs classes non liées par la relation de subsomption. Une instance est caractérisée par un ensemble de valeurs de propriétés.
- **Constructeur d'axiomes.** OWL permet de préciser le nombre de valeurs qu'une instance peut prendre pour une propriété donnée grâce aux axiomes owl:minCardinality, owl:cardinality et owl:maxCardinality. L'axiome owl:equivalentClass indique que deux classes sont équivalentes, c'est-à-dire qu'elles possèdent les mêmes instances. A l'inverse, l'axiome owl:disjointWith indique que les extensions de deux classes données sont disjointes, c'est-à-dire qu'elles n'ont aucune instance en commun. De même, OWL permet d'indiquer que deux propriétés sont équivalentes, c'est à dire qu'elles présentent des valeurs pour les mêmes instances et que ces valeurs sont identiques pour les mêmes instances, par l'axiome owl:equivalentProperty. En OWL, deux instances d'une ontologie ayant des identifiants différents peuvent être le même élément. Ainsi, OWL ne fait pas l'*hypothèse d'unicité des noms* qui consiste à considérer que les identifiants permettent d'identifier de manière unique un élément dans une ontologie. En conséquence, il propose les axiomes owl:sameAs, owl:differentFrom et owl:AllDifferent permettant d'indiquer l'égalité ou l'inégalité des instances.

Exemple. L'exemple suivant présente la définition de la classe PostAdministrator contenant l'ensemble des messages créés par un administrateur, et la classe PostUser contenant les autres messages.

```
Class (PostAdministrator complete Post
                    restriction (has_creator allValuesFrom (Administrator)))
Class (PostUser complete Post complementOf (PostAdministrator))
```

Explication. Cet exemple utilise la syntaxe abstraite de OWL [Patel-Schneider et al., 2004], plus lisible que la syntaxe RDF/XML. La classe PostAdministrator est définie comme étant équivalente à (complete) l'intersection de la classe Post et de l'ensemble des instances qui prennent une instance de la classe Administrator comme valeur de la propriété has_creator (PostAdministrator). La classe PostUser est définie comme étant équivalente à l'intersection de la classe Post et du complémentaire de la classe PostAdministrator.

Les constructeurs de OWL-DL, présentés précédemment, ont été choisis de manière à ce que les raisonnements associés à ce langage restent décidables. Le problème d'inférence en OWL DL a ainsi pu être classé comme étant aussi difficile que ce problème dans les langages des logiques de description de la famille $\mathcal{SHOIN}(\mathcal{D})$. Des travaux ont montré que ces langages sont de complexité au pire des cas de temps exponentiel non déterministe (*NExpTime*) [Tobies, 2001]. En pratique, il n'existe pas d'algorithme

connu pour implanter un moteur d'inférence sur de tels langages satisfaisant aux exigences usuelles en termes de qualité et temps de réponse [Horrocks et al., 2003]. C'est pour cela, que certains constructeurs ont été simplifiés ou supprimés pour obtenir une version de OWL DL allégée nommée OWL Lite. Voici ces simplifications :

– la création de classes par les opérations booléennes `owl:unionOf` et `owl:complementOf` ou par énumération de ses individus est interdite ;
– la restriction `owl:hasValue` ne peut pas être utilisée ;
– les axiomes de cardinalité `owl:minCardinality`, `owl:cardinality` et `owl:maxCardinality` sont limités à prendre la valeur 0 ou 1.

Le troisième niveau de OWL se nomme OWL Full. Il se caractérise par une compatibilité totale avec RDF et RDF-Schema. En conséquence, cette version permet de traiter une instance comme une classe. Elle autorise également l'utilisation des constructeurs OWL comme paramètres de ses propres constructeurs. Il est par exemple possible d'indiquer qu'une classe donnée ne doit pas avoir plus de 2 super-classes. Ce pouvoir d'expression fait que les raisonnements associés à OWL Full ne sont pas décidables.

Nous venons de voir que le modèle d'ontologies OWL se focalise sur la définition de concepts définis, c'est-à-dire appartenant à une OCNC. Les constructeurs offerts par ce modèle sont issus des logiques de description. Dans la section suivante, nous présentons le modèle d'ontologies F-Logic qui se focalise également dans la définition de OCNC mais, cette fois-ci, en définissant des constructeurs issus de la logique des frames.

3.2.2 F-Logic

F-Logic (Frame Logic) [Kifer et al., 1995] est un langage de modélisation de connaissances qui combine les capacités de modélisation des modèles orientés-objets avec l'expressivité de la logique. Cette combinaison permet d'obtenir un langage disposant des constructeurs suivants.

– **Constructeur de classes.** F-Logic permet de définir des classes. Ces classes peuvent être organisées dans une hiérarchie en utilisant l'héritage multiple (opérateur ::). Par exemple, pour définir que la classe `Homme` est une sous classe de la classe `Humain`, nous écrivons en F-Logic `Homme` :: `Humain`. Contrairement aux modèles d'ontologies précédents, les classes ne sont pas identifiées de manière universelle via des mécanismes comme les BSU en PLIB ou les espaces de noms en OWL. Cependant, OntoBroker[8] une des implantations les plus connues de F-Logic supporte le mécanisme d'espaces de noms.
– **Constructeur de propriétés.** F-Logic permet de définir des propriétés monovaluées (⇒) et multivaluées (⇛). Ces propriétés doivent être définies en même temps que leur classe de définition. Par exemple, l'expression `Homme[nom ⇒ string]` permet de définir la propriété `nom` définie sur la classe `Homme`. Il permet également de définir des méthodes. Les auteurs de F-Logic précisent qu'il n'y a pas de différence essentielle entre les propriétés et les méthodes, ces dernières permettent simplement de caractériser une instance par une valeur en fonction d'autres valeurs appelées *ar-*

[8]`http://www.ontoprise.de`

guments. Par exemple, l'expression Homme[salaire@date ⇒ integer] permet de définir une méthode salaire pour la classe Homme dont la valeur dépend d'une date.

- **Constructeur de types de données.** Les types de données sont utilisés en F-Logic pour définir le codomaine des propriétés ainsi que les arguments d'une méthode. F-Logic permet d'utiliser des types primitifs (integer, real, string et date) et les classes comme types de données. Il supporte les collections de type ensemble.
- **Constructeur d'instances.** Une instance F-Logic est caractérisée par les classes auxquelles elle appartient (multi-instanciation) ainsi que ses valeurs de propriétés. Par exemple, l'expression Bob : Homme[nom → "bob"] indique que l'instance de la classe Homme dont l'identifiant est Bob possède la valeur "bob" pour la propriété nom. Comme pour RDF-Schema, F-Logic permet qu'une instance soit traitée comme une classe et ainsi elle peut avoir des instances. Cette capacité est qualifiée de *méta-modélisation*.
- **Constructeur de règles** Pour offrir des capacités déductives, F-Logic est équipé d'un langage de règles. Une règle F-Logic se présente sous la forme *conclusion* ← *precondition*. La *précondition* est appelée le corps de la règle. Elle est composée d'expressions F-Logic indiquant l'appartenance d'une instance à une classe, une valeur de propriété ou une méthode. Ces expressions peuvent comporter des variables quantifiées avec l'opérateur existentiel (∃) ou universel (∀) et peuvent être composées par les opérateurs booléens (∨, ∧ et ¬). La *conclusion* est appelée la tête de la règle. Elle est également composée d'une expression F-Logic du même type que celles présentes dans la précondition. Les variables utilisées dans la tête de la règle doivent être utilisées dans la précondition. L'exécution d'une règle consiste à rechercher les valeurs des variables qui permettent de satisfaire la précondition. Les faits déduits sont ceux obtenus en remplaçant les variables de la conclusion par les valeurs qui satisfont la précondition.

Le langage de règles de F-Logic permet d'introduire des concepts définis dans une ontologie. Voici deux exemples illustrant cette capacité.

Exemple. Définir la classe PostDupont comme étant les messages écrits par l'utilisateur Dupont.

P : PostDupont ← P : Post ∧ P[has_creator → U[last_name → "Dupont"]]

Explication. La précondition de cette règle introduit la variable P instance de la classe Post. Par défaut, la quantification de cette variable est universelle. Pour chaque variable P dont le créateur (has_creator → U) est Dupont ([last_name → "Dupont"]), la conclusion est déduite. Celle-ci indique que la variable P appartient à la classe PostDupont.

Exemple. Définir la propriété author qui indique le nom de l'utilisateur qui a créé un message.

P[author → N] ← P : Post ∧ N : String ∧ P[has_creator → U[last_name → N]]

Explication. La précondition introduit les variables P et N représentant respectivement une instance de la classe Personne et une chaîne de caractères (String). Pour chaque post P qui a comme créateur U, le nom de U est recherché (N). La conclusion indique que l'auteur du message P est le nom N.

Nous venons de voir que le langage F-Logic se focalise sur la définition de OCNC en permettant d'utiliser un langage de règles. Nous abordons maintenant le cas des OL.

3.3 Traitements pour les OL

Une OL est une terminologie portant sur les catégories d'objets d'un domaine, les propriétés de ces objets ainsi que les relations qui les lient. Elle comporte le terme de classe (`classe`) et la relation de propriété (`est_propriété_de`), ce qui permet de représenter les catégories d'objets et leurs propriétés (par exemple, `nom est_propriété_de personne`). Elle comporte également des relations linguistiques entre termes telles que la synonymie, l'homonymie, l'hyperonymie ou l'hyponymie. Ce type d'ontologie apparaît dans deux cas :

- quand on veut construire l'ontologie conceptuelle d'un domaine par analyse d'un corpus de documents. Les termes sont progressivement extraits puis reliés par des relations telles que `est_synonyme_de` ou `est_propriété_de`. Une telle ontologie linguistique peut ensuite être utilisée soit pour définir une ontologie conceptuelle de domaine [Aussenac-Gilles et al., 2000], soit pour analyser/indexer des documents avec un vocabulaire contrôlé [Aussenac-Gilles and Mothe, 2004] ;
- quand on veut indexer des documents par les éléments d'une ontologie conceptuelle. Il faut alors associer aux différents éléments de cette ontologie les termes les plus fréquents de représentation de ces concepts et les termes synonymes, mais il faut aussi représenter l'homonymie pour lever l'ambiguïté des termes correspondants [Roussey et al., 2002].

Cette approche restant encore peu formalisée, chaque auteur utilise ses propres structures. Nous nous contenterons donc de présenter ici les éléments linguistiques que permettent de représenter les différents modèles d'ontologies présentés jusqu'à présent.

3.3.1 Description textuelle

Dans les différents modèles d'ontologies présentés, les classes comme les propriétés peuvent être associées à une ou plusieurs descriptions textuelles. En RDF-Schema et OWL, cette description est optionnelle. Lorsqu'elle est définie, elle est composée d'un ou plusieurs noms `rdfs:label` et d'un ou plusieurs commentaires `rdfs:comment`. En PLIB, la description textuelle d'une classe ou d'une propriété est obligatoirement composée d'un nom (`preferred_name`), d'un nom court (`short_name`) et d'une définition (`definition`). Elle peut être complétée par une note (`note`) et une remarque (`remark`). En F-Logic, seul le nom choisi pour identifier la classe ou la propriété peut être utilisé comme description textuelle.

3.3.2 Multilinguisme

La description textuelle qui peut être associée aux concepts d'une ontologie peut être définie dans plusieurs langues naturelles. RDF-Schema et OWL supportent le *multilinguisme* en permettant que les valeurs d'attributs et de propriétés de type texte puissent être définies dans plusieurs langues naturelles. Dans ces modèles d'ontologies, le multilinguisme est ainsi disponible pour décrire les ontologies et les données. Nous illustrons ces capacités dans les deux exemples suivants.

Exemple. Décrire la classe `User` de l'ontologie SIOC en français.

```
<rdfs:label xml:lang="fr">Utilisateur</rdfs:label>
<rdfs:comment xml:lang="fr">Un compte sur un site en ligne</rdfs:comment>
```

Cet exemple montre que la syntaxe RDF/XML permet d'utiliser le tag XML `xml:lang` afin de définir la valeur des attributs `rdfs:label` et `rdfs:comment` en français. Comme le montre l'exemple suivant, nous pouvons utiliser le même mécanisme pour les données.

Exemple. Donner un titre en français et en anglais au message dont l'URI se termine par `post-sioc`.

```
<sioc:title xml:lang="fr">
    Une ontologie pour les communautés en-ligne : l'ontologie SIOC
</sioc:title>
<sioc:title xml:lang="en">
    An ontology for online communities: the SIOC ontology
</sioc:title>
```

Le multilinguisme est également supporté par le modèle PLIB. Chaque ontologie PLIB indique explicitement la ou les langues naturelles dans lesquelles ces concepts sont décrits. Les valeurs des attributs de type texte de PLIB, comme par exemple `preferred_name`, `definition` ou `remark`, peuvent alors être définies dans ces différentes langues naturelles. En PLIB, le multilinguisme est également supporté au niveau des données. Ce modèle permet en effet de définir des types énumérés (`non_quantitative_code_type`) dont les valeurs peuvent être données dans plusieurs langues naturelles.

3.3.3 Relations entre les mots

Les ontologies linguistiques utilisent des relations entre les termes et notamment les suivantes :

- *la synonymie* : deux termes ont une signification identique ;
- *l'homonymie* : deux termes ont la même forme orale ou écrite mais des sens différents ;
- *l'antonymie* : deux termes ont un sens opposé ;
- *la similarité* : deux termes ont un sens proche ;
- *l'hyperonymie* (respectivement *l'hyponymie*) : un terme à son extension qui englobe (respectivement est inclus dans) l'extension d'un autre terme.

Les modèles d'ontologies présentés précédemment ne supportent pas la plupart de ces relations. A notre connaissance, seul le modèle PLIB supporte la relation de synonymie (`synonymous_names`). Ce constructeur permet d'associer des noms synonymes à une classe ou à une propriété. Notons également que le constructeur de OWL `owl:equivalentClass` peut être utilisé pour représenter la synonymie lorsque les classes représentent un mot d'une langue naturelle. De même, si plusieurs noms sont associés à une classe RDF-Schema, ces noms peuvent être considérés comme des synonymes.

3.4 Noyau commun aux modèles d'ontologies

Cette section présente une vue synthétique des modèles d'ontologies étudiés en mettant en évidence le noyau commun à ces différents modèles et les extensions propres à chacun.

Le tableau 1.1 liste les différents modèles d'ontologies que nous avons étudiés en indiquant les constructeurs qu'ils fournissent. Dans ce tableau, nous utilisons le symbole • pour indiquer qu'un construc-

teur est fourni par un modèle d'ontologies. Lorsqu'un constructeur est partiellement supporté, nous utilisons le symbole ○. C'est par exemple, le cas de l'héritage multiple qui n'est pas explicitement supporté en PLIB mais qui peut être représenté via l'opérateur `case_of`. Enfin, si le constructeur est absent et ne peut pas être représenté via d'autres constructeurs, le symbole - est utilisé. D'autres part, les deux traits verticaux permettent de séparer les différentes couches d'une ontologie (OCC, OCNC et OL).

Constructeurs	**RDF-Schema**	**PLIB**	**OWL**	**F-Logic**
OCC				
Ontologie/namespace	●	●	●	○
Classe	●	●	●	●
Héritage multiple de classe	●	○	●	●
Propriété	●	●	●	●
Propriété dépendante du contexte	-	●	-	○
Subsomption de propriété	●	-	●	-
Types de données				
_primitifs	●	●	●	●
_classe	●	●	●	●
_collections	●	●	●	●
_unité mesure/monnaie	-	●	-	-
Instance				
_identification universelle	●	-	●	-
_multi-instanciation	●	○	●	●
_meta-modélisation	●	-	Full	●
OCNC				
Opérateurs booléens sur les classes	-	-	●	-
Restrictions	-	-	●	-
Classes définies par règles	-	-	-	●
Propriétés définies par règles	-	-	-	●
OL				
Description textuelle	●	●	●	○
Multilinguisme	●	●	●	-
Synonymie	○	●	○	-

Tab. 1.1 – Synthèse des constructeurs offerts par les modèles d'ontologies

Ce tableau montre que même si les modèles d'ontologies ont été conçus par différentes communautés pour différents contextes d'utilisation, il existe un ensemble de constructeurs communs à ces modèles permettant de construire des OCC. Ce noyau commun est composé de constructeurs d'ontologies, de classes, de propriétés, de types de données et d'instances. Ces constructeurs sont décrits dans les sections suivantes.

3.4.1 Le constructeur d'ontologies

Le constructeur d'ontologies permet de regrouper la définition d'un ensemble de concepts qui sont des classes ou des propriétés. Une ontologie définit un domaine d'unicité des noms aussi appelé *espace de noms* permettant d'identifier les concepts qu'elle définit de manière unique. Elle est souvent décrite par des informations sur le fournisseur de cette ontologie. Certains modèles d'ontologies tels que OWL et PLIB fournissent également des descripteurs permettant de gérer les versions des ontologies conçues et de les décomposer en modules.

3.4.2 Le constructeur de classes

Le constructeur de classes est un mécanisme d'abstraction permettant de regrouper un ensemble d'instances présentant des caractéristiques communes. Dans un univers particulier, une classe est associée à un ensemble d'instances appelé son *extension*. Une classe a une *définition en intention* qui décrit le concept sous-jacent. Cette définition est généralement composée des éléments suivants :

– un identifiant. Il permet de référencer cette classe. Cet identifiant est universel et unique. Il est parfois complété par un numéro de version afin de gérer l'évolution des concepts d'une ontologie ;
– une description textuelle. Elle permet de rattacher une classe à une connaissance pré-existante de l'utilisateur. Une ontologie étant une conceptualisation acceptée par une vaste communauté d'utilisateurs, elles sont souvent utilisées dans un contexte international comme par exemple le Web. En conséquence, les définitions textuelles associées aux concepts qu'elle définit sont souvent définies dans plusieurs langues naturelles ;
– les classes qu'elle généralise et spécialise. Les classes sont organisées dans une hiérarchie où elles sont liées par une relation de subsomption. La plupart des modèles supportent l'héritage multiple ou permettent de le représenter.

Les éléments précédents permettent de rattacher une classe à un savoir préexistant partagé par le lecteur. Ils définissent également des conditions nécessaires d'appartenance d'une instance à une classe. Ils permettent ainsi de définir une *classe primitive* (concepts primitifs).

3.4.3 Le constructeur de propriétés

Le constructeur de propriétés permet de décrire les instances d'une classe. Les propriétés, comme les classes, possèdent toujours un identifiant et une partie textuelle éventuellement définie dans plusieurs langues naturelles. Chacune des propriétés doit être définie sur une classe des instances qu'elle décrit. Cette classe est le *domaine de définition* de la propriété. Dans certains formalismes, comme par exemple RDF-Schema, ce domaine peut être l'intersection de plusieurs classes. Il peut également être facultatif. Dans ce cas, le domaine de la propriété est constitué de l'ensemble des objets appartenant au domaine visé par l'ontologie. Notons que ceci peut être représenté en définissant cette classe implicite. Une propriété a également un *codomaine* qui permet de restreindre son domaine de valeurs. Enfin, les modèles d'ontologies permettent de faire la distinction entre les propriétés *monovaluées*, c'est-à-dire qui ne présentent au maximum qu'une valeur pour une instance donnée, des propriétés *multivaluées* qui peuvent présenter plusieurs valeurs pour une même instance.

3.4.4 Le constructeur de types de données

Le constructeur de types de données permet de définir le codomaine des propriétés d'une ontologie. Les modèles d'ontologies permettent la définition de types simples, principalement les types entier, réel, chaîne de caractères, booléen et date. Une classe peut également être utilisée comme type de données. Dans ce cas, la valeur d'une telle propriété est l'identifiant d'une instance de la classe formant son codomaine. Enfin, les modèles d'ontologies permettent la définition de type collection dont les éléments sont soit d'un type simple, soit des identifiants d'instances de classes.

3.4.5 Le constructeur d'instances

Le constructeur d'instances permet de définir l'extension d'une classe dans un certain univers. A l'inverse de ce qui se passe pour les classes, aucun modèle d'ontologies ne fait l'hypothèse d'existence d'un identifiant unique et universel pour les instances. En PLIB et F-Logic, les instances ne sont identifiées que localement, par rapport au système où elles sont définies. En RDF-Schema et OWL, l'identifiant est un URI, qui peut être utilisée pour référencer cette instance en dehors de ce système. Cependant, une instance n'a pas forcément qu'un seul identifiant. Ainsi, un même objet du monde réel pourra être identifié différemment dans différents systèmes.

Une instance est définie en indiquant ses classes d'appartenance. Parmi ces classes, le modèle PLIB distingue une *classe de base*, c'est-à-dire la plus basse classe dans la hiérarchie de subsomption à laquelle une instance appartient. Les modèles d'ontologies permettent de définir plusieurs classes de base pour une instance. Deux mécanismes sont utilisés à ces fins : la *multi-instanciation* et l'*agrégat d'instances*. A la différence de la multi-instanciation qui n'impose aucune restriction par défaut[9], l'agrégat requiert que les propriétés définies sur les classes de base d'une instance soient disjointes. Cette restriction permet de ne pas dupliquer la valeur d'une instance pour une propriété.

Une instance est également caractérisée par un ensemble de valeurs de propriétés. Dans les bases de données cette caractérisation est faite sous l'*hypothèse de typage exact* qui requiert que chaque instance propose une valeur, qui peut être la valeur `NULL`, pour chaque propriété définie dans le modèle conceptuel. Dans une ontologie cette caractérisation est plus souple, elle respecte l'hypothèse que nous nommons *hypothèse de typage ontologique* qui permet à une instance de n'être caractérisée que par un sous-ensemble des propriétés définies sur ses classes de base.

3.5 Synthèse sur les modèles d'ontologies

Notre analyse des modèles d'ontologies nous a amené au constat qu'il existe un noyau commun à la plupart des modèles d'ontologies proposés dans la littérature. En conséquence, nous considérons désormais que les ontologies de domaine présentent toutes la base commune suivante.

> Une ontologie de domaine définit un espace de noms dans lequel des éléments sont créés sous la forme de classes et de propriétés. Elle est constituée d'un ensemble de classes primitives, liées par des relations de subsomption et associées à des propriétés dont le codomaine peut être une classe, un type simple ou collection. Les classes et les propriétés peuvent être

[9] L'axiome `owl:disjointWith` peut être utilisé pour imposer des restrictions sur la multi-instanciation.

référencées grâce à des identifiants. Elles sont définies à la fois de façon textuelle et par les relations formelles les unissant aux autres concepts de l'ontologie. Des instances peuvent être définies. Elles sont caractérisées par leur appartenance à des classes et par des valeurs de propriétés qui admettent dans leur domaine la (ou l'une des) classe à laquelle appartient l'instance.

En plus de ce noyau commun, les différents modèles proposent un ensemble de constructeurs permettant soit de définir plus précisément et de façon contextuelle les concepts primitifs d'une ontologie (PLIB) soit d'introduire des constructeurs d'équivalences conceptuelles permettant ainsi de construire une OCNC (OWL, F-Logic). Ces différents constructeurs sont complémentaires car ils permettent de construire des ontologies selon le modèle en oignon. Ceci permet, comme nous l'avons montré à la section 2.5, de répondre à une large gamme de problèmes.

4 Les Bases de Données à Base Ontologique (BDBO)

Les ontologies étant utilisées dans des domaines de plus en plus nombreux pour gérer des données volumineuses (bases de données techniques, portail Web), la quantité des données définies par référence à des ontologies a considérablement augmenté. En conséquence, la gestion de ces données en mémoire centrale ne satisfait pas les besoins de performances et de fiabilité requis pour de nombreuses applications. Pour résoudre ce problème, des propositions ont été faites pour permettre la persistance d'ontologies et de leurs instances dans une base de données. Dans cette section, nous analysons les architectures proposées pour ces bases de données : les bases de données à base ontologiques.

4.1 Définition du concept de BDBO

Nous appelons *base de données à base ontologique* (BDBO) [Dehainsala, 2007, Pierra et al., 2005] une source de données qui contient des ontologies, un ensemble de données et des liens entre ces données et les éléments ontologiques qui en définissent le sens. Nous nommons *données à base ontologique* les données contenues dans une BDBO. De nombreuses BDBO ont été proposées dans la littérature incluant Sesame [Broekstra et al., 2002], RDFSuite [Alexaki et al., 2001], Jena [B.McBride, 2001, Wilkinson et al., 2003, Carroll et al., 2004], OntoDB [Pierra et al., 2005, Dehainsala et al., 2007a], OntoMS [Park et al., 2007], DLDB [Pan and Heflin, 2003], RStar [L.Ma et al., 2004], KAON [Bozsak et al., 2002], 3Store [Harris and Gibbins, 2003] et PARKA [Stoffel et al., 1997]. Ces BDBO se différencient suivant :

- le modèle d'ontologies supporté ;
- le schéma de base de données utilisé pour stocker des ontologies représentées selon ce modèle ;
- le schéma de base de données utilisé pour représenter les données à base ontologique (instances) ;
- les mécanismes utilisés pour définir le lien entre les données (instances) et les éléments des ontologies (concepts).

Dans la section suivante, nous montrons les différentes approches suivies par les BDBO pour la représentation des ontologies.

4.2 Représentation des ontologies

Deux approches principales ont été suivies pour représenter les ontologies dans une BDBO, l'une dépendant du modèle d'ontologies supporté et l'autre non.

4.2.1 Représentation générique

La première représentation est indépendante du modèle d'ontologies supporté dans la BDBO. Elle est ainsi qualifiée de *générique*. Cette représentation consiste en une unique table à trois colonnes `(sujet, prédicat, objet)`. Cette représentation consiste donc à décomposer en triplets l'ensemble des éléments des ontologies comme par exemple les classes et les propriétés. Chaque élément `elt` d'une ontologie est ainsi défini par les triplets suivants :

- le triplet `(elt, rdf:type, entité)` qui permet d'indiquer le type de l'élément. Ce type est une des entités définies en RDF-Schema comme par exemple `rdfs:Class` ou `rdfs:Property` ;
- les triplets `(elt, attribut, valeur)` qui permettent de caractériser l'élément défini en lui assignant des valeurs d'attributs RDF-Schema comme par exemple `rdfs:label` ou `rdfs:comment`.

Cette représentation est en particulier adoptée par les BDBO Jena et 3Store.

Exemple. La figure 1.7 illustre cette représentation sur un extrait de l'ontologie SIOC.

Triplet		
sujet	**prédicat**	**objet**
http://rdfs.org/sioc/ns#Forum	rdf:type	rdfs:Class
http://rdfs.org/sioc/ns#Forum	rdfs:label	Forum
http://rdfs.org/sioc/ns#Forum	rdfs:comment	A discussion area …
http://rdfs.org/sioc/ns#Forum	rdfs:subClassOf	http://rdfs.org/sioc/ns#Container
http://rdfs.org/sioc/ns#has_host	rdf:type	rdf:Property
http://rdfs.org/sioc/ns#has_host	rdfs:label	has host
http://rdfs.org/sioc/ns#has_host	rdfs:comment	The Site that hosts this Forum
http://rdfs.org/sioc/ns#has_host	rdfs:domain	http://rdfs.org/sioc/ns#Forum
http://rdfs.org/sioc/ns#has_host	rdfs:range	http://rdfs.org/sioc/ns#Site
…	…	…

Fig. 1.7 – Extrait de la représentation générique de l'ontologie SIOC

Explication. Cet exemple présente la représentation générique de la classe `Forum` et de sa propriété `has_host`. Le premier triplet permet d'indiquer que la ressource dont l'URI est `http://rdfs.org/-sioc/ns#Forum` est une classe. Les trois triplets suivants permettent de caractériser cette classe par un nom, un commentaire et sa super-classe. La représentation de la propriété `has_host` est réalisée par les triplets suivants. Le premier triplet indique que `has_host` est une propriété et les suivants la caractérisent par un nom, un commentaire, son domaine et son codomaine.

4.2.2 Représentation spécifique

La seconde représentation est dépendante du modèle d'ontologies supporté par la BDBO. Elle est ainsi qualifiée de *spécifique*. Elle consiste en une représentation du modèle d'ontologies dans le modèle relationnel ou relationnel-objet supporté par le SGBD sous-jacent à la BDBO. Actuellement, il n'y a pas de consensus sur ce qu'est la représentation optimale pour un modèle d'ontologies donné. Chaque BDBO définit son propre schéma en fonction des constructeurs du modèle d'ontologies qu'elle supporte. Cette représentation est adoptée par les BDBO Sesame, RDFSuite, RSTAR, OntoDB, OntoMS, DLDB, PARKA et KAON.

Exemple. La figure 1.8 présente un extrait d'une représentation spécifique possible pour l'ontologie SIOC décrite en RDF-Schema.

Class			
id	**URI**	**label**	**comment**
1	http://rdfs.org/sioc/ns#Container	Container	An area in ...
2	http://rdfs.org/sioc/ns#Forum	Forum	A discussion ...
3	http://rdfs.org/sioc/ns#Site	Site	The location of ...

SubClassOf	
sub	**sup**
2	1

Property					
id	**URI**	**label**	**comment**	**domain**	**range**
11	http://rdfs.org/sioc/ns#has_host	has host	The Site ...	2	3

FIG. 1.8 – Extrait de la représentation spécifique de l'ontologie SIOC

Explication. La représentation spécifique présentée comporte les tables suivantes :

- `Class` qui permet de stocker les classes des ontologies. Cette table est composée d'une colonne `URI` permettant de stocker l'identifiant externe d'une classe. Afin d'optimiser les traitements, un identifiant interne à la base de données (`id`) est également associé aux classes. Cette table permet également de stocker les noms (`label`) et commentaires (`comment`) associés aux classes. Notons que si nous souhaitons associer plusieurs noms et/ou plusieurs commentaires à une même classe, la normalisation de cette table nécessite de définir deux nouvelles tables pour stocker ces informations ;
- `SubClassOf` qui permet de stocker la hiérarchie des classes en indiquant pour chaque classe ses super-classes ;
- `Property` qui permet de stocker les propriétés. Comme les classes, les propriétés sont associées à un identifiant interne (`id`) et externe (`URI`) et à des noms (`label`) et des commentaires (`comment`). Le domaine (`domain`) et codomaine (`range`) des propriété sont également spécifiés dans cette table par une référence à des classes. Notons que, même si ce n'est pas montré dans cet exemple, le codomaine peut également référencer un type de données (entier, chaîne de caractères, etc.). A nouveau, si nous souhaitons permettre que ce domaine ou codomaine soit défini par une liste de classes, de nouvelles tables sont nécessaires pour chacune de ces informations.

4.2.3 Tendance actuelle de représentation des ontologies

La tendance actuelles des BDBO est de proposer une représentation spécifique dont les évaluations de performances par rapport à un ensemble de requêtes typiques se sont montrées meilleures que la représentation générique [Theoharis et al., 2005, Alexaki et al., 2001, L.Ma et al., 2004]. Ainsi les BDBO Sesame, RDFSuite et RStar présentent une telle représentation pour le modèle RDF-Schema ; OntoMS, DLDB et PARKA pour OWL ; KAON pour le modèle du même nom et OntoDB pour le modèle PLIB.

Ces BDBO sont ainsi conçues spécifiquement pour un modèle d'ontologies particulier. Cependant, compte tenu de la diversité des modèles d'ontologies utilisés, la plupart des BDBO propose des mécanismes permettant leur utilisation avec d'autres modèles d'ontologies. Ainsi, de nombreuses BDBO fournissent la possibilité d'importer et d'exporter leurs données vers d'autres modèles d'ontologies. Par exemple, Sesame permet d'exporter ces données en OWL, KAON en RDF et OntoDB en RDF-Schema et OWL Lite. D'autre part, la BDBO OntoDB propose une structure pour stocker et faire évoluer le modèle d'ontologies utilisé (PLIB). La représentation de cette partie, nommée *méta-schéma*, permet d'adapter ce système aux évolutions du modèle d'ontologies PLIB, mais aussi, lorsque le besoin s'en fait sentir, d'étendre le modèle d'ontologies utilisé par des constructeurs provenant d'autres modèles. Ceci montre la nécessité de pouvoir manipuler les données d'une BDBO quel que soit le modèle d'ontologies supporté.

Nous venons de voir que la tendance des BDBO est d'utiliser un schéma de bases de données traditionnel pour stocker les ontologies. Dans la section suivante, nous abordons les propositions faites pour stocker les données.

4.3 Représentation des données à base ontologique (instances)

Comme pour la représentation des ontologies, deux approches principales ont été suivies pour représenter les données, l'une dépendant des ontologies stockées dans une BDBO et l'autre non.

4.3.1 Représentation générique

Deux propositions de représentation, indépendantes des ontologies stockées, ont été faites. La première, qualifiée de *verticale*, consiste en une table à trois colonnes `(sujet, prédicat, objet)` où l'ensemble des informations de niveau données (instances des classes et valeurs de propriétés) est décomposé en triplets. Dans cette représentation, chaque instance `i` d'une classe est définie par les triplets suivants :

- les triplets `(i, rdf:type, C)` qui permettent d'indiquer les classes auxquelles l'instance `i` appartient ;
- les triplets `(i, propriété, valeur)` qui permettent de caractériser l'instance en lui assignant des valeurs de propriétés.

L'approche verticale est en particulier suivie par les BDBO Jena, 3Store et RStar.

Exemple. La figure 1.9 présente une représentation verticale d'instances de la classe `Post`.

Explication. Le premier triplet indique que la ressource dont l'URI se termine par `post-sioc` est une

Triplet		
sujet	**prédicat**	**objet**
http://lisi.../post-sioc	rdf:type	http://rdfs.org/sioc/ns#Post
http://lisi.../post-sioc	http://rdfs.org/sioc/ns#title	Creating connections …
http://lisi.../post-sioc	http://rdfs.org/sioc/ns#has_container	http://lisi.../weblog
http://lisi.../post-sioc	http://rdfs.org/sioc/ns#has_creator	http://lisi.../author/cloud
http://lisi.../post-sioc	http://rdfs.org/sioc/ns#content	SIOC provides a unified …
http://lisi.../post-sioc	http://rdfs.org/sioc/ns#has_reply	http://lisi.../comment-928
…	…	…

FIG. 1.9 – Représentation verticale des données décrites par l'ontologie SIOC

instance de la classe `Post`. Les triplets suivants décrivent cette instance par des valeurs pour les propriétés `title`, `has_container`, `has_creator`, `content` et `has_reply`.

La seconde approche qualifiée de *binaire* consiste à utiliser une table unaire par classe pour représenter les instances de cette classe et une table binaire par propriété pour représenter les valeurs de propriétés des instances. Ainsi, les deux types de triplets identifiés dans la représentation verticale sont séparés. Les triplets indiquant le type des instances sont représentés dans les tables unaires qui correspondent aux classes des ontologies et les triplets indiquant la valeur des propriétés sont représentées dans les tables binaires qui correspondent aux propriétés des ontologies. L'approche binaire est suivie par Sesame, RDF-Suite, DLDB et PARKA.

Exemple. La figure 1.10 présente une représentation binaire d'instances de la classe `Post`.

Post
URI
http://lisi.../post-sioc

Title	
sujet	**objet**
http://lisi.../post-sioc	Creating connections …

Has_container	
sujet	**objet**
http://lisi.../post-sioc	http://lisi.../weblog

Has_creator	
sujet	**objet**
http://lisi.../post-sioc	http://lisi.../author/cloud

FIG. 1.10 – Représentation binaire des données décrites par l'ontologie SIOC

Explication. La représentation binaire comporte 4 tables. La table `Post` permet de stocker les instances de la classe du même nom. L'appartenance de la ressource dont l'URI se termine par `post-sioc` à cette table permet ainsi d'indiquer que cette ressource appartient à la classe `Post`. Les autres tables permettent de stocker les valeurs des propriétés `title`, `has_container` et `has_creator`. Par exemple, la présence du tuple `(post-sioc, weblog)` dans la table `title` indique que cette instance à la valeur `weblog` pour la propriété `title`.

Que ce soit dans la représentation verticale ou binaire, une instance donnée ayant un certain nombre de propriétés est décomposée en plusieurs tuples dans différentes tables (approche binaire) ou dans la table de triplets (approche verticale). Ces instances sont ainsi directement liées à une ontologie sans être structurées par un schéma significatif pour un utilisateur de la base de données. La notion de schéma de données au sens usuel des bases de données classiques n'est donc pas présente. Ces choix de stockage qui reviennent à décomposer chaque instance en la séparant de ses valeurs de propriétés sont dictés par

la souplesse de représentation des données à base ontologique que permettent les langages de définition d'ontologies RDF-Schema et OWL. Les instances dans ces langages ne sont pas fortement typées. Elles peuvent appartenir à un nombre quelconque de classes non liées par la relation de subsomption et peuvent même initialiser des propriétés non définies dans le contexte de leurs classes.

Néanmoins, dans bien des cas, les instances d'une ontologie RDF-Schema ou OWL respectent des hypothèses de typage : elles appartiennent à une seule classe de base et n'ont une valeur que pour les propriétés définies sur cette classe. C'est par exemple, le cas des applications qui se basent sur les données et les schémas d'une base de données pour construire des ontologies OWL et leurs instances [de Laborda and Conrad, 2006]. Dans ce cas, la représentation des données à base ontologique peut se rapprocher de la structure relationnelle. C'est ce que proposent les approches de représentation présentées dans la section suivante.

4.3.2 Représentation spécifique

Les BDBO OntoMS et OntoDB proposent une représentation des données à base ontologique dépendant de l'ontologie qui décrit ces données. Cette représentation consiste à stocker les instances d'une classe ainsi que ses valeurs de propriétés dans une table relationnelle. Elle est ainsi qualifiée de *représentation horizontale*. La table associée à une classe possède une colonne pour chaque propriété utilisée pour décrire au moins une instance de cette classe. Lorsque qu'une propriété est multivaluée, ses valeurs peuvent être représentées soit en utilisant les types tableaux introduits dans SQL99 (approche suivie par OntoDB), soit en utilisant de nouvelles tables pour ces propriétés (approche suivie par OntoMS).

Exemple. La figure 1.11 présente une représentation horizontale d'instances de la classe `Post`.

Post					
URI	**title**	**has_container**	**has_creator**	**content**	**has_reply**
post-sioc	Creating connections …	weblog	cloud	Sioc …	comment-928

Fig. 1.11 – Représentation horizontale des données décrites par l'ontologie SIOC

Explication. La table `Post` permet de stocker l'ensemble des instances appartenant à la classe `Post`. Dans cet exemple, nous supposons que seules les propriétés `URI`, `title`, `has_container`, `has_creator`, `content` et `has_reply` sont utilisées pour décrire les instances de cette classe. La table contient donc une colonne pour chacune de ces propriétés.

Les évaluations de performances menées sur OntoMS [Park et al., 2007] et OntoDB [Dehainsala et al., 2007a] ont montré que cette représentation permet d'obtenir les meilleurs temps de réponse pour la plupart des requêtes typiques sur les données. Les hypothèses permettant ce gain de performance sont explicitées dans [Dehainsala et al., 2007a]. Les données à base ontologique respectent des *hypothèses de typage ontologique fort* : (1) chaque instance appartient à exactement une classe, appelée sa classe de base, qui correspond à la classe minimum pour la relation d'ordre définie par la relation de subsomption (2) chaque propriété est définie dans le contexte d'une classe qui définit son contexte d'application (3) seules les propriétés applicables dans le contexte d'une classe peuvent être utilisées pour décrire les instances de cette classe.

Outre, le gain de performance obtenu, l'approche horizontale présente l'avantage de conserver la notion de schéma des données. En effet, dans cette représentation, les tables utilisées représentent explicitement la structure des données. Cet aspect a été utilisé dans OntoDB pour permettre l'intégration automatique de bases de données en utilisant cette BDBO [Nguyen-Xuan, 2006]. L'exploitation de cette architecture pour résoudre d'autres problèmes de bases de données tels que la conception de modèles conceptuels ou l'indexation sémantique de bases de données est actuellement en cours.

4.4 Synthèse sur les bases de données à base ontologique

Les BDBO répondent aux besoins de gérer une quantité volumineuse de données décrites par référence à des ontologies (données à base ontologique). Elles permettent de stocker, dans une base de données, des ontologies et les instances qu'elles décrivent. Les instances peuvent être structurées par un schéma comme dans une base de données traditionnelle. Les éléments de ce schéma sont alors liés à une ontologie pour en définir la sémantique. Le schéma des instances peut être construit à partir de l'ontologie. Il peut également avoir été conçu indépendamment d'une ontologie (indexation sémantique). Dans les BDBO utilisant une représentation générique (binaire ou verticale) pour les instances, celles-ci sont directement liées à une ontologie sans être structurées par un schéma significatif pour un utilisateur de la base de données. Néanmoins, dans bien des cas, le schéma des données, implicite dans ces représentations, pourrait être explicité. La tendance actuelle des BDBO est de gérer des données de plus en plus volumineuses, de séparer ontologies et données, et de permettre d'importer/exporter ontologies et données représentées avec différents modèles d'ontologies.

5 Conclusion : application du concept d'ontologie aux bases de données

Ayant analysé le concept d'ontologie, les modèles d'ontologies et les BDBO dans le contexte des bases de données, nous allons maintenant conclure ce chapitre en proposant un apport des ontologies aux bases de données.

Un des buts essentiels d'une base de données est d'une part d'assurer une gestion efficace des données et d'autre part de permettre l'accès aux données indépendamment de leur représentation physique. L'architecture ANSI/SPARC [ANSI/X3/SPARC, 1975] a été proposée pour remplir ces objectifs. Elle distingue les deux niveaux d'accès suivants :

– *le niveau physique*. Il définit comment les données sont stockées et gérées en utilisant le système de gestion de fichiers ;
– *le niveau logique*. Il définit comment les données sont structurées en utilisant le modèle de données de la base de données (par exemple, le modèle relationnel ou objet).

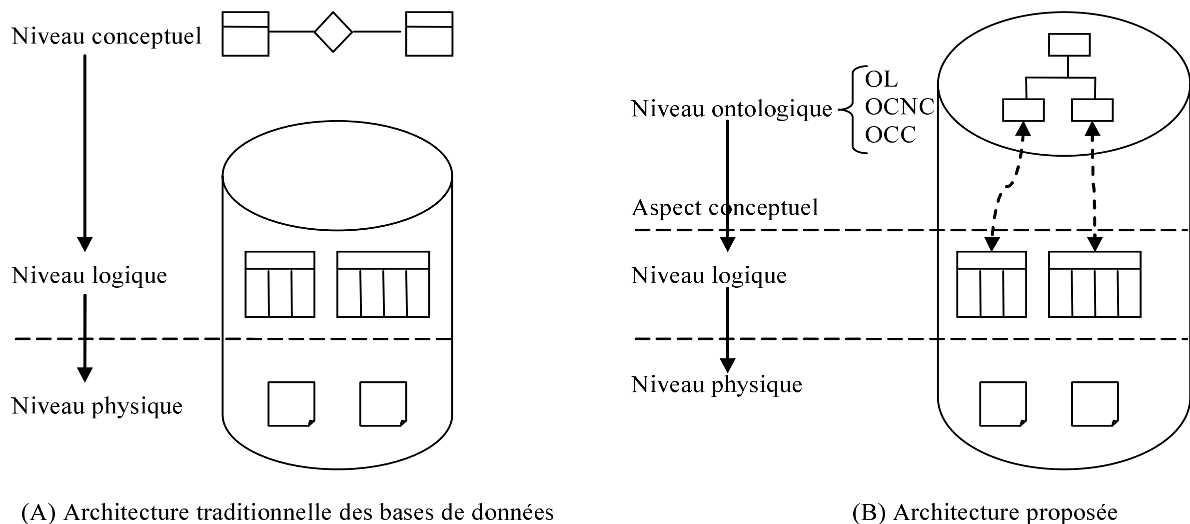
La conception d'une base de données suivant cette architecture passe par la transformation d'un modèle conceptuel en un modèle logique. Cette transformation s'accompagne d'une perte de sémantique des données ce qui pose des problèmes lorsque, par exemple, il est nécessaire d'échanger des données entre deux bases de données, ou de générer une interface d'accès aux données pour un utilisateur final.

En tant que modèle permettant d'exprimer la sémantique des données, les ontologies semblent une solution pertinente à ces problèmes. Dans ce chapitre, l'analyse que nous avons effectuée des multiples

notions d'ontologie que l'on rencontre actuellement dans la littérature nous ont conduit aux résultats suivants :

- la description sémantique des données par une ontologie peut se faire selon trois couches (canoniques, non canoniques et linguistiques) que l'on peut lier selon le modèle en oignon (cf. section 2) ;
- les modèles d'ontologies sont complémentaires pour la conception d'ontologies et possèdent tous un noyau commun (cf. section 3) ;
- les BDBO permettent de gérer un volume important de données à base ontologique. La tendance actuelle des BDBO est de séparer ontologie et données. Le faible typage proposé par les modèles d'ontologies tels que OWL conduit à des structures de BDBO où le schéma des données, au sens traditionnel des bases de données, n'est pas représenté. En pratique, dans beaucoup de cas, les données sont néanmoins fortement typées. Ceci est même obligatoire avec les ontologies PLIB et F-Logic. Il est donc raisonnable de s'intéresser aux BDBO qui permettent la représentation du schéma des données pour chaque classe d'une ontologie. Ceci permet d'aborder différents problèmes de bases de données tels que la conception de bases de données, l'indexation sémantique de bases de données ou l'intégration de bases de données à l'aide de ces structures (cf. section 4).

Ces trois résultats nous ont conduits à généraliser les architectures de BDBO existantes en proposant l'extension de l'architecture ANSI/SPARC avec le *niveau ontologique*. Ce niveau définit la sémantique des données. Il est constitué des descriptions sémantiques fournies par une ontologie. Il peut être décomposé selon les trois couches du modèle en oignon. Il n'est pas lié à un modèle d'ontologies particulier. Cette architecture étendue est présentée sur la figure 1.12.



(A) Architecture traditionnelle des bases de données

(B) Architecture proposée

Fig. 1.12 – Notre proposition d'architecture de bases de données

L'architecture traditionnelle de bases de données est présentée dans la partie (A) de cette figure, située à gauche. Un modèle conceptuel représenté dans un formalisme tel que le modèle Entité/Relation est conçu. Il est ensuite souvent utilisé pour générer automatiquement le modèle logique des données constitué d'un ensemble de tables dans les SGBD relationnels ou relationnels-objets. Ce modèle logique est lui même représenté au niveau physique à l'aide d'un ensemble de fichiers.

Dans la partie (B) nous proposons l'extension de cette architecture en intégrant les deux éléments suivants :

- *le niveau ontologique*. Il est composé d'ontologies qui définissent les concepts de différents domaines d'étude sous la forme de classes et de propriétés. Ces ontologies sont indépendantes des besoins des applications pour lesquelles la base de données est conçue. Cependant, ces ontologies peuvent être spécialisées pour représenter les éventuels concepts manquants par rapport à ces besoins. Lorsqu'elles sont conçues selon le modèle en oignon, ces ontologies comportent toujours une couche canonique. Elles peuvent éventuellement comporter une couche non canonique. Elles comportent toujours un minimum d'aspects linguistiques et, en particulier, des termes qui dénotent les concepts représentés ;
- *l'aspect conceptuel*. Cet aspect est représenté par le lien entre le niveau ontologique et le niveau logique. Ce lien indique l'ensemble des concepts des ontologies qui sont exploités pour satisfaire les besoins des applications pour lesquelles la base de données est conçue. Cet ensemble de concepts, une fois choisi, peut être utilisé pour générer automatiquement le modèle logique des données.

Le problème abordé dans cette thèse est de fournir un langage de définition, manipulation et interrogation de données pour l'architecture ANSI/SPARC étendue que nous proposons. Les trois niveaux de cette architecture (physique, logique et ontologique) sont actuellement implantés dans différentes approches avec des techniques différentes comme l'a montré notre étude des BDBO. Cependant, peu de travaux se sont intéressés à la définition d'un langage d'exploitation qui (1) soit *homogène*, c'est-à-dire qu'il permette l'accès aux différents niveaux significatifs pour un utilisateur de cette architecture et (2) tire profit des spécificités de ces différents niveaux. En effet, outre le langage SQL qui a été défini pour permettre d'exploiter les niveaux physique et logique de l'architecture ANSI/SPARC, d'autres langages comme SPARQL [Prud'hommeaux and Seaborne, 2006] ou RQL [Karvounarakis et al., 2002] ont été définis. Mais, ces nouveaux langages n'exploitent pas les caractéristiques particulières des différents niveaux définis par cette extension de l'architecture ANSI/SPARC. Nous montrons ceci dans le chapitre suivant en identifiant les besoins d'exploitation induits par l'architecture de bases de données proposée.

Chapitre 2

Exigences pour un langage d'exploitation de BDBO

Sommaire

1 Introduction **53**

2 Exigences liées à l'architecture de BDBO proposée **53**

2.1 Exigences liées au modèle en oignon 54

2.2 Exigences liées à la compatibilité avec l'architecture traditionnelle des bases de données 56

2.3 Exigences sur le pouvoir d'expression du langage 57

2.4 Exigences sur l'implantation du langage 59

3 Analyse des langages de BDBO conçus pour le Web Sémantique **60**

3.1 Les langages conçus pour le formalisme RDF 60

3.2 Les langages conçus pour le modèle d'ontologies RDF-Schema 66

4 Analyse des autres langages de BDBO **77**

4.1 Le langage CQL associé au modèle d'ontologies PLIB 77

4.2 Le langage SOQA-QL indépendant d'un modèle d'ontologies particulier . 82

5 Conclusion **86**

Résumé. Dans le chapitre précédent, nous avons mené une analyse du domaine des ontologies qui nous a conduit à proposer une extension de l'architecture ANSI/SPARC pour les bases de données. Dans ce chapitre, nous étudions les conséquences de cette extension sur le langage de définition, de manipulation et d'interrogation de données qui constitue l'outil principal d'exploitation d'une base de données respectant cette architecture étendue. Dans un premier temps, nous présentons notre analyse de l'architecture proposée en termes d'exigences pour un tel langage [Jean et al., 2005]. Ces exigences sont basées sur les principes fondamentaux introduits par l'architecture ANSI/SPARC. Dans un second temps, nous évaluons les langages existants par rapport à ces exigences. Nous montrons en quoi ces langages ne satisfont pas complètement ces exigences et identifions les approches qui permettent d'en satisfaire certaines. Cette analyse montre la nécessité d'un nouveau langage de BDBO et identifie les approches suivies par les langages proposés dans la littérature qui sont pertinentes pour la conception d'un tel langage.

1 Introduction

L'architecture ANSI/SPARC définie pour les SGBD a fortement influencé la conception des langages d'exploitation qui leur ont été associés. La définition de cette architecture est en effet associée à un ensemble de principes tels que l'indépendance des différents niveaux d'un SGBD que doit respecter un langage d'exploitation de tels systèmes.

Dans le chapitre précédent nous avons expliqué en quoi les ontologies constituent pour nous un nouveau modèle en informatique avec de nouvelles capacités. Nous avons mené cette analyse dans une perspective d'exploitation de bases de données afin de déterminer les apports possibles des ontologies à l'architecture actuelle des bases de données. Le résultat de ce travail est une extension de l'architecture ANSI/SPARC pour les BDBO. Le premier objectif de ce chapitre est de présenter les exigences que nous avons établies pour un langage de BDBO résultant de cette extension [Jean et al., 2005]. Pour réduire le périmètre de ce travail, nous n'avons pas considéré les problèmes liés à la gestion de la sécurité, des autorisations et des transactions dans l'architecture proposée. Nous nous limitons aux aspects fonctionnels, c'est-à-dire aux traitements des données contenues dans une BDBO. Pour déterminer les exigences induites par les traitements nécessaires, nous analysons l'architecture proposée en nous basant sur les principes fondamentaux introduits par l'architecture ANSI/SPARC.

Une fois ces exigences définies, il reste à analyser les langages proposés dans la littérature par rapport à ces exigences. Le second objectif de ce chapitre est aussi de montrer les limites des langages existants pour satisfaire les exigences définies. Nous essaierons notamment de déterminer ce qui fait que ces langages ne satisfont pas complètement ces exigences afin de donner des orientations pour la conception d'un nouveau langage les satisfaisant.

Ce chapitre est organisé comme suit. Dans la section suivante, nous discutons des conséquences de l'architecture de BDBO que nous avons proposée en termes d'exigences pour un langage d'exploitation de tels systèmes. Dans la section 3, nous analysons les langages de BDBO proposés dans le contexte du Web Sémantique et dans la section 4 les langages de BDBO conçus dans un autre contexte que le Web Sémantique, notamment l'intégration de sources de données et les bases de données techniques. Enfin, la section 5 conclut ce chapitre en synthétisant les limitations des langages existants par rapport à nos exigences et en présentant les perspectives qu'ouvrent l'analyse de ces langages pour la conception d'un langage satisfaisant ces exigences.

2 Exigences liées à l'architecture de BDBO proposée

Avant de présenter les exigences que nous avons identifiées, il est nécessaire de préciser les hypothèses inhérentes à l'architecture de BDBO proposée.

Cette architecture présente la particularité d'être compatible avec les bases de données traditionnelles. En conséquence, les *hypothèses du monde clos* (les faits non présents dans la BDBO sont considérés comme faux) et d'*unicité des noms* (deux identifiants différents référencent deux éléments différents) sont faites. Dans cette architecture, comme dans toute BDBO, chaque donnée est associée à la notion ontologique qui en définit le sens. Concernant la structure des données à gérer, les modèles d'ontologies supportant la multi-instanciation autorisent chaque instance à appartenir à plusieurs classes non liées par

la relation de subsomption et à présenter une valeur pour les différentes propriétés définies sur ces classes. Ainsi, chaque instance peut avoir sa propre structure indépendamment de celle des autres instances. Sous cette hypothèse, que nous qualifions de *typage ontologique faible*, les instances d'une même classe ne présentent pas forcément la même structure. Comme nous l'avons vu au chapitre 1, section 4, la gestion de telles instances dans les BDBO requiert de les représenter selon un schéma qui diffère de la notion usuelle de schéma en base de données.

En conséquence, nous avons choisi de ne pas supporter la multi-instanciation qui permet à une instance d'appartenir à plusieurs classes de base. Ainsi, nous imposons la contrainte qu'une instance ne doit avoir qu'une et une seule classe de base. Cette hypothèse pourrait en fait être légèrement relaxée en imposant que, si une instance appartient à plusieurs classes de base, celles-ci ne possèdent aucune propriété en commun. Grâce à cette contrainte et au fait que les propriétés sont associées à un domaine, les instances d'une même classe présentent une structure similaire composée de tout ou partie des propriétés définies sur cette classe. En décidant alors de représenter pour chaque classe l'ensemble des propriétés valuées par au moins une instance (en complétant les autres instances par des valeurs `NULL`), chaque instance est représentée par un tuple de même structure. Nous qualifions cette hypothèse de *typage ontologique fort*. Cette hypothèse permet d'abord de conserver la notion usuelle de schéma des données nécessaire pour l'intégration de bases de données ou l'indexation de bases de données existantes. Par ailleurs, cette structure permet une représentation des données beaucoup plus efficace que celles proposées pour la gestion de données utilisant le typage ontologique faible [Dehainsala et al., 2007a]. Notons que la contrainte d'unicité de la classe de base, pour une instance, a déjà été faite pour les mêmes raisons (structure fixée et stockage efficace) dans les Bases de Données Orientées-Objets (BDOO) [Banerjee et al., 1987, Abiteboul and Bonner, 1991].

Ayant précisé les hypothèses inhérentes à l'architecture de BDBO proposée, nous discutons des exigences résultant de cette architecture. Ces exigences sont regroupées dans les quatre catégories suivantes :

- exigences liées au modèle en oignon ;
- exigences liées à la compatibilité avec l'architecture des bases de données traditionnelles ;
- exigences sur le pouvoir d'expression du langage ;
- exigences sur l'implantation du langage.

2.1 Exigences liées au modèle en oignon

La première grande innovation de l'architecture ANSI/SPARC traditionnelle est la distinction entre la représentation interne des données au niveau physique et la représentation logique de celles-ci. Cette distinction permet de définir, manipuler et interroger les données au niveau logique indépendamment de leur implantation physique.

L'extension de cette architecture que nous proposons permet également la distinction entre la représentation logique des données (structure) et la représentation ontologique de celles-ci (sémantique). Cette distinction permet de définir, manipuler et interroger les données au niveau ontologique indépendamment de leur représentation logique. Cette capacité permet ainsi d'interroger différentes BDBO, utilisant la même ontologie mais des schémas différents, avec les mêmes requêtes (*requêtes ontologiques*).

Exigence 1 (*Expression de requêtes au niveau ontologique*)

> *Le langage doit permettre d'exprimer des requêtes sur les données, indépendamment de leur schéma, à partir des ontologies contenues dans une BDBO.*

La seconde grande innovation de l'architecture ANSI/SPARC est la possibilité de créer des schémas externes (vues) dans une base de données. Cette possibilité permet à chaque utilisateur de définir sa propre perception de la base de données à partir du schéma des données et ceci, indépendamment des schémas externes des autres utilisateurs. L'architecture ANSI/SPARC permet ainsi la définition de plusieurs schémas externes représentant différentes vues sur la base de données avec des possibilités de recouvrement. Des requêtes peuvent être exprimées sur ces schémas externes, le SGBD se chargeant d'interpréter ces requêtes en fonction du schéma logique des données.

Dans l'extension que nous proposons, la couche OCNC d'une ontologie offre les mêmes capacités que les vues au niveau ontologique. Elle permet à chaque utilisateur de définir sa propre perception du domaine d'étude en représentant un ensemble de concepts non canoniques à partir des concepts canoniques d'une ontologie. La définition de tels concepts ainsi que la possibilité de les utiliser dans les requêtes est donc une exigence d'un langage d'exploitation pour cette architecture.

Exigence 2 (*Définition de concepts non canoniques*)

> *Le langage doit permettre de définir des concepts non canoniques à partir des concepts canoniques d'une ontologie. Il doit également permettre d'exprimer des requêtes ontologiques à partir de ces concepts non canoniques, se chargeant ainsi d'interpréter ces requêtes en fonction des concepts canoniques associés.*

La dernière couche du modèle en oignon est constituée de la partie OL. Lorsqu'une ontologie est construite selon le modèle en oignon, sa partie OL associe à chacun de ses concepts un nom sous forme de terme, et une définition textuelle. La définition permet notamment à des êtres humains de comprendre l'ontologie et les noms permettent d'y faire référence. Or, le contexte d'utilisation d'une ontologie est souvent international. Par exemple, un des objectifs du Web Sémantique est de favoriser l'échange d'informations contenues sur le Web qui est un outil international. Donc, une exigence essentielle est que le système puisse supporter des OL multilingues et que le langage permette à des personnes provenant de différents pays et donc pouvant parler différentes langues de référencer les éléments d'une ontologie directement par leurs dénominations dans leurs propres langues.

Exigence 3 (*Exploitation linguistique*)

> *Le langage doit permettre d'exploiter les dénominations linguistiques, éventuellement données dans plusieurs langues naturelles, qui peuvent être associées à chaque concept d'une ontologie.*

Le modèle en oignon repose également sur le fait que les modèles d'ontologies sont complémentaires. En effet, comme nous l'avons vu au chapitre 1, section 3, il existe de nombreux modèles d'ontologies proposés dans la littérature. Or, ils présentent chacun des particularités qui peuvent être simultanément nécessaires pour le problème à traiter. Par exemple, dans le contexte du projet e-Wok Hub[10], visant à gérer la mémoire de plusieurs projets d'ingénierie sur la capture et le stockage de CO_2, les experts

[10]http://www-sop.inria.fr/acacia/project/ewok/index.html

doivent développer des ontologies pour couvrir ce domaine. Ils doivent choisir un modèle d'ontologies sachant que, d'une part, il est nécessaire de modéliser les concepts du domaine physique avec précision, c'est-à-dire en définissant leurs dimensions physiques, associées à des unités de mesure et à un contexte d'évaluation. Ce besoin suggère l'utilisation d'un modèle d'ontologies tel que PLIB. D'autre part, l'utilisation de constructeurs introduits par les modèles d'ontologies issus de la logique de description, tels que OWL, apparaissent également nécessaires pour permettre d'améliorer la qualité des recherches documentaires, également essentielles pour le problème visé. Cet exemple montre le besoin d'un langage permettant de manipuler ontologies et données quel que soit le modèle d'ontologies utilisé.

Exigence 4 (*Indépendance par rapport à un modèle d'ontologies donné*)

> *Le langage doit permettre de décrire, de manipuler et d'interroger des BDBO lorsque les ontologies sont définies avec des constructeurs pouvant provenir de différents modèles d'ontologies tels que OWL, PLIB ou F-Logic.*

Nous venons de présenter les exigences résultant de la décomposition du niveau ontologique de notre architecture selon les trois couches du modèle en oignon. Notre architecture présente également la particularité de vouloir être compatible avec l'architecture traditionnelle des bases de données. Ceci induit de nouvelles exigences.

2.2 Exigences liées à la compatibilité avec l'architecture traditionnelle des bases de données

Comme nous l'avons précisé dans la conclusion du chapitre 1, notre but est de définir un langage qui permette l'accès aux différents niveaux significatifs de l'architecture que nous proposons. Notre architecture étant basée sur l'architecture ANSI/SPARC, elle intègre le niveau logique. Le langage doit donc non seulement permettre d'accéder aux données à partir du niveau ontologique (cf. section précédente) mais également au niveau logique. Or, le langage SQL a été défini pour manipuler les données à ce niveau. Afin de bénéficier de la vaste adoption de ce langage, ainsi que des nombreux travaux menés sur l'optimisation de requêtes SQL dans les SGBD, le langage doit être compatible avec le langage SQL.

Exigence 5 (*Compatibilité avec SQL*)

> *Le langage doit permettre d'accéder au niveau logique d'une BDBO en étant compatible avec le langage SQL.*

L'inconvénient de construire une architecture à plusieurs niveaux telle que celle que nous proposons est de complexifier les traitements sur les données au fur et à mesure que l'on monte dans les niveaux. La complexité introduite a généralement un impact sur l'efficacité de ces traitements. Dans de telles architectures, une manière d'optimiser les traitements à un niveau donné, est d'accéder au niveau inférieur. Par exemple, dans l'architecture ANSI/SPARC, pour optimiser certaines requêtes SQL portant sur le niveau logique de cette architecture, on peut utiliser la connaissance que l'on a du niveau physique en utilisant des *hints* proposés par la plupart des SGBD commerciaux. Un hint est une directive pour forcer l'optimiseur de requêtes à choisir un traitement spécifique, pour une requête donnée, afin de l'optimiser. Par exemple, Oracle permet de forcer l'optimiseur de requêtes à utiliser un index donné (syntaxe `/*+ INDEX`

`(nom_table nom_index) */`). Ce SGBD permet également d'accéder au niveau physique d'une base de données dans le langage de définition de données proposé. Ainsi, une table peut être créée en indiquant le *tablespace* dans lequel les données qui correspondent à cette table seront stockées.

Dans l'architecture que nous proposons nous avons ajouté le niveau ontologique au dessus du niveau logique. Donc, afin de permettre l'optimisation des traitements à ce niveau, le langage doit permettre d'accéder au niveau inférieur, c'est-à-dire le niveau logique.

Exigence 6 (*Définition, manipulation et interrogation du schéma des données*)

> *Le langage doit permettre de définir, manipuler et rechercher le schéma des données à partir de l'ontologie.*

La figure 2.1 montre comment les exigences que nous avons définies jusqu'à présent se positionnent par rapport à l'architecture que nous proposons. Les exigences 1, 2 et 3 requièrent de pouvoir effectuer des requêtes ontologiques construites à partir des définitions canoniques, non canoniques et linguistiques que peut fournir une ontologie. L'exigence 4 requiert de pouvoir utiliser différents modèles d'ontologies pour pouvoir représenter les ontologies d'une BDBO. Les exigences 5 et 6 concernent le niveau logique de notre architecture. Elles requièrent d'une part de pouvoir utiliser le langage SQL pour manipuler ce niveau et d'autre part de pouvoir passer du niveau ontologique au niveau logique afin de permettre l'optimisation des traitements.

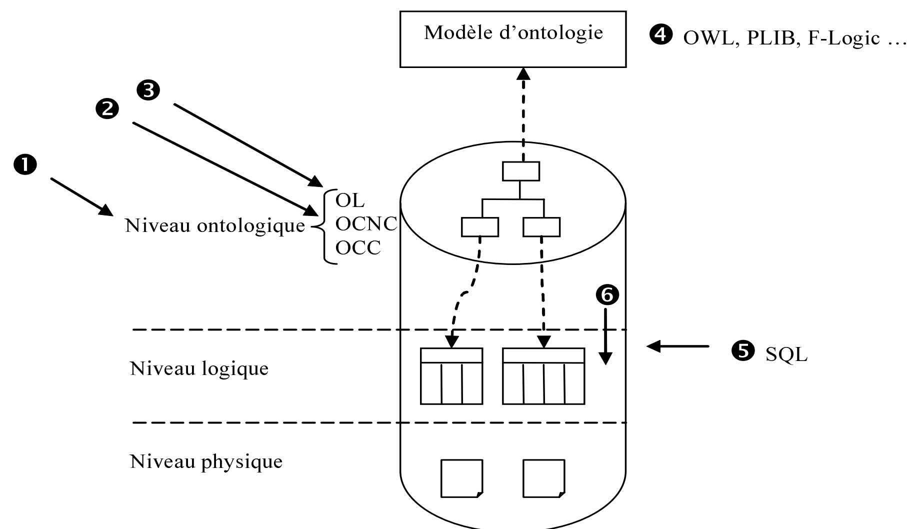


Fig. 2.1 – Positionnement des exigences par rapport à l'architecture de bases de données proposée

2.3 Exigences sur le pouvoir d'expression du langage

Une BDBO contient un ensemble d'ontologies constituées principalement de classes et de propriétés. Elles sont représentées par un modèle d'ontologies constituées d'entités et d'attributs. Le langage doit permettre de définir et manipuler ces ontologies et leur modèle.

Exigence 7 (*Définition et manipulation des ontologies et de leur modèle*)

> *Le langage doit permettre d'ajouter, modifier et supprimer les éléments constituants des ontologies tels que les classes ou les propriétés. Il doit également permettre d'ajouter, modifier et supprimer les entités et les attributs qui permettent de les représenter.*

Dans les bases de données traditionnelles, le langage de manipulation de données de SQL permet de définir les tuples des tables et leurs valeurs d'attributs. La manipulation de ces données est réalisée sous l'*hypothèse de typage exact* qui suppose que la table dans laquelle sont placés ces tuples est composée de l'ensemble des attributs permettant de décrire ces instances. Ce langage présente d'une part l'intérêt d'être déclaratif, c'est-à-dire de ne pas nécessiter la spécification des traitements nécessaires à la mise à jour des données. Il présente d'autre part l'intérêt d'avoir un pouvoir d'expression important, étant lié au langage d'interrogation de données.

Une BDBO contient les instances d'une ontologie. Ces instances sont caractérisées par des valeurs de propriétés. Cette caractérisation des instances se fait, comme nous l'avons indiqué (cf. section 2), sous l'hypothèse de typage ontologique fort qui autorise le schéma des instances d'une classe (ensemble des propriétés pour lesquelles elle présente une valeur) d'être composé seulement que d'un sous-ensemble des propriétés définies sur cette classe. Actuellement, pour permettre la mise à jour des données, la majorité des BDBO présentées au chapitre 1, section 4, sont équipées d'API (Application Programming Interface). Dans ces BDBO, la mise à jour des données est donc réalisée de façon procédurale et avec un pouvoir d'expression limité aux primitives fournies par ces API. Un langage permettant de manipuler les instances d'une BDBO sous l'hypothèse de typage fort, de manière déclarative et en étant couplé au langage d'interrogation de données, est donc utile. De plus, un langage de définition de données est nécessaire pour permettre de définir les classes et les propriétés qui décrivent ces instances.

Exigence 8 (*Définition et manipulation des données*)

> *Le langage doit permettre d'ajouter, modifier et supprimer les instances d'une BDBO ainsi que de définir les classes qu'elles instancient et les propriétés qui les caractérisent.*

Les ontologies sont des conceptualisations d'un domaine visant à couvrir les besoins de nombreuses applications. La conception d'une ontologie vise donc à décrire un domaine d'étude en étant le plus exhaustif possible. En conséquence, les ontologies contiennent généralement de nombreux concepts. Par exemple, l'ontologie IEC [IEC61360-4, 1999] portant sur le domaine des composants électroniques contient environ 200 classes et plus de 1000 propriétés. Or, les utilisateurs des ontologies contenues dans une BDBO sont rarement ses concepteurs. En conséquence, il est nécessaire de permettre à un utilisateur de découvrir les ontologies contenues dans une BDBO. Le langage doit donc permettre d'interroger les ontologies d'une BDBO. Il doit ainsi fournir des opérateurs pour retrouver par exemple les sous-classes d'une classe donnée ou les propriétés définies et héritées par une classe.

Exigence 9 (*Interrogation des ontologies*)

> *Le langage doit permettre d'exprimer des requêtes sur les éléments définis dans les ontologies contenues dans une BDBO ainsi que sur les attributs qui les décrivent.*

La satisfaction de l'exigence précédente permet à un utilisateur d'extraire une partie d'une ontologie. Lorsque cette ontologie est associée à des instances, il peut être nécessaire d'extraire ces instances en même temps que les ontologies. Ceci permet d'obtenir en une seule requête la description complète des instances de différentes classes. Dans les langages traditionnels, cette opération nécessite la composition de deux requêtes. Cette capacité est en particulier fondamentale dans les architectures distribuées (Web service, Peer-to-Peer, ...) où le trafic réseau doit être minimisé.

Par ailleurs, puisqu'une ontologie contient généralement la définition de nombreux concepts, la hiérarchie de classes qu'elle propose est souvent composée de plusieurs niveaux. Lorsqu'une requête est effectuée sur une classe d'un certain niveau, elle retourne l'ensemble des instances qui peuvent appartenir à cette classe ou à une de ses sous-classes. Elles peuvent ainsi être associées à une description ontologique différente qu'il peut être utile de pouvoir retrouver. Ces exemples montrent qu'il est nécessaire d'offrir la possibilité de combiner l'interrogation des ontologies et des données d'une BDBO.

Exigence 10 (*Interrogation à la fois des ontologies et des données*)

> *Le langage doit permettre d'exprimer des requêtes qui portent à la fois sur les ontologies et sur les données d'une BDBO. Il doit permettre de rechercher des classes et de retourner en même temps leurs instances (des ontologies vers les données). Il doit également permettre de retourner la description ontologique d'une instance (des données vers les ontologies).*

Les exigences définies dans cette section permettent d'offrir un pouvoir d'expression important pour exploiter l'architecture de BDBO proposée. Dans la section suivante, nous présentons les exigences qui sont généralement attendues pour un langage de bases de données et qui sont donc requises pour un langage supportant l'architecture de BDBO proposée.

2.4 Exigences sur l'implantation du langage

Les langages usuels des bases de données sont basés sur une sémantique formelle. Cette formalisation permet de prouver les propriétés fondamentales du langage (par exemple, la *complétude relationnelle*) tout en offrant une base solide à la fois pour l'implantation du langage et pour son étude théorique (expressivité, complexité, ou optimisation de requêtes). Une telle formalisation est ainsi requise pour le langage.

Exigence 11 (*Définition d'une sémantique formelle*)

> *Le langage doit être défini à partir d'une modélisation formelle permettant d'assurer la composition et la complétude du langage tout en fournissant une base solide pour son implantation et l'étude de son optimisation.*

Pour faciliter l'utilisation du langage SQL, de nombreux outils ont été développés. Le langage SQL peut ainsi être utilisé de manière interactive sur une base de données. Des éditeurs ont également été proposés pour permettre la construction aisée de requêtes sans avoir besoin de connaître la syntaxe SQL [Zloof, 1977]. Ce langage peut aussi être utilisé depuis un langage de programmation grâce à des interfaces telles que l'interface JDBC pour JAVA. Des outils similaires sont requis pour un langage de BDBO.

Exigence 12 (*Outils permettant l'exploitation du langage*)

> *L'implantation du langage doit être accompagnée d'outils permettant une exploitation de ce langage similaire à ce qui existe pour SQL.*

Les exigences définies dans cette section constituent une spécification de besoins concernant un langage d'exploitation de l'architecture ANSI/SPARC que nous avons proposée et les outils qui doivent lui être associés afin d'en faciliter l'utilisation. Un langage répondant à cet ensemble d'exigences, permettrait d'exploiter pleinement cette architecture.

3 Analyse des langages de BDBO conçus pour le Web Sémantique

De nombreux langages de BDBO ayant été proposés dans la littérature, il est nécessaire d'analyser si ces langages répondent aux exigences définies précédemment afin de déterminer si la conception d'un nouveau langage est nécessaire. Pour chaque langage considéré, nous avons ainsi relevé puis analysé les solutions qu'il propose pour répondre à ces différentes exigences. Ne pouvant pas être exhaustif, compte tenu de la multitude des langages proposés, nous avons choisi de ne présenter que l'analyse des langages que nous avons considérés comme étant les plus pertinents par rapport à nos exigences. Nous commençons par les langages de BDBO conçus dans le contexte du Web Sémantique.

Les langages proposés dans le contexte du Web Sémantique sont nombreux. Dans un état de l'art récent [Bailey et al., 2005], ces langages sont classés en sept catégories (famille de SPARQL, famille de RQL, langages inspirés de XPath, XSLT ou XQuery, langages en anglais controlé, langages avec règles réactives, langages déductifs et autres langages). Nous avons choisi de présenter dans cette section l'évaluation d'un représentant des deux catégories (famille de SPARQL et famille de RQL) qui correspondent le plus aux exigences que nous avons définies.

3.1 Les langages conçus pour le formalisme RDF

La première catégorie de langage est nommée « famille de SPARQL ». Elle regroupe les langages qui considèrent l'ensemble des informations, que ce soit les ontologies ou les données, comme un ensemble de triplets. Ils permettent ainsi d'interroger des données RDF sans savoir si le moteur de requêtes interprétera la sémantique qui peut être associée aux différents éléments de ces triplets comme par exemple le vocabulaire RDF-Schema. Dans cette catégorie se trouvent, en particulier, les langages SPARQL [Prud'hommeaux and Seaborne, 2006], RDQL [Seaborne, 2004], SquishQL [Miller et al., 2002] et TriQL [Carroll et al., 2005]. Nous avons choisi d'évaluer le langage SPARQL car il est en cours de standardisation par le W3C, très cité dans la littérature et utilisé dans de nombreux travaux dans le domaine du Web Sémantique.

3.1.1 Présentation du langage SPARQL

SPARQL [Prud'hommeaux and Seaborne, 2006] est un langage de requêtes pour des données représentées en RDF. Il propose quatre types de requêtes :

- les requêtes `SELECT` permettent d'extraire des informations du graphe RDF interrogé ;
- les requêtes `CONSTRUCT` permettent de créer de nouveaux triplets à partir du résultat d'une requête ;
- les requêtes `DESCRIBE` permettent d'obtenir la description d'une ressource donnée. Les spécifications de SPARQL ne précisent pas la nature de la description d'une ressource. Elles imposent seulement que cette description soit un sous-ensemble du graphe interrogé ;
- les requêtes `ASK` retournent un booléen indiquant si la requête à une solution ou n'en a pas.

Requête Select. Une requête SPARQL `SELECT` se présente sous la forme générale[11] suivante :

```
PREFIX namespacesList
SELECT variablesList
 [FROM sourcesRDFList]
 WHERE graphPattern
[ORDER BY expList]
```

La clause `PREFIX` permet d'indiquer des alias sur les espaces de noms utilisés dans la requête. La clause `FROM` permet d'indiquer la ou les sources RDF interrogées. Cette clause est optionnelle. Si elle n'est pas spécifiée, l'environnement dans lequel la requête est exécutée est chargé de fournir la source de triplets. Par exemple, lorsqu'une requête SPARQL est exécutée sur une BDBO, les données RDF interrogées sont celles contenues dans la BDBO. Dans les exemples fournis dans cette section, nous supposons que la source de données interrogée est implicite. La clause `WHERE` est constituée d'un ensemble de triplets pouvant contenir des variables (préfixées par `?`). Un interpréteur de requêtes SPARQL recherche les valeurs de ces variables pour lesquelles les triplets de la clause `WHERE` sont inclus dans le graphe RDF interrogé. Il retourne le sous-ensemble de ces valeurs correspondant aux variables spécifiées dans la clause `SELECT`. Ce résultat peut être trié en spécifiant des expressions dans la clause `ORDER BY`.

Exemple. Retourner la valeur des propriétés `name` et `email` pour les ressources décrites par ces propriétés. Les résultats doivent être triés par ordre croissant sur les noms.

```
  PREFIX sioc: <http://rdfs.org/sioc/ns#>
  SELECT ?name ?email
   WHERE { ?x sioc:name ?name .
           ?x sioc:email ?email }
ORDER BY ASC(?name)
```

Explication. Dans cette requête, la variable `?x` est introduite dans deux triplets (éventuellement séparés par un point) de la clause `WHERE`. Cette variable représente les ressources RDF présentant une valeur pour les propriétés `name` et `email` définies sur l'ontologie SIOC dont l'espace de nom est `http://rdfs.org/sioc/ns`. Les clauses `SELECT` et `ORDER BY` indiquent que cette requête retourne les noms (`?name`) et adresses email obtenues (`?email`) par ordre alphabétique croissant (`ASC`) sur les noms.

[11]Nous n'indiquons ici que les principales clauses d'une requête SPARQL.

SPARQL permet d'indiquer des conditions sur les variables utilisées dans la requête. Ces conditions sont définies dans la clause `WHERE` grâce à l'opérateur `FILTER`. Cet opérateur prend en paramètre une expression booléenne. Par exemple, l'ajout de `FILTER (?name = "Dupont")` dans la requête précédente permet de ne retourner que les noms et adresses emails des ressources dont le nom est `Dupont`.

SPARQL permet également d'indiquer que des triplets sont optionnels dans la clause `WHERE` grâce à l'opérateur `OPTIONAL`. En conséquence, des résultats ne satisfaisant pas les triplets optionnels seront quand même retournés. Et, lorsqu'une variable utilisée uniquement dans des triplets optionnels n'a pas d'affectation correspondant au graphe interrogé, la valeur `UNBOUND` est retournée. Par exemple, si nous modifions la requête de l'exemple précédent pour que le triplet `?x foaf:email ?email` soit optionnel (`OPTIONAL { ?x sioc:email ?email }`), les ressources présentant une valeur pour la propriété `name` mais pas pour la propriété `email` seront quand même retournées avec la valeur `UNBOUND` pour la variable `?email`.

SPARQL dispose également de l'opérateur `UNION` pour joindre deux ensembles de triplets dans la clause `WHERE`. Pour une requête où deux ensembles de triplets sont liés par cet opérateur, un résultat est retourné dès lors qu'il permet de satisfaire l'un des deux ensembles de triplets. Pour prendre un exemple, supposons qu'il existe une deuxième version de l'ontologie SIOC définissant un espace de noms référencé par l'alias `sioc2` et considérons la requête suivante :

```
  PREFIX sioc: <http://rdfs.org/sioc/ns#>
  PREFIX sioc2: <http://rdfs.org/sioc2/ns#>
  SELECT ?name ?email
   WHERE { { ?x sioc:name ?name .
             ?x sioc:email ?email }
           UNION
          { ?x sioc2:name ?name .
            ?x sioc2:email ?email } }
ORDER BY ASC(?name)
```

Cette requête complète celle de l'exemple précédent pour retourner les ressources ayant un nom et une adresse email quelle que soit la version de l'ontologie SIOC utilisée.

Requête Construct. Une requête `CONSTRUCT` propose les clauses `FROM`, `WHERE` et `ORDER BY` présentées précédemment. Par contre, dans de telles requêtes, la clause `SELECT` est remplacée par la clause `CONSTRUCT`. Cette clause est construite comme une clause `WHERE`, c'est-à-dire par un ensemble de triplets pouvant contenir des variables. Le résultat d'une telle requête est un nouveau graphe RDF construit en substituant les variables contenues dans la clause `CONSTRUCT` par les valeurs qui satisfont la clause `WHERE`. L'exemple suivant montre comment une requête `CONSTRUCT` permet de construire un nouveau graphe RDF utilisant les propriétés définies dans l'ontologie référencée par l'alias `sioc2` à partir du graphe RDF utilisant l'ontologie référencée par l'alias `sioc`. Pour simplifier cet exemple ainsi que les suivants, les espaces de noms ne sont pas précisés.

Exemple. Créer chaque ressource de l'ontologie `sioc` décrite par les propriétés `name` et `email` dans l'ontologie `sioc2`. Dans cette ontologie, ces ressources devront être caractérisées par les propriétés `name` et `email` de l'ontologie `sioc2`.

```
CONSTRUCT { ?x sioc2:name ?name
            ?x sioc2:email ?email }
    WHERE { ?x sioc:name ?name .
            ?x sioc:email ?email }
```

Explication. Pour chaque ressource décrite par un nom et une adresse email grâce aux propriétés définies dans l'ontologie `sioc`, deux triplets sont retournées dans le graphe résultat qui utilisent cette fois-ci les propriétés `name` et `email` définies dans l'ontologie `sioc2`.

Nous ne détaillons pas les requêtes `ASK` et `DESCRIBE` disponibles en SPARQL qui ne permettent pas de répondre à plus d'exigences que les requêtes que nous venons de présenter.

3.1.2 Analyse du langage SPARQL par rapport aux exigences définies

Exigence 1 (Expression de requêtes niveau ontologique)

Le langage SPARQL permet d'exprimer des requêtes sur les données à partir des ontologies. Cette capacité est illustrée dans l'exemple suivant.

Exemple. Rechercher les `Item` qui ont eu une réponse.

```
SELECT ?i
 WHERE { ?i rdf:type sioc:Item .
         ?i sioc:has_reply ?r }
```

Explication. Le premier triplet caractérise les instances de la classe `Item` (`?i`). Le second permet de rechercher, parmi ces instances, celles qui présentent une valeur pour la propriété `has_reply`.

Cependant, le résultat de la requête précédente est directement dépendant des triplets représentés dans la BDBO. Si pour une classe `C`, la BDBO représente un triplet `(i, rdf:type, C)` pour chaque instance directe ou indirecte de `C`, alors la requête précédente retourne également les instances de la classe `Post`, sous-classe de la classe `Item`. Si, par contre, elle ne représente un triplet `(i, rdf:type, C)` que pour les instances directes de `C`, (les autres liens pouvant se déduire de la relation de subsomption), cette requête ne retournera que des instances de la classe `Item` à moins que l'interpréteur de requêtes ne réalise la clôture transitive de la relation de subsomption. Ainsi, cette requête retournera soit les instances directes de la classe `Item` soit également ses instances indirectes selon les triplets représentés dans la BDBO et l'interprétation qui en est faite.

Le langage SPARQL ne répond donc que partiellement à notre exigence de pouvoir exprimer des requêtes ontologiques indépendamment de la représentation au niveau logique des données. Il permet d'exprimer de telles requêtes mais le résultat de ces requêtes est dépendant des triplets RDF représentés dans la BDBO ou de leur interprétation.

Exigence 2 (Définition de concepts non canoniques)

SPARQL ne propose pas de langage de définition de données. Il ne permet donc pas de définir des concepts non canoniques. Cependant, si ces concepts sont construits manuellement ou via une API dans

la BDBO, SPARQL permet d'en calculer automatiquement l'extension à partir des concepts canoniques comme le montre l'exemple suivant.

Exemple. Calculer l'extension de la classe `ForumWithoutHost` définie comme étant l'ensemble des forums qui n'ont pas d'hôte.

```
CONSTRUCT { ?f rdf:type sioc:ForumWithoutHost }
    WHERE { ?f rdf:type sioc:Forum .
            OPTIONAL {?f sioc:has_host ?h} FILTER ( !bound(?h) ) }
```

Explication. La clause `WHERE` de cette instruction permet de rechercher les instances de la classe `Forum` qui n'ont pas de valeur pour la propriété `has_host` en utilisant la fonction `bound` dans une clause `FILTER`. Cette fonction retourne vrai si et seulement si la variable prise en paramètre n'a pas la valeur `UNBOUND.` La clause `CONSTRUCT` permet d'associer les instances trouvées à la classe `Forum-WithoutSite` en ajoutant un triplet de typage.

SPARQL satisfait donc partiellement l'exigence 2. Il ne permet pas de définir des concepts non canoniques mais permet par contre d'en calculer l'extension via des requêtes `CONSTRUCT`. Notons cependant que cette extension doit être modifiée explicitement chaque fois que l'extension des concepts canoniques sur lesquels elle se base est modifiée (différence avec les vues dans les bases de données).

Exigence 3 (Exploitation linguistique)

SPARQL propose des opérateurs pour exploiter le fait que les valeurs de type chaîne de caractères en RDF peuvent prendre une valeur pour plusieurs langues naturelles. Il permet d'une part de préciser dans une requête la langue naturelle dans laquelle une chaîne de caractères doit être évaluée (suffixe `@lg` où `lg` est un code à deux lettres identifiant une langue naturelle donnée). De plus, il fournit des fonctions pour faciliter l'exploitation du multilinguisme. Par exemple, la fonction `lang` permet de retrouver la langue dans laquelle une chaîne de caractères est donnée. SPARQL satisfait donc l'exigence 3.

Exigence 4 (Indépendance par rapport à un modèle d'ontologies donné)

SPARQL est défini pour le langage RDF et donc indépendamment d'un modèle d'ontologies donné. SPARQL considère ainsi une ontologie définie dans un modèle d'ontologies particulier comme un ensemble de triplets utilisant les constructeurs d'un modèle d'ontologies décrit également en RDF dans un autre espace de noms. Il permet donc d'interroger toute ontologie exprimée en RDF mais cela sans attribuer une quelconque sémantique aux constructeurs du modèle d'ontologies utilisé.

Exigences 5 et 6 (Compatibilité avec l'architecture traditionnelle des bases de données)

Même si la syntaxe proposée par SPARQL suit la forme d'une requête SQL (`SELECT-FROM-WHERE`), elle est adaptée à l'interrogation de triplets et diverge donc de SQL (exigence 5). D'autre part, le schéma des données étant considéré comme immuable (sujet, prédicat, objet), le langage SPARQL ne fournit pas d'opérateur pour définir, manipuler et interroger ce schéma (exigence 6).

Exigences 7 et 8 (Définition et manipulation des ontologies et des données)

Comme nous l'avons indiqué précédemment, le langage SPARQL ne propose pas de langage de définition et de manipulation de données. En conséquence, les données RDF doivent être modifiées manuellement dans les fichiers RDF utilisés ou par API si ces données sont stockées dans une BDBO.

Exigences 9 et 10 (Interrogation des ontologies et à la fois des ontologies et des données)

SPARQL permet l'interrogation des ontologies en les considérant comme un ensemble de triplets (exigence 9). SPARQL permet également d'interroger à la fois les ontologies et les données en permettant d'utiliser une variable à la place d'un prédicat (exigence 10). Cette capacité est illustrée dans l'exemple suivant.

Exemple. Rechercher l'ensemble des classes OWL dont le nom commence par `For` avec les URI des instances de ces classes ainsi que leurs valeurs de propriétés.

```
SELECT ?c ?i ?p ?val
 WHERE { ?c rdf:type owl:Class .
         ?c rdfs:label ?l . FILTER REGEX (?l, "^For") .
         ?p rdfs:domain ?c .
         ?i rdf:type ?c .
         ?i ?p ?val }
```

Explication. Les deux premières lignes de la clause `WHERE` permettent de rechercher les classes OWL dont le nom commence par `For`. Le triplet suivant permet de retrouver les propriétés attachées à ces classes via le prédicat `rdfs:domain`. Les deux derniers triplets permettent de retrouver les instances des classes trouvées et de rechercher la valeur de ces instances pour les propriétés déterminées précédemment.

Exigences 11 et 12 (Implantation du langage)

La sémantique du langage SPARQL est définie formellement dans les spécifications de ce langage [Prud'hommeaux and Seaborne, 2006]. Une sémantique compositionnelle lui a également été associée dans [Pérez et al., 2006]. Enfin, les outils conçus autour de ce langage sont nombreux[12]. Elles permettent une utilisation aisée de ce langage.

Synthèse et conclusion

L'analyse du langage SPARQL par rapport aux exigences définies est synthétisée dans le tableau 2.1. Nous utilisons le symbole • pour indiquer qu'une solution est proposée pour répondre à une exigence. Lorsque cette solution ne permet de satisfaire que partiellement une exigence, nous utilisons le symbole ◦. Enfin, le symbole - est utilisé si aucune solution n'est proposée pour une exigence donnée. Nous utiliserons des tableaux similaires pour tous les langages analysés.

Le langage SPARQL est donc un langage très puissant pour l'interrogation de triplets RDF. Il propose une syntaxe simple et donc facile à appréhender. La simplicité du modèle de données sous-jacent

[12] cf. `http://esw.w3.org/topic/SparqlImplementations`

Exigences	SPARQL
Expression de requête niveau ontologique	○
Définition de concepts non canoniques	○
Exploitation linguistique	•
Indépendance par rapport à un modèle d'ontologies donné	○
Compatibilité avec SQL	-
Définition, manipulation et interrogation du schéma des données	-
Définition et manipulation des ontologies	-
Définition et manipulation des données	-
Interrogation des ontologies	•
Interrogation à la fois des ontologies et des données	•
Définition d'une sémantique formelle	•
Outils permettant l'exploitation du langage	•

TAB. 2.1 – Analyse du langage SPARQL par rapport aux exigences définies

et l'utilisation d'un typage faible permettent d'interroger des données basées sur un modèle d'ontologies quelconque à condition toutefois que tous les triplets soient explicitement représentés. Notons que lorsqu'une table unique est utilisée pour représenter les données, dans laquelle, de plus, toutes les relations transitives (par exemple l'héritage) doivent être saturées, ce langage devient inopérant pour des problèmes de grande taille. En effet, les requêtes nécessitent alors souvent des auto-jointures de la table de triplets [Dehainsala et al., 2007a]. Ce problème est actuellement l'objet de plusieurs travaux de recherche [Abadi et al., 2007, Wilkinson, 2006, Chong et al., 2005].

Dans le contexte de l'architecture de BDBO que nous avons proposée, le langage SPARQL présente les limitations suivantes :

– le résultat des requêtes est dépendant des triplets représentés dans la BDBO et de l'interprétation qui en est faite par le moteur de requêtes. Cette dépendance va à l'encontre de notre exigence de disposer d'un langage permettant d'interroger les données au niveau ontologique indépendamment de la représentation logique des données ou de leur interprétation ;
– le langage SPARQL ne propose pas de langages de définition et de manipulation de données qui, dans le cadre d'une BDBO, sont nécessaires pour ne pas avoir à modifier ni à recharger des données contenues dans des fichiers RDF ;
– le langage SPARQL ne permet pas de manipuler les données d'une BDBO au niveau logique.

3.2 Les langages conçus pour le modèle d'ontologies RDF-Schema

La seconde catégorie de langage est nommée « famille de RQL ». Elle regroupe les langages qui font la distinction entre les ontologies et les données (instances des classes). Le modèle d'ontologies supporté par ces langages est RDF-Schema. Ainsi, contrairement aux langages de la famille de SPARQL, ils fournissent des opérateurs permettant d'exploiter la sémantique du vocabulaire introduit par RDF-Schema tel que la relation de subsomption entre classes. Dans cette catégorie se trouvent, en particulier,

les langages RQL [Karvounarakis et al., 2002], SeRQL [Broeskstra and Kampman, 2003] et eRQL [Tolle and Wleklinski, 2004]. Nous avons choisi d'évaluer le langage RQL, le plus complet et le plus référencé dans la littérature.

3.2.1 Présentation du langage RQL

La proposition autour du langage RQL est composée de trois langages : un langage d'interrogation de données (RQL), un langage de manipulation de données (RUL) et un langage de définition de vues (RVL). Chacun de ces différents langages permet de répondre à certaines de nos exigences. Nous les présentons donc dans les sections suivantes.

3.2.2 Le langage d'interrogation de données (RQL)

Le langage RQL [Karvounarakis et al., 2002] a été l'un des premiers langages permettant d'interroger des données et des ontologies représentées en RDF-Schema. Il est basé sur un modèle de données formel qui distingue les niveaux données, ontologies et modèles d'ontologies. Le niveau modèle d'ontologies est composé des entités `Class` et `Property`. Ces entités peuvent être spécialisées en utilisant le constructeur RDF-Schema `subClassOf`. Le niveau ontologie est composé de classes, instances de l'entité `Class`, et de propriétés, instances de l'entité `Property`. Enfin, le niveau données est constitué des instances de ces classes et de leurs valeurs de propriétés. Par rapport au modèle RDF-Schema, le modèle de données de RQL présente les restrictions suivantes :

– une classe de niveau ontologie ne peut pas être subsumée par une entité de niveau modèle d'ontologies ;
– le domaine et codomaine d'une propriété doivent toujours être définis et doivent être uniques ;
– les cycles dans la hiérarchie de subsomption sont interdits.

Ces restrictions ont pour but d'une part de distinguer clairement les différents niveaux d'abstraction introduits par RDF-Schema et d'autre part de clarifier la sémantique de ce modèle.

Le langage RQL, basé sur ce modèle, a été conçu selon une approche fonctionnelle similaire à l'approche suivie pour la conception du langage OQL [Cattell, 1993]. Ainsi, il permet de poser des requêtes de base simplement en appelant des fonctions. Par exemple, l'appel de la fonction `SubClassOf(Personne)` est une requête RQL retournant l'ensemble des sous-classes de la classe `Personne`. L'ensemble des constructeurs du modèle d'ontologies RDF-Schema (par exemple, `subPropertyOf`, `domain` ou `range`) sont ainsi codés dans la grammaire RQL comme des mots clés faisant référence à une fonction prédéfinie. Des requêtes de base peuvent également être exprimées pour interroger les données d'une BDBO en utilisant le nom des classes et des propriétés comme une fonction retournant leur extension. Par exemple, la requête `Forum` retourne l'ensemble des instances de la classe du même nom tandis que la requête `title` retourne des couples `(i,v)` où `i` est une instance qui présente la valeur `v` pour la propriété `title`.

Le langage RQL propose également une syntaxe `SELECT-FROM-WHERE` permettant d'exprimer des requêtes avec un pouvoir d'expression supérieur aux requêtes de base. La clause `FROM` introduit des *ex-*

pressions de chemins, séparées par des virgules, dans le graphe RDF-Schema interrogé. Ces expressions de chemins peuvent être des expressions de chemins de base définies dans RQL (listées dans [Karvounarakis et al., 2004]). Chacune de ces expressions de chemins de base identifie une liste de noeuds et/ou d'arêtes dans le graphe interrogé via des variables. Ces expressions de chemins de base retournent ainsi une relation dont les attributs sont les variables introduites. Par exemple, l'expression `c{X}`, où `c` est une classe, retourne une relation unaire dont l'unique attribut est la variable `X` et qui contient l'ensemble des instances (directes et indirectes) de la classe `c`. Ces expressions de chemins de base sont combinées en fonction des variables qu'elles ont en commun. Si elles n'ont aucune variable en commun le produit cartésien est réalisé entre les relations qu'elles produisent. Sinon, la jointure est réalisée sur les variables (attributs des relations) qu'elles ont en commun. Une expression de chemin présente dans la clause `FROM` peut également être construite en concaténant des expressions de chemins de base avec l'opérateur '`.`'. Cet opérateur réalise la jointure implicite entre les relations produites par les expressions de chemins de base. La clause `WHERE` permet de définir des conditions sur les variables introduites dans la clause `FROM`. Le résultat de la requête est défini en projetant les variables dont les valeurs participent au résultat dans la clause `SELECT`.

Pour interroger des données, des ontologies et à la fois des ontologies et des données, RQL propose des expressions de chemins de base à ces différents niveaux. Nous illustrons ces capacités sur quelques exemples. Le premier exemple est une requête sur les données.

Exemple. Retourner les utilisateurs dont l'adresse email se termine par `@ensma.fr`.

```
SELECT X, Y
  FROM sioc:User{X}, {X}sioc:email{Y}
 WHERE Y LIKE "%@ensma.fr"
 USING NAMESPACE sioc=&http://rdfs.org/sioc/ns#
```

Explication. RQL permet de définir l'alias (`sioc`) sur l'espace de noms `http://rdfs.org/sioc/ns` (doit être préfixé par &) dans la clause `USING NAMESPACE`. Cet alias peut être utilisé pour identifier les éléments utilisés dans les requêtes. Par exemple, l'expression `sioc:User` identifie la classe `User` définie dans l'ontologie SIOC. La clause `FROM` est composée de deux expressions de chemins de base définies dans RQL impliquant la classe `User` et la propriété `email` de l'ontologie SIOC (ces éléments sont préfixés par l'alias `sioc`). `User{X}` introduit la variable `X` sur l'ensemble des instances de la classe `User`. L'expression `{X}email{Y}` retourne les instances `X` ayant une valeur `Y` pour la propriété `email`. La variable `X` étant commune à ces deux expressions de chemins de base, une jointure est réalisée permettant ainsi que la variable `Y` ne parcourt que les valeurs de la propriété `email` pour les instances de la classe `User`. Notons que cette jointure aurait également pu être réalisée implicitement par l'expression `sioc:User{X}.sioc:email{Y}`. La variable `Y` est soumise à une condition dans la clause `WHERE` permettant de ne récupérer que les instances dont la valeur de la propriété `email` se termine par `@ensma.fr`. Enfin, l'utilisation des variables `X` et `Y` dans la clause `SELECT` permet de retrouver l'URI des instances parcourues ainsi que leur valeur pour la propriété `email`. Notons que cette requête aurait pu être écrite en RQL sans préciser que les instances recherchées doivent appartenir à la classe `User` puisque cette classe est le domaine de la propriété `email`. Nous qualifions ce type de requêtes, où le type des instances est implicite, de *requêtes non typées*.

Le deuxième exemple est une requête sur les ontologies.

Exemple. Retourner les propriétés applicables sur la classe User ainsi que leur codomaine.

```
SELECT @P, range(@P)
  FROM {$C}@P
 WHERE $C = sioc:User
```

Explication. En RQL, les variables préfixées par $ itèrent sur les classes tandis que celles préfixées par @ itèrent sur les propriétés. L'expression {$C}@P est une expression de chemin de base définie qui retourne des couples ($C,@P) où chaque $C est la classe qui est le domaine de la propriété @P ou une sous-classe de ce domaine. La restriction indiquée dans la clause WHERE spécifie que $C est la classe User et donc que @P est une propriété définie sur User ou une de ses super-classes. La projection définie dans la clause SELECT retourne la propriété @P ainsi que son codomaine grâce à la fonction range.

Enfin, voici un exemple combinant l'interrogation des ontologies et des données.

Exemple. Retourner les informations de niveau ontologie et de niveau données concernant l'instance dont l'URI est www.lisi.ensma.fr/Dupont.

```
1. SELECT X,
2.       (SELECT $W,
3.               (SELECT @P, Y
                    FROM {X;$W}@P{Y})
           FROM $W{X})
    FROM Resource{X}
   WHERE X = "www.lisi.ensma.fr/Dupont"
```

Explication. Cette requête est composée d'une requête principale (1.) et de deux sous-requêtes imbriquées (2. et 3.). La requête principale (1.) recherche à partir de la classe racine (Resource) l'instance X dont l'URI est www.lisi.ensma.fr/Dupont. La première sous-requête imbriquée (2.) utilise l'expression de chemin de base $W{X} qui retourne les classes $W et leurs instances directes X. Ainsi, $W parcourt les classes de base de l'instance www.lisi.ensma.fr/Dupont. Enfin, la seconde sous-requête imbriquée (3.) utilise l'expression de chemin de base {X;$W}@P{Y}. Cette expression retourne les instances (X), leurs classes de base ($W) et leurs valeurs (Y) de propriétés (@P). Ainsi, elle permet de rechercher pour chacune des classes de base de l'instance d'URI www.lisi.ensma-.fr/Dupont, les propriétés définies sur cette classe ainsi que ses valeurs pour ces propriétés. Le résultat de cette requête est un couple ('www.lisi.ensma.fr/Dupont', l). l est une liste de couples (c, pv) où c est une classe de base de l'instance www.lisi.ensma.fr/Dupont et pv est la liste des couples propriétés-valeurs de cette instance pour les propriétés définies sur la classe c.

Ces différents exemples montrent la puissance d'expression offerte par RQL pour interroger une BDBO.

3.2.3 Le langage de manipulation de données (RUL)

RUL [Magiridou et al., 2005] est une proposition de langage de manipulation de données de BDBO supportant le modèle d'ontologies RDF-Schema. Il fournit pour cela trois opérations : INSERT, DELETE

et MODIFY. Ces opérations sont définies pour les classes et pour les propriétés.

Pour les classes, ces opérations permettent d'ajouter, modifier et supprimer des liens d'instanciation entre des instances et une classe. Ainsi, ces opérations ne manipulent pas directement les instances. La manipulation de ces instances n'est qu'un « effet de bord » de ces opérations. Prenons l'exemple de l'opération DELETE. Son rôle n'est pas de supprimer une instance i, il est de supprimer le lien d'instanciation entre cette instance et une classe C. Si C a des super-classes, l'instance i n'est pas supprimée. Des liens d'instanciations entre i et ses super-classes sont en effet créés. Seul dans les cas où C n'a pas de super-classe, l'instance est supprimée.

La sémantique de ces opérations est définie pour assurer que seuls les liens d'instanciation entre une instance et ses classes de base soient représentés. Par exemple, si une instance i appartenant à une classe C est insérée dans une classe C_{sub}, sous-classe de C, alors le lien d'instanciation entre i et C est supprimé et celui entre i et C_{sub} est ajouté.

Cette sémantique est également définie pour respecter la sémantique de RDF-Schema. Ainsi, si un lien d'instanciation entre une instance i et une classe C est supprimé, i ne peut plus présenter de valeurs pour les propriétés définies sur C et ainsi, ces valeurs de propriétés sont supprimées.

Voici quelques exemples illustrant la syntaxe et la sémantique de ces opérations.

Exemple. Ajouter l'instance d'URI http://www.lisi.ensma.fr/Dupont à la classe User.

```
INSERT User(&http://www.lisi.ensma.fr/Dupont)
```

Explication. Si l'instance http://www.lisi.ensma.fr/Dupont n'existe pas dans la BDBO, cette opération l'ajoute comme instance de la classe User. Sinon, si elle appartenait déjà à une sous-classe de User, cette instruction est sans effet. Si elle appartenait à la classe Resource, super-classe de User, le lien d'instanciation entre cette instance et Resource est remplacé par un lien d'instanciation entre cette instance et User.

Les instances à ajouter, modifier ou supprimer d'une classe peuvent déjà exister dans la BDBO. En conséquence, RUL permet d'utiliser les clauses FROM et WHERE de RQL afin de rechercher ces instances.

Exemple. Supprimer de la classe Post les messages dont le titre ne contient pas le mot Forum.

```
DELETE Post(X)
  FROM Post{X}.title{Y}
 WHERE NOT Y LIKE "%Forum%"
```

Explication. Les clauses FROM et WHERE de cette expression permettent de rechercher les messages dont le contenu ne contient pas le mot Forum. La clause DELETE permet de supprimer le lien entre ces instances, référencées par la variable X, et la classe Post. Comme effet de bords, cette opération (1) ajoutera des liens d'instanciation entre ces instances et la classe Item, super-classe de Post et (2) supprimera les valeurs de propriétés de ces instances qui ne sont pas définies sur la classe Item, comme par exemple title.

Exemple. Attribuer l'URI http://www.lisi.ensma.fr/msg1 au message dont le titre est : L'ontologie SIOC.

```
MODIFY Post(X <- &http://www.lisi.ensma.fr/msg1)
  FROM {X}title{Y}
 WHERE Y = "L'ontologie SIOC"
```

Explication. Les clauses `FROM` et `WHERE` de cette expression permettent de rechercher le message `X` dont le titre est « `L'ontologie SIOC` ». La clause `MODIFY` supprime ce message et le remplace par un message dont l'URI est `http://www.lisi.ensma.fr/msg1`. Ainsi, l'ensemble des valeurs de propriétés définies (respectivement portant) sur la ressource `X` sont maintenant des valeurs de propriétés de la ressource `http://www.lisi.ensma.fr/msg1`.

Pour les propriétés, les opérations `INSERT`, `DELETE` et `MODIFY` permettent d'ajouter, modifier et supprimer des valeurs de propriétés associées à une instance. Voici un exemple de caractérisation d'une instance.

Exemple. Indiquer que l'adresse email de l'instance d'URI `www.lisi.ensma.fr/Dupont` est `dupont@ensma.fr`.

```
INSERT email(&www.lisi.ensma.fr/Dupont, "dupont@ensma.fr")
```

Explication. Cette opération caractérise l'instance `www.lisi.ensma.fr/Dupont` par la valeur `dupont@ensma.fr` pour la propriété `email`. Cette opération n'est valide que si l'instance `www.lisi.ensma.fr/Dupont` appartient au domaine de la propriété `email`, c'est-à-dire la classe `User`.

Comme pour les classes, la sémantique de ces opérations est définie pour assurer qu'il n'y ait pas de redondance entre l'extension d'une propriété et celle de ses sous-propriétés. Ainsi, si un couple `(i, v)` appartenant à l'extension d'une propriété `p` est insérée dans l'extension d'une propriété p_{sub}, sous-propriété de `p`, alors le couple `(i,v)` est supprimé de l'extension de `p` et est ajouté à l'extension de p_{sub}.

Le langage RUL permet donc de manipuler les instances d'une BDBO ainsi que leurs valeurs de propriétés. Il présente les caractéristiques suivantes :

- manipulation des données au travers des liens d'instanciation ;
- respect de la sémantique de RDF-Schema par effet de bord (par exemple, suppression des valeurs de propriétés non applicables sur une instance) ;
- représentation uniquement des liens d'instanciation directs.

3.2.4 Le langage de définition de vues (RVL)

L'utilisation du langage RQL nécessite de connaître l'ontologie interrogée. Les requêtes doivent en effet être écrites selon les classes et les propriétés représentées. Elles ne peuvent pas être écrites selon la perception qu'un utilisateur a de ce domaine. Le langage RVL [Magkanaraki et al., 2004] a été conçu pour résoudre ce problème.

Ce langage permet de créer une nouvelle ontologie dont les éléments (classes, propriétés et instances) peuvent être importés ou dérivés à partir des éléments d'une autre ontologie. Ce langage permet donc de construire une OCNC dont les concepts non canoniques sont représentés comme des vues. La création de telles vues suit les étapes suivantes.

1. Création de l'espace de noms de l'ontologie dans lesquelles les vues vont être créées. Par exemple, l'instruction suivante crée l'espace de noms `http://www.lisi.ensma.fr/SIOC.rdf` dont l'alias est `myview`.

   ```
   CREATE NAMESPACE myview=&http://www.lisi.ensma.fr/SIOC.rdf#
   ```

2. Création/importation des classes et des propriétés dans cette ontologie. La création de classes se fait en utilisant le constructeur `rdfs:Class` qui prend en paramètre l'identifiant de la classe créée. La création de propriétés se fait en utilisant le constructeur `rdf:Property` qui prend en paramètre l'identifiant de la propriété créée ainsi que son domaine et son codomaine. Les relations de subsomption entre classes et entre propriétés peuvent être indiquées via l'opérateur de subsomption noté < >. Par exemple, les instructions suivantes permettent de créer les classes `PostSioc`, `PostOntology` (super-classe de `PostSioc`) et `VUser` ainsi que la propriété `creates` dont le domaine est `VUser` et le codomaine est `PostSioc`.

   ```
   VIEW rdfs:Class("PostSioc"), rdfs:Class("PostOntology"),
        rdfs:Class("VUser"), PostOntology<PostSioc>,
        rdf:Property("creates", VUser, PostSioc)
   ```

   L'importation de classes et de propriétés d'une autre ontologie peut être réalisée en utilisant une requête RQL portant sur les ontologies. Par exemple, la requête suivante permet d'importer toutes les classes dont l'espace de noms a pour alias `sioc` qui sont des sous-classes (peut être exprimé avec l'opérateur < en RQL) de la classe `Space`.

   ```
   VIEW Class(X)
   FROM Class{X}
   WHERE namespace(X) = \"sioc\" and X < sioc:Space
   ```

3. Définition de l'extension des classes et des propriétés en utilisant le langage RQL. L'utilisation du langage RQL pour définir l'extension des classes et des propriétés est similaire à l'utilisation de requêtes `CONSTRUCT` en SPARQL. Voici un exemple qui illustre cela.

   ```
   VIEW PostSioc(Y),VUser(X),creates(X,Y),name(X,W)
   FROM {Y;sioc:Post}sioc:createdBy{X}.sioc:name{W}, {Y}sioc:title{Z}
   WHERE Z like "%ontologie%"
   USING NAMESPACE sioc = &http://rdfs.org/sioc/ns#
   ```

   Les clauses `FROM` et `WHERE` sont des clauses RQL. Elles permettent de rechercher les messages qui contiennent le mot `ontologie` dans leur titre ainsi que l'auteur de ce message. La clause `VIEW` spécifie comment les extensions des classes et des propriétés de l'OCNC seront définies à partir des variables introduites dans les deux autres clauses. Dans cet exemple, les messages et les utilisateurs retournés sont insérés respectivement dans les classes `PostSioc` et `VUser`. Ces utilisateurs sont caractérisés par les propriétés `create` et `name` dont les extensions sont définies à partir des propriétés correspondantes (`name`) ou inverses (`createdBy`) dans l'ontologie source.

4. Ajout d'instances aux classes et aux propriétés. RVL permet non seulement de définir l'extension des classes et propriétés créées à partir d'une requête RQL mais aussi de les instancier. Par contre,

les instances ainsi créées ne sont pas ajoutées aux classes à partir desquelles l'extension de cette classe peut être définie (étape 3). Pour ajouter des instances, RVL permet d'utiliser les noms des classes et des propriétés comme des constructeurs. Les instructions suivantes permettent de créer une instance de la classe `PostOntology` dont l'URI est `http://www.lisi.ensma.fr/SemWeb`, caractérisée par la valeur `Semantic Web` pour la propriété `title`.

```
VIEW PostOntology(&http://www.lisi.ensma.fr/SemWeb),
     title(&http://www.lisi.ensma.fr/SemWeb, "Semantic Web")
```

Le langage RVL permet donc de construire des concepts non canoniques à partir des concepts canoniques d'une ontologie. La caractéristique de ce langage est de distinguer clairement ces deux types de concepts. En effet, ils sont définis dans des espaces de noms différents. De plus, les concepts non canoniques peuvent être instanciés manuellement sans que les instances créées ne soient ajoutées aux concepts canoniques à partir desquels ils ont été définis.

Ayant présenté les langages d'interrogation, de manipulation de données et de définition de vues proposés par RQL, nous sommes maintenant en mesure d'analyser ce langage par rapport à nos exigences.

3.2.5 Analyse du langage RQL par rapport aux exigences définies

Exigence 1 (Expression de requêtes niveau ontologique)

Le langage RQL permet d'exprimer des requêtes sur les données à partir des ontologies dont le résultat est indépendant de la représentation des données dans la BDBO. Le langage RQL satisfait donc l'exigence 1.

Exigence 2 (Définition de concepts non canoniques)

Le langage de vue associé à RQL (RVL) permet de représenter des opérateurs non canoniques tels que les restrictions de OWL. Cette capacité est illustrée dans l'exemple suivant.

Exemple. Construire la classe `UserDupont` définie comme étant l'ensemble des utilisateurs portant le nom `Dupont`. Cette classe est donc une restriction OWL `hasValue` portant sur la propriété `last_name` dont la valeur doit être `Dupont`.

```
CREATE NAMESPACE myview=&http://www.lisi.ensma.fr/exemple-view.rdf#
VIEW rdfs:Class("UserDupont"),
     Property(P, UserDupont, range(P))
     FROM Property{P}
     WHERE domain(P) >= sioc:User
VIEW UserDupont(U), first_name(U, FN), last_name(U, LN)
FROM sioc:User{U}.sioc:first_name{FN}, {U}last_name{LN}
WHERE LN = "Dupont"
```

Explication. En RVL, les vues sont distinctes des classes servant à les construire. Nous devons donc créer un nouvel espace de noms (`http://www.lisi.ensma.fr/exemple-view.rdf`) dans lequel seront

définis les concepts non canoniques. Dans cette espace de noms, la classe non canonique `User-Dupont` est créée et chaque propriété applicable sur la classe `User` de l'ontologie `sioc` est importée dans cet espace de noms en la rattachant via son domaine à la classe `UserDupont`. L'instruction `VIEW` suivante permet de peupler cette classe en recherchant les instances de la classe `User` ayant pour nom `Dupont`. Elle permet également de caractériser ces instances dans la vue. Dans cet exemple, pour rester concis, nous avons seulement recherché les valeurs des propriétés `first_name` et `last_name` pour ses instances. Pour obtenir la restriction OWL il faudrait rechercher également les valeurs des autres propriétés applicables sur la classe `User`.

Cet exemple montre que RVL permet de créer des concepts non canoniques. Notons par contre que la distinction entre les concepts non canoniques et les concepts canoniques ne permet pas de définir de relation de subsomption entre ces deux types de concepts. En conséquence, il est nécessaire de reproduire ce comportement manuellement (importation des propriétés et de leur extension dans l'OCNC).

Exigence 3 (Exploitation linguistique)

Nous avons vu que RDF-Schema permet d'associer chaque classe à des noms définis en plusieurs langues naturelles. Il permet également de définir la valeur de chaînes de caractères dans plusieurs langues naturelles. RQL n'exploite pas ces définitions linguistiques.

Exigence 4 (Indépendance par rapport à un modèle d'ontologies donné)

Pour gérer la diversité des modèles d'ontologies, le modèle de données sur lequel repose le langage RQL n'est pas complètement figé. En effet, ce modèle de données peut être étendu en spécialisant les entités prédéfinies `Class` et `Property`. RQL permet ainsi certaines extensions du modèle d'ontologies mais il ne permet pas d'ajouter des entités si elles n'héritent pas de ces deux entités prédéfinies. Cette limitation empêche, par exemple, d'introduire les constructeurs de PLIB permettant de représenter les documents décrivant les concepts d'une ontologie ou le constructeur `Ontology` qui permet, en OWL, de regrouper l'ensemble des concepts définis dans une ontologie (requis pour l'exigence 4). De plus, l'extension du modèle de données de RQL avec de nouveaux attributs permettant de caractériser les concepts d'une ontologie nécessite l'utilisation du constructeur de propriétés. Par exemple, si on souhaite caractériser les classes par un numéro de version, cet attribut ne peut être créé que comme une propriété RDF-Schema. En conséquence, la recherche des propriétés avec RQL retournera également cet attribut. Ainsi, la modification du niveau modèle d'ontologies modifie également automatiquement le niveau ontologie. Enfin, même si ces capacités (partielles) sont offertes par RQL, elles ne sont pas supportées ou du moins pas explicitées sur les BDBO RDF-Suite [Alexaki et al., 2001] et Sesame [Broekstra et al., 2002] sur lesquelles il a été implanté. RQL supporte donc partiellement l'exigence 4.

Exigences 5 et 6 (Compatibilité avec l'architecture traditionnelle des bases de données)

La syntaxe proposée par le langage RQL est adaptée à la structure de graphe d'un document RDF et donc elle diverge de SQL (exigence 5).

D'autre part, RQL traite chaque instance comme un URI indépendamment de ses classes d'appartenance ou de ses valeurs de propriétés. Il ne permet donc pas de manipuler explicitement la structure de

ces données (exigence 6).

Exigence 7 (Définition et manipulation des ontologies)

RQL ne propose pas de langage pour définir le modèle d'ontologies utilisé. Par contre, le langage RVL permet de créer un nouvel espace de noms dans lequel de nouvelles classes et de nouvelles propriétés peuvent être créées. Cependant, l'objectif de RVL n'étant pas de fournir un langage de manipulation des ontologies complet, il permet uniquement de caractériser les classes par un identifiant et les propriétés par un identifiant ainsi que leur domaine et leur codomaine. Les autres caractéristiques des classes et propriétés comme par exemple leurs noms dans différentes langues naturelles ou leurs descriptions ne peuvent pas être définies. RQL satisfait donc partiellement l'exigence de pouvoir définir et manipuler les ontologies d'une BDBO.

Exigence 8 (Définition et manipulation des données)

Le langage RUL permet d'ajouter, de mettre à jour et de supprimer les instances et de les caractériser par des valeurs de propriétés. RQL permet donc de manipuler les données d'une BDBO. Notons par contre que les opérateurs proposés par le langage RUL (manipulation des liens d'instanciation) sont peu habituels par rapport à l'approche proposée par les langages traditionnels de bases de données et qu'ils ont des effets de bord (par exemple, l'ajout de liens d'instanciation ou la suppression de valeurs de propriétés).

Les classes et les propriétés à partir desquelles sont définies les instances peuvent être créées avec le langage RVL. Par contre, ce langage ne permet ni de les modifier ni de les supprimer. RQL supporte donc partiellement l'exigence de pouvoir définir les données d'une BDBO.

Exigences 9 et 10 (Interrogation des ontologies et à la fois des ontologies et des données)

Au niveau du langage d'interrogation, RQL permet d'interroger les ontologies (exigence 9) mais également à la fois les ontologies et les données (exigence 10). Notons néanmoins que dans [Haase et al., 2004], les auteurs ont relevé les limitations suivantes sur le pouvoir d'expression de RQL :

- l'absence d'opérateurs permettant de manipuler les collections RDF-Schema ;
- l'absence d'opérateurs permettant de trier les résultats ;
- l'absence d'opérateurs permettant d'exprimer des expressions de chemins optionnelles, c'est-à-dire qui retourne une instance même si celle-ci ne présente pas une valeur pour une des propriétés impliquées dans une expression de chemin.

Malgré ces limitations, le langage RQL satisfait une grande partie de nos exigences en termes de pouvoir d'expression.

Exigences 11 et 12 (Implantation du langage)

La sémantique du langage RQL est définie formellement dans [Karvounarakis et al., 2004] (exigence 11). Cette base formelle a permis d'implanter ce langage sur les BDBO RDFSuite et Sesame. No-

tons de plus que le langage RQL est équipé d'une interface graphique [Athanasis et al., 2004] permettant de construire une requête visuellement. Le langage RQL satisfait donc nos exigences sur l'implantation de ce langage.

Synthèse et conclusion

Le tableau 2.2 synthétise l'analyse du langage RQL par rapport aux exigences définies.

Exigences	RQL
Expression de requête niveau ontologique	•
Définition de concepts non canoniques	•
Exploitation linguistique	-
Indépendance par rapport à un modèle d'ontologies donné	◦
Compatibilité avec SQL	-
Définition, manipulation et interrogation du schéma des données	-
Définition et manipulation des ontologies	◦
Définition et manipulation des données	◦
Interrogation des ontologies	•
Interrogation à la fois des ontologies et des données	•
Définition d'une sémantique formelle	•
Outils permettant l'exploitation du langage	•

TAB. 2.2 – Analyse du langage RQL par rapport aux exigences définies

Le langage RQL est donc un langage complet permettant la manipulation, l'interrogation et la définition de vues sur des ontologies et des données représentées en RDF-Schema. Il satisfait un grand nombre de nos exigences mais présente, par contre, les limitations suivantes :

– RQL ne permet pas de prendre en compte l'ensemble des spécificités d'un modèle d'ontologies particulier. Il propose des capacités d'extension de son modèle de données mais celles-ci sont limitées au raffinement des entités `Class` et `Property` et entraîne le mélange des différents niveaux d'une BDBO ;
– RQL ne permet pas d'exploiter la couche linguistique d'une ontologie construite selon le modèle en oignon ;
– RQL ne permet pas de manipuler les données d'une BDBO au niveau logique ;
– RQL ne propose pas un langage complet pour la définition d'ontologies.

Nous venons de montrer les limitations des langages de BDBO conçus dans le contexte du Web Sémantique. Notons que nous n'avons pas évalué de langage défini pour le modèle d'ontologies OWL. Contrairement aux langages proposés pour RDF et RDF-Schema il existe peu de langage proposé pour OWL. C'est pour cette raison que ces langages n'ont pas été inclus dans l'état de l'art établi dans [Bailey et al., 2005]. OWL-QL [Fikes et al., 2004] est un exemple de tel langage. Cependant, ce langage ne propose pas de constructeurs spécifiques à OWL et son pouvoir d'expression est inférieur à celui des

langages SPARQL et RQL. C'est pour ces raisons que nous ne l'avons pas évalué.

4 Analyse des autres langages de BDBO

Les modèles d'ontologies créés en dehors du contexte du Web Sémantique sont nombreux. Nous pouvons citer comme exemple Ontolingua [Farquhar et al., 1997], KIF [Genesereth, 1991] ou Carin [Levy and Rousset, 1998]. Des langages d'interrogation sont généralement disponibles pour permettre d'interroger des ontologies représentées selon ces modèles d'ontologies. Dans cette section, nous avons choisi de présenter les langages CQL [Mizoguchi-Shimogori et al., 2002] et SOQA-QL [Ziegler et al., 2005] qui ont été créés dans une perspective d'exploitation de bases de données.

4.1 Le langage CQL associé au modèle d'ontologies PLIB

Le langage CQL a été conçu comme une extension de SQL pour les bases de données permettant de stocker des ontologies PLIB. Il présente ainsi la particularité d'avoir été conçu dans le soucis de conserver une compatibilité avec l'architecture des bases de données traditionnelles.

4.1.1 Présentation du langage CQL

CQL [Mizoguchi-Shimogori et al., 2002] est un langage développé au sein des laboratoires de recherche de TOSHIBA pour définir, manipuler et interroger des bases de données stockant une hiérarchie de classes. Les systèmes de gestion de données PLIB (PLIB-LMS) constituent une importante application de ce langage. En conséquence, sa conception a été influencée par le modèle d'ontologies PLIB. Notre présentation et évaluation de ce langage sont basées sur la version 2.0 des spécifications de ce langage diffusées à l'adresse `http://www.toplib.com/En/aboutCQL.php`.

CQL propose un langage de définition, manipulation et interrogation de données. Le langage de définition de données permet de créer des classes conformes au modèle PLIB. Une classe est créée en indiquant un des types de classe définis en PLIB comme par exemple `Item_class`. Elle possède un identifiant unique qui s'obtient en concaténant l'identifiant du dictionnaire dans lequel elle est définie avec l'identifiant de l'organisation qui l'a créée ainsi que son code. Par exemple, `sioc_dic.sioc_org.Post` est l'identifiant de la classe dont le code est `Post`, définie par l'organisation `sioc_org` dans le dictionnaire `sioc_dic`. Pour simplifier l'écriture des exemples, nous utilisons l'expression `sioc_org:Post` comme identifiant de cette classe. La création d'une classe nécessite également d'indiquer la valeur de ses attributs. Les attributs disponibles sont similaires à ceux définis dans la norme PLIB (par exemple, `definition` ou `applicable_properties`).

Exemple. Créer une classe de type `Item_class` dont l'identifiant complet est `sioc_dic.sioc_org.Post`, dont le nom en anglais et en français est `Post`, dont la super-classe est `Item` et qui a comme propriétés applicables les propriétés `title` et `content` qui ont déjà été créées.

```
CREATE CLASS sioc_org:Post OF ITEM_CLASS (
  SUPER_CLASS sioc_org:Item,
  PREFERRED_NAME.EN 'Post',
```

```
    PREFERRED_NAME.FR 'Post',
    APPLICABLE_PROPERTIES (
      sioc_org:Post.title,
      sioc_org:Post.content )
  )
```

Explication. La classe `sioc_dic.sioc_org.Post` (identifiant raccourci `sioc_org:Post`) est créée comme sous-classe de la classe `Item` en indiquant la valeur de l'attribut `SUPER_CLASS`. La définition de la valeur de l'attribut `PREFERRED_NAME` montre que CQL supporte les définitions multi-langues. Suivant la terminologie PLIB, l'attribut `APPLICABLE_PROPERTIES` permet d'indiquer les propriétés *applicables* pour les instances de cette classe, c'est-à-dire celles qui peuvent être utilisées pour décrire ces instances. Cet exemple suppose que ces propriétés ont déjà été créées. Ces propriétés sont identifiées par rapport à leur classe de définition. Ainsi, la propriété `title` est identifiée par l'expression `sioc_org:Post.title`. Cette propriété étant définie dans le contexte de la classe `sioc_org:Post`, l'utilisation de l'identifiant complet des propriétés n'est pas nécessaire. Cependant, les auteurs de CQL ne précisent pas si le langage permet de n'utiliser que le code d'une propriété. Nous utilisons donc les identifiants complets.

Le langage de manipulation de données de CQL permet de créer des instances de classes. Pour cela, il requiert qu'une extension soit explicitement créée pour cette classe. Par exemple, l'expression suivante permet de créer une extension pour la classe `User`.

```
CREATE EXTENSION ON sioc_org:User;
```

Quand une extension a été créée, CQL permet ensuite de créer une table relationnelle pour stocker les instances de cette classe comme le montre l'exemple suivant.

Exemple. Créer une table de nom `table_post` pour stocker les instances de la classe `Post`. Cette table présente un attribut pour les propriétés `title` et `content` et la clé de cette table est l'attribut correspondant à la propriété `title`.

```
CREATE TABLE table_post ON sioc_org:Post (
  sioc_org:Post.title,
  sioc_org:Post.content,
  CONSTRAINT KEY(sioc_org:Post.content )
)
```

Explication. La création de cette table se fait en indiquant son nom (`table_post`) ainsi que la classe correspondante (`Post`). Les colonnes de cette table sont définies par les propriétés à partir desquelles elles doivent être construites. Dans cet exemple, ce sont les propriétés `title` et `content`. La clé primaire de cette table peut être identifiée avec une syntaxe différente de celle de SQL (`CONSTRAINT KEY`).

Des instances peuvent ensuite être insérées dans cette table par une syntaxe similaire à SQL.

Exemple. Insérer une instance dans la table créée précédemment.

```
INSERT INTO table_post ON sioc_org:Post
  (sioc_org:Post.title,  sioc_org:Post.content )
  VALUES ('Description de l'ontologie sioc', 'L'ontologie SIOC ...')
```

Explication. L'insertion d'une instance se fait en précisant la table dans laquelle les données sont insérées `table_post`. CQL nécessite également que la classe à laquelle cette table est associée soit indiquée (`Post`). Le nom des tables étant unique, préciser cette classe semble inutile. Les auteurs ne donnent pas d'explication à cette redondance syntaxique. Le reste de l'instruction est similaire à une instruction `INSERT` de SQL. Les propriétés valuées sont précisées avant le mot clé `VALUES` qui est suivi des valeurs de ces propriétés.

Le langage d'interrogation de CQL est décomposé en deux sous-langages : le langage d'interrogation des ontologies et le langage d'interrogation des données.

Une requête sur l'ontologie à la forme suivante :

$$\begin{array}{rl} \texttt{SELECT} & \texttt{attribute}_1, \cdots, \texttt{attribute}_n \\ \texttt{FROM} & \texttt{entity}_1, \cdots, \texttt{entity}_m \\ \texttt{WHERE} & \texttt{exp_att}_1 \ \ \texttt{OP_log}_1 \ \ \texttt{exp_att}_2 \cdots \ \ \texttt{OP_log}_k \ \ \texttt{exp_att}_{k+1} \\ \texttt{ORDER BY} & \texttt{attribute}_1, \cdots, \texttt{attribute}_n \end{array}$$

où :

- les $\texttt{attribute}_i$ sont des attributs, tels que `definition` ou `preferred_name`, définis sur les classes et propriétés ;
- les $\texttt{entity}_i$ représentent une entité du modèle d'ontologies telle que `Class`, `Property` ou `Supplier` ;
- les $\texttt{exp_att}_i$ sont des expressions booléennes sur les attributs ;
- les $\texttt{OP_log}_i$ sont des opérateurs booléens (éventuellement parenthèsés).

L'exemple suivant illustre l'utilisation de cette syntaxe.

Exemple. Retourner le domaine, le nom préféré en français ainsi que l'identifiant des propriétés dont le nom court en anglais se termine par `name`.

```
SELECT NAME_SCOPE, PREFERRED_NAME.FR, BSU_CODE
  FROM PROPERTY
 WHERE SHORT_NAME.EN LIKE '%name';
```

Explication. Cet exemple montre que la syntaxe proposée par CQL est similaire à celle de SQL. La clause `FROM` permet d'introduire un itérateur sur les propriétés (mot clé `PROPERTY`). La clause `WHERE` permet de restreindre les propriétés parcourues à celles dont le nom court en anglais (mot clé `SHORT_NAME`) se termine par `name`. Pour chaque propriété restante, la clause `SELECT` permet de retrouver son domaine (`NAME_SCOPE`), son nom préféré en français (`PREFERRED_NAME.FR`) et son identifiant (`BSU_CODE`).

Un requête sur les données à la forme suivante :

```
SELECT property_1, ···, property_n
  FROM class_table_1, ···, class_table_m
 WHERE exp_prop_1  OP_log_1  exp_prop_2 , ···,  OP_log_k  exp_prop_k+1
 ORDER BY property_1, ···, property_p
```

où :

- les $\texttt{property}_i$ sont des identifiants de propriétés. Les fonctions d'agrégats (`MIN`, `MAX`, `AVG` et `SUM`) sont disponibles. La syntaxe * peut être utilisée pour obtenir la valeur de l'ensemble des propriétés applicables sur les classes présentes dans la clause `FROM` ;
- les $\texttt{class_table}_i$ représentent des identifiants de classes suivis éventuellement du nom d'une table. Cet identifiant peut être suivi de l'opérateur * qui permet de référencer les instances de cette classe et de toutes ses sous-classes ;
- les $\texttt{exp_prop}_i$ sont des expressions booléennes sur les propriétés. Les opérateurs de quantification universelle `ALL_OF` et de quantification existentielle `SOME_OF` sont fournis pour les collections ;
- les $\texttt{OP_log}_i$ sont des opérateurs booléens (éventuellement parenthèsés).

Voici un exemple de requête sur les données.

Exemple. Retourner le contenu des messages dont le titre contient le mot `sioc`.

```
SELECT sioc_org:Post.content,
  FROM sioc_org:Post*
 WHERE sioc_org:Post.title LIKE '%sioc%'
```

Explication. La clause `FROM` de cette requête retourne les instances de la classe de code `Post` et de ses sous-classes. Une seule table étant associée à cette classe, il n'est pas nécessaire de préciser le nom de la table dans laquelle les instances sont recherchées. La clause `SELECT` retourne la valeur de la propriété `content`. La clause `WHERE` s'interprète comme pour une requête SQL classique. Elle permet donc de restreindre le résultat retourné aux messages dont la valeur de la propriété `title` contient le mot `sioc`.

En CQL, l'interrogation des ontologies et des données est ainsi clairement distinguée. Ce langage ne permet pas d'interroger conjointement les ontologies et les données d'une BDBO.

4.1.2 Analyse du langage CQL par rapport aux exigences définies

Exigences 1, 2, 3 et 4 (Exigences liées au modèle en oignon)

Le langage CQL permet d'exprimer des requêtes au niveau ontologique indépendamment des tables de la BDBO qui sont créées pour structurer ces données dans la BDBO (exigence 1). Par contre, il ne permet pas de définir des concepts non canoniques (exigence 2). L'aspect linguistique d'une ontologie est pris en compte par CQL, d'une part, pour décrire les éléments d'une ontologie et, d'autre part, pour permettre la définition de valeur d'attributs dans différentes langues naturelles. CQL satisfait donc l'exigence 3. Notons cependant qu'il ne supporte pas le multilinguisme au niveau des données.

Au niveau du modèle d'ontologies supporté, le langage CQL est lié au modèle PLIB. En effet, la majorité des noms d'attributs et des noms d'entités PLIB sont définis comme des mots clés de ce langage et, aucune modification n'est possible dans ce modèle. CQL ne satisfait donc pas l'exigence 4.

Exigences 5 et 6 (Compatibilité avec l'architecture traditionnelle des bases de données)

Le langage CQL permet de définir des tables au niveau logique d'une BDBO pour stocker les instances des classes (exigence 6). Ces instances peuvent être recherchées à partir du nom de ces tables. La syntaxe proposée pour cela est proche de celle de SQL (exigence 5).

Exigences 7 et 8 (Définition et manipulation des ontologies et des données)

Le langage CQL propose un langage de définition et de manipulation de données permettant de créer des ontologies et de leur associer des instances selon l'hypothèse de typage ontologique fort. Il répond donc à l'exigence de pouvoir définir et manipuler les données d'une BDBO. Par contre, il ne permet pas de manipuler le modèle d'ontologies représentant les ontologies créées. Il répond donc partiellement à l'exigence de pouvoir définir et manipuler les ontologies d'une BDBO.

Exigences 9 et 10 (Interrogation des ontologies et à la fois des ontologies et des données)

Le langage d'interrogation qu'il propose permet d'interroger des ontologies (exigence 9). Par contre, une distinction claire est faite entre l'interrogation des ontologies et l'interrogation des données. CQL ne permet ainsi pas d'effectuer des requêtes à la fois sur les ontologies et les données (exigence 10).

Exigences 11 et 12 (Implantation du langage)

Niveau implantation, à notre connaissance, aucune sémantique formelle n'a été associée au langage CQL (exigence 11). Par contre, la suite d'outils TopLib[13] a été développée pour faciliter et exploiter ce langage (exigence 12).

Synthèse et conclusion

Le tableau 2.3 synthétise l'analyse du langage CQL par rapport aux exigences définies.

Ainsi, le langage CQL présente la particularité de conserver un degré de compatibilité avec les bases de données traditionnelles. Il présente cependant les limitations principales suivantes par rapport à nos exigences :

- être lié au modèle PLIB. Le langage encode les différents éléments de ce modèle dans sa grammaire ;
- séparer l'interrogation de l'ontologie et des données en deux langages distincts qui ne peuvent pas être combinés ;
- ne pas permettre la définition de concepts non canoniques ;

[13] `http://www.toplib.com/`

Exigences	CQL
Expression de requête niveau ontologique	•
Définition de concepts non canoniques	-
Exploitation linguistique	•
Indépendance par rapport à un modèle d'ontologies donné	-
Compatibilité avec SQL	•
Définition, manipulation et interrogation du schéma des données	•
Définition et manipulation des ontologies	◦
Définition et manipulation des données	•
Interrogation des ontologies	•
Interrogation à la fois des ontologies et des données	-
Définition d'une sémantique formelle	-
Outils permettant l'exploitation du langage	•

TAB. 2.3 – Analyse du langage CQL par rapport aux exigences définies

– ne pas être fondé sur une sémantique formelle. Seule la syntaxe de CQL est définie dans ces spécifications, ce qui rend difficile de déterminer la sémantique des énoncés de ce langage.

Lié au modèle PLIB, le langage CQL ne satisfait pas nos exigences. Dans la section suivante nous évaluons le langage SOQA-QL conçu pour être indépendant d'un modèle d'ontologies particulier.

4.2 Le langage SOQA-QL indépendant d'un modèle d'ontologies particulier

Le second langage que nous présentons est le langage SOQA-QL conçu dans le contexte de l'intégration de données.

4.2.1 Présentation du langage SOQA-QL

Le langage SOQA-QL [Ziegler et al., 2005] a été conçu dans le contexte du projet SIRUP visant à permettre l'intégration de sources de données hétérogènes en fonction des besoins utilisateurs. Il permet d'interroger les ontologies et les données qu'elles décrivent indépendamment du modèle d'ontologies utilisé. L'approche suivie est basée sur la définition d'un modèle nommé *SOQA Ontology Meta Model* contenant les constructeurs importants de différents modèles d'ontologies. Les différents constructeurs de ce modèle d'ontologies sont les suivants.

– Le constructeur d'ontologies (`Ontology`). Ce constructeur permet de définir une ontologie avec un nom, un auteur, une documentation et un numéro de version.
– Le constructeur de classes (`Concept`). Ce constructeur permet de définir une classe avec un nom, une documentation et une définition. Il permet d'organiser les classes dans une hiérarchie utilisant la relation de subsomption. Une classe a donc des super-classes et sous-classes directes et indirectes. Une classe est également décrite par les nouvelles propriétés et méthodes dont elle constitue le domaine ainsi que les relations dont elle fait partie.

– Le constructeur de propriétés (`Attribute`). Ce constructeur permet de définir une propriété avec un nom, une documentation, une définition, son type de données et sa classe de définition.

– Le constructeur de méthodes (`Method`). Une méthode retourne une valeur à partir de valeurs de paramètres. Ce constructeur permet de définir une méthode avec une classe de définition, un nom, une documentation, une définition, ses paramètres, son type de retour et sa classe de définition.

– Le constructeur de relation (`Relationship`). Une relation permet d'établir un lien entre différentes classes. Ce constructeur permet de définir une relation avec un nom, une documentation, une définition, son arité et les classes liées par cette relation.

– Le constructeur d'instances (`Instance`). Ce constructeur permet de définir des instances avec un nom, la classe dont elle est membre[14] et ses valeurs de propriétés.

Ce modèle d'ontologies a été utilisé pour définir une interface fonctionnelle d'accès à liaison préalable (*SIRUP Ontology Query API*). Cette API est constituée d'un ensemble de primitives permettant de récupérer les éléments d'une ontologie conforme à ce modèle.

Voici trois méthodes de cette API :

```
(M1) public Collection<Concept> getConcepts()
(M2) public Collection<Concept> getDirectSuperConcepts(String conceptName)
(M3) public Collection<Instance> getInstances(String conceptName)
```

La méthode `M1` permet de récupérer l'ensemble des classes (nommées concepts dans cette API) définies dans l'ontologie manipulée. La méthode `M2` permet de récupérer l'ensemble des super-classes directes d'une classe dont le nom est passé en paramètre de la méthode. Enfin, la méthode `M3` permet de récupérer l'ensemble des instances d'une classe dont le nom est passé en paramètre de la méthode.

Cette API constitue la base du langage SOQA-QL. Ce langage a en effet été défini en partant de SQL et en l'étendant pour qu'il puisse offrir les mêmes capacités que la SIRUP Ontology Query API. La syntaxe d'une requête SOQA-QL pour interroger les ontologies est la suivante :

SELECT attribute$_1$, $\cdots$, attribute$_n$
FROM metaModelElementSet$_1$, $\cdots$, metaModelElementSet$_m$
WHERE att_exp$_1$ OP_log$_1$ att_exp$_2$ $\cdots$ OP_log$_k$ att_exp$_{k+1}$
ORDER BY attribute$_1$, $\cdot$, attribute$_p$

Chaque élément de la clause `FROM` (`metaModelElementSet`) est une collection (qui peut être réduite à un élément) de classes, propriétés, méthodes ou relations. Les éléments de la clause `SELECT` (`proj`) sont des attributs du modèle d'ontologies (par exemple `documentation` ou `name`). Les clauses `WHERE` et `ORDER BY` permettent comme en SQL de filtrer les éléments spécifiés dans la clause `FROM` et de trier le résultat. L'exemple suivant illustre l'utilisation de cette syntaxe.

Exemple. Rechercher la documentation associée à la classe `Post` de l'ontologie dont l'espace de noms a pour alias `sioc`.

```
SELECT DOCUMENTATION FROM sioc:Post
```

Explication. Dans cet exemple, une seule classe est spécifiée dans la clause `FROM`. SOQA-QL traite cette

[14]Même si les auteurs ne l'explicitent pas, il semble que seule la mono-instanciation soit prise en compte.

classe (Post) comme une collection réduite à un élément. La clause SELECT permet de projeter cette classe sur l'attribut DOCUMENTATION afin de retrouver la documentation la décrivant.

Afin d'intégrer les capacités de la SIRUP Ontology Query API, SOQA-QL est équipé d'une fonction pour chaque primitive de cette API. L'exemple suivant montre comment SOQA-QL permet d'intégrer les capacités de recherche des super-classes directes d'une classe (méthode M2 présentée précédemment).

Exemple. Rechercher la valeur des attributs (nom, documentation, etc.) décrivant les super-classes directes de la classe Post.

```
SELECT * FROM DIRECTSUPERCONCEPTS(sioc:Post)
```

Explication. Conformément à sa signature dans la SIRUP Ontology Query API, la fonction DIRECTSUPERCONCEPTS appliquée sur la classe Post retourne la collection des super-classes directes de cette classe. La clause SELECT permet de projeter cet ensemble de classes sur l'ensemble des attributs du modèle d'ontologies SOQA définis pour décrire les classes (syntaxe *).

Dans la clause SELECT, SOQA-QL ne permet d'utiliser que des attributs du modèle d'ontologies et pas de propriétés d'une classe donnée. Ainsi, le langage SOQA-QL a été essentiellement défini pour interroger les ontologies. Afin de permettre de retrouver tout de même les valeurs de propriétés des instances, SOQA-QL fournit les fonctions INSTANCES et VALUE. INSTANCES retourne l'ensemble des instances d'une ou plusieurs classes prises en paramètre. VALUE retourne la valeur d'une propriété prise en paramètre pour une instance donnée. L'utilisation de ces fonctions est illustrée dans l'exemple suivant.

Exemple. Rechercher la valeur de la propriété email pour les instances des sous-classes de la classe User.

```
SELECT VALUE(sioc:email)
  FROM INSTANCES(SUBCONCEPTS(sioc:User))
```

Explication. Dans la clause FROM l'appel de la fonction SUBCONCEPTS permet de retrouver les sous-classes de la classe User. La fonction INSTANCES appliquée à cette collection de classes retourne l'ensemble des instances de ces classes. La fonction VALUE utilisée dans la clause SELECT permet de projeter ces instances sur la propriété EMAIL. Comme le montre cette requête, SOQA-QL permet d'interroger à la fois l'ontologie (SUBCONCEPTS) et les données (VALUE).

4.2.2 Analyse du langage SOQA-QL par rapport aux exigences définies

Exigences 1, 2, 3 et 4 (Exigences liées au modèle en oignon)

Le langage SOQA-QL permet d'exprimer des requêtes au niveau ontologique indépendamment de la structure des données dans la BDBO (exigence 1). Par contre, il ne permet pas la définition de concepts non canoniques (exigence 2). L'aspect linguistique d'une ontologie est pris en compte dans le modèle d'ontologies SOQA (nom, documentation, définition, etc.) et donc dans le langage SOQA-QL. Cependant, le multilinguisme n'est pas supporté. SOQA-QL supporte donc partiellement l'exigence 3. Enfin, le langage SOQA-QL est le premier langage ayant été conçu indépendamment d'un modèle d'ontologies particulier (exigence 4). Notons cependant que cette capacité est limitée aux constructeurs fournis dans le modèle d'ontologies noyau défini pour ce langage.

Exigences 5 et 6 (Compatibilité avec l'architecture traditionnelle des bases de données)

SOQA-QL est un des rares langages d'ontologies dont la syntaxe étend celle de SQL (exigence 5). Il permet ainsi d'effectuer la projection des entités définies dans le modèle d'ontologies sur des attributs. Les requêtes sur les ontologies se présentent donc comme une requête SQL classique. Notons par contre que la syntaxe proposée pour l'interrogation des données diffère de celle de SQL. Pour retourner la valeur d'une propriété pour les instances d'une classe, il est nécessaire d'utiliser les fonctions `INSTANCES` et `VALUE`. Le langage SOQA-QL satisfait donc partiellement l'exigence 5.

En SOQA-QL, les instances sont associées directement aux classes sans la notion de schéma (niveau logique). Il ne permet ainsi pas de manipuler explicitement la structure de ces données (exigence 6).

Exigences 7 , 8, 9 et 10 (Exigences en termes de pouvoir d'expression)

Le langage SOQA-QL ne propose pas de langage de définition et de manipulation de données (exigence 8) . Il ne propose également pas de langage pour définir et manipuler les ontologies d'une BDBO (exigence 7). Ainsi, il ne permet pas de manipuler le modèle d'ontologies utilisé. En conséquence les particularités d'un modèle d'ontologies donné ne peuvent pas être prises en compte par ce langage. Par exemple, le modèle d'ontologies PLIB permet de décrire les classes et les propriétés d'une ontologie par une remarque, une note, une illustration ainsi que de nombreux autres attributs. La valeur de ces attributs ne peut pas être récupérée en utilisant SOQA-QL. De même, OWL permet de définir des restrictions sur des propriétés. SOQA-QL ne permet pas de récupérer les restrictions portant sur une propriété donnée.

Par contre, SOQA-QL propose un langage d'interrogation qui permet d'interroger les ontologies et à la fois les ontologies et les données (exigences 9 et 10) en utilisant les fonctions `INSTANCES` et `VALUE`.

Exigences 11 et 12 (Implantation du langage)

Au niveau implantation, la sémantique du langage SOQA-QL est définie en termes de la SIRUP Ontology Query API qui constitue l'algèbre de ce langage (exigence 11). Cette API a été implantée pour différents modèles d'ontologies. A notre connaissance ces implantations n'ont pas été réalisées sur des BDBO mais sur des fichiers. Enfin, ce langage est équipé d'un outil permettant de l'utiliser en ligne de commande (exigence 12). Le langage SOQA-QL satisfait donc nos exigences sur l'implantation de ce langage.

Synthèse et conclusion

Le tableau 2.4 synthétise l'analyse du langage SOQA-QL par rapport aux exigences définies.

L'innovation du langage SOQA-QL est d'être fondé sur un modèle d'ontologies noyau contenant les constructeurs fondamentaux de nombreux modèles d'ontologies. Les limitations principales de ce langage par rapport à nos exigences sont les suivantes :

- le modèle d'ontologies noyau est figé. En conséquence, les spécificités de certains modèles d'ontologies ne peuvent pas être prises en compte par le langage SOQA-QL ;
- les couches OCNC et OL d'une ontologie ne sont pas supportées par le langage SOQA-QL ;
- SOQA-QL ne propose pas de langage de définition et de manipulation de données.

Exigences	**SOQA-QL**
Expression de requête niveau ontologique	•
Définition de concepts non canoniques	-
Exploitation linguistique	◦
Indépendance par rapport à un modèle d'ontologies donné	•
Compatibilité avec SQL	◦
Définition, manipulation et interrogation du schéma des données	-
Définition et manipulation des ontologies	-
Définition et manipulation des données	-
Interrogation des ontologies	•
Interrogation à la fois des ontologies et des données	•
Définition d'une sémantique formelle	•
Outils permettant l'exploitation du langage	•

TAB. 2.4 – Analyse du langage SOQA-QL par rapport aux exigences définies

5 Conclusion

Dans ce chapitre, nous avons tout d'abord analysé l'architecture de BDBO proposée en termes d'exigences. Les principales exigences établies résultent d'une part de l'ajout du niveau ontologique dans l'architecture ANSI/SPARC décomposée selon les trois couches du modèle en oignon. Elles résultent, d'autre part, du fait que notre architecture est compatible avec celle des SGBD existants et donc, conserve le traitement des données au niveau logique.

Nous avons ensuite analysé différentes catégories de langages proposées dans la littérature par rapport à ces exigences. Nous avons vu dans un premier temps que les langages de BDBO conçus pour le Web Sémantique (SPARQL et RQL) ne présentent pas de compatibilité avec l'architecture usuelle des bases de données. Ils ne permettent ainsi pas de manipuler les données d'une BDBO au niveau logique et propose une syntaxe et une sémantique différentes de SQL. De plus, les langages construits pour RDF tels que SPARQL sont dépendants de l'interprétation des triplets représentés dans une BDBO. Quant au langage RQL, défini pour RDF-Schema, il présente des capacités limitées pour prendre en compte les spécificités introduites par certains modèles d'ontologies et n'exploite pas les définitions linguistiques d'une ontologie.

Dans un second temps, nous avons étudié les langages CQL et SOQA-QL qui ont été conçus dans un autre contexte que le Web Sémantique. Nous avons montré que CQL ne satisfait pas nos exigences parce qu'il est lié à un modèle d'ontologies et qu'il ne permet pas de manipuler ce dernier. Concernant le langage SOQA-QL nous avons vu qu'il est indépendant d'un modèle d'ontologies particulier mais qu'il ne permet pas, par contre, de prendre en compte les spécificités d'un modèle d'ontologies particulier. De plus, aucun de ces deux langages ne permet de manipuler des classes non canoniques.

Même si ces langages ne satisfont pas les exigences définies, ils proposent des approches permettant d'en satisfaire une partie. Nous retenons notamment les approches suivantes proposées par ces langages :

- construire le langage sur la base d'un modèle d'ontologies noyau contenant les constructeurs partagés par différents modèles d'ontologies proposés dans la littérature pour que le langage soit indépendant d'un modèle d'ontologies particulier (approche proposée par SOQA-QL) ;
- construire le langage à partir de SQL pour permettre la manipulation des données au niveau logique (approche proposée par CQL) ;
- ajouter des opérateurs permettant de rechercher la valeur d'attributs et/ou de propriétés dans plusieurs langues naturelles (approche proposée par SPARQL) ;
- utiliser le mécanisme de vue des bases de données pour permettre de définir des concepts non canoniques (approche proposée par RQL).

Ayant établi qu'il n'existe pas de langage satisfaisant l'ensemble des exigences que nous avons établies, nous proposons un nouveau langage nommé OntoQL. La conception de ce langage est inspirée des approches que nous venons d'énoncer. Nous présentons ce langage dans la partie suivante.

Deuxième partie

Notre proposition : le langage OntoQL

Chapitre 3

Traitements des données à base ontologique d'une BDBO

Sommaire

1 **Introduction** **93**
2 **Exploitation des données à base ontologique au niveau logique** **94**
2.1 Modèle de données du niveau logique 94
2.2 Langage de définition, de manipulation et d'interrogation de données . . . 96
2.3 Utilisation 99
3 **Exploitation des données à base ontologique au niveau ontologique. Application à la couche OCC** **100**
3.1 Modèle de données du niveau ontologique, couche OCC 100
3.2 Aspects syntaxiques 104
3.3 Langage de Définition de Données (LDD) 106
3.4 Langage de Manipulation de Données (LMD) 111
3.5 Langage d'Interrogation de Données (LID) 112
4 **Exploitation des données à base ontologique au niveau ontologique. Application à la couche OCNC** **119**
4.1 Problématique 119
4.2 Langage de Définition de Vues (LDV) 120
4.3 Utilisation 123
5 **Exploitation des données à base ontologique au niveau ontologique. Application à la couche OL** **127**
5.1 Modèle de données du niveau ontologique, couche OL 127
5.2 Aspects syntaxiques 128
5.3 Langage de définition, de manipulation et d'interrogation de données . . . 129
6 **Conclusion** **131**

Résumé. Dans le chapitre précédent, nous avons défini les exigences de conception d'un langage d'exploitation de l'architecture ANSI/SPARC étendue proposée et nous avons vu que nous ne connaissions pas de langage qui y réponde de manière satisfaisante. Nous présentons à présent notre proposition de langage, nommé OntoQL, conçu pour répondre à ces

différentes exigences [Jean et al., 2006a, Jean et al., 2006b]. Dans ce chapitre, nous mettons l'accent sur les traitements des données à base ontologique que permet de réaliser le langage OntoQL lorsqu'un utilisateur connaît le modèle logique des données et/ou les ontologies qui les décrivent. Les données à base ontologique devant pouvoir être manipulées tant au niveau logique de l'architecture ANSI/SPARC étendue qu'au niveau ontologique suivant les trois catégories du modèle en oignon (OCC, OCNC, OL), nous avons choisi de concevoir le langage OntoQL en couches, chacune correspondant à un des niveaux d'accès nécessaires. Chaque couche est définie par un modèle de données ainsi que par les opérateurs du langage permettant de manipuler des données conformes à ce modèle. Pour définir ces modèles de données et ces opérateurs, nous avons cherché à rester le plus proche possible du langage SQL afin que langage OntoQL permette un accès homogène aux différents niveaux d'accès proposés. Le résultat est un langage qui (1) est compatible avec un sous-ensemble de SQL pour permettre de manipuler les données à base ontologique au niveau logique, comme dans une base de données traditionnelle et (2) adapte et étend ce sous-ensemble de SQL pour permettre de traiter les données à base ontologique à partir du niveau ontologique selon les trois catégories du modèle en oignon.

1 Introduction

Dans le chapitre précédent, nous avons justifié la nécessité d'un nouveau langage de BDBO en définissant les exigences pour l'architecture ANSI/SPARC étendue proposée. Pour répondre à ce besoin d'un nouveau langage, nous avons conçu le langage OntoQL [Jean et al., 2006a, Jean et al., 2006b]. Plutôt que de présenter ce langage globalement, en identifiant ses sous-langages de définition, manipulation et interrogation de données, nous avons choisi de le décrire selon deux familles de traitements et de fonctions qu'il offre : les traitements des données à base ontologique (présentés dans ce chapitre) et les traitements des ontologies et à la fois des ontologies et des données (présentés dans le chapitre suivant).

La composante du langage OntoQL décrite dans ce chapitre est sa capacité d'exploiter les données à base ontologique selon les différents niveaux de l'architecture ANSI/SPARC étendue. Ces traitements supposent que l'utilisateur connaisse le modèle logique des données à base ontologique (hypothèse également faite dans les bases de données traditionnelles) et/ou les ontologies qui en décrivent le sens. Les traitements présentés dans ce chapitre sont ainsi similaires à ceux proposés dans les bases de données traditionnelles sauf qu'ils permettent, en plus, d'accéder aux données d'une base de données indépendamment de son modèle logique ; et cela, à partir du niveau ontologique correspondant à chacune des couches du modèle en oignon.

Compte tenu des différents niveaux d'accès nécessaires aux données à base ontologique, nous avons conçu le langage OntoQL en couches. Chaque couche permet d'exploiter les données à base ontologique suivant un des niveaux de l'architecture ANSI/SPARC étendue. Ces couches sont les suivantes :

- la couche d'accès aux données à base ontologique au niveau logique ;
- la couche d'accès aux données à base ontologique au niveau OCC dans le modèle en oignon ;
- la couche d'accès aux données à base ontologique au niveau OCNC dans le modèle en oignon ;
- la couche d'accès aux données à base ontologique au niveau OL dans le modèle en oignon.

Le travail de définition d'un langage nécessitant à la fois de définir le modèle de données considéré, composé de structures de données et des constructeurs associés, ainsi que les opérateurs permettant de manipuler des données conformes à ce modèle, il a été mené pour chacune de ces couches.

Pour que le langage OntoQL conserve une compatibilité ascendante avec le langage SQL (exigence 5) et propose une syntaxe uniforme pour exploiter les données à base ontologique aux différents niveaux de l'architecture ANSI/SPARC étendue, nous nous sommes basés sur le modèle de données et les opérateurs de SQL pour définir les différentes couches d'accès offertes par le langage. Le résultat est un langage qui (1) est compatible avec un sous-ensemble de SQL pour permettre de manipuler les données à base ontologique au niveau logique, comme dans une base de données traditionnelle et (2) adapte et étend ce sous-ensemble de SQL pour permettre de traiter les données à base ontologique au niveau ontologique selon les trois catégories du modèle en oignon.

Ce chapitre est organisé comme suit. La section 2 présente la couche d'accès aux données à base ontologique au niveau logique en présentant le sous-ensemble de SQL avec lequel le langage OntoQL est compatible. Les trois sections suivantes sont consacrées aux couches d'accès aux données à base ontologique au niveau ontologique. Nous présentons d'abord, dans la section 3, la couche d'accès aux données à partir des concepts canoniques d'une ontologie. Puis, nous montrons dans la section 4 comment les concepts non canoniques peuvent être représentés et utilisés pour accéder aux données à base

ontologique en utilisant le mécanisme de vue des bases de données. Enfin, la section 5 présente l'accès aux données à base ontologique en utilisant la couche linguistique d'une ontologie qui permet d'exprimer des requêtes en utilisant les noms associés aux concepts d'une ontologie. Ce chapitre est conclu à la section 6 en synthétisant les possibilités offertes par le langage OntoQL pour manipuler les données à base ontologique.

2 Exploitation des données à base ontologique au niveau logique

Le langage OntoQL doit permettre d'effectuer des traitements sur les données au niveau logique en étant compatible avec le langage SQL (exigence 5). Cependant, la norme SQL actuelle nommée SQL2003 [Eisenberg et al., 2004] est une norme complexe qu'aucun SGBD actuel ne supporte complètement. Souhaitant que le langage OntoQL reste simple à appréhender, nous avons choisi de le rendre compatible avec un sous-ensemble de cette norme. Ainsi, le modèle de données du niveau logique de l'architecture ANSI/SPARC étendue est un sous-ensemble du modèle relationnel-objet (cf. section 2.1) ; et, les langages de définition, de manipulation et d'interrogation du langage OntoQL permettant d'exploiter les données au niveau logique sont des sous-ensembles de ceux proposés par le langage SQL (cf. section 2.2).

2.1 Modèle de données du niveau logique

Pour définir le sous-ensemble du modèle relationnel-objet constituant le modèle de données du niveau logique de l'architecture ANSI/SPARC étendue, nous avons sélectionné les principaux éléments du modèle relationnel-objet communément supportés par les SGBD. Vu la multitude de SGBD existants, nous avons sélectionné les SGBD que nous avons considérés comme représentatifs de ceux couramment utilisés. Comme SGBD open source, nous avons choisi PostgreSQL 8.2[15] et MySQL 5.0[16] et comme SGBD commerciaux, nous avons choisi Oracle 10g[17], DB2 9[18] et SQL Server 2005[19]. Notons que de nombreux autres SGBD auraient pu être choisis comme par exemple Sybase ou Firebird.

2.1.1 Implantation du modèle de données relationnel-objet dans les SGBD usuels

Le tableau 3.1 présente les résultats d'une étude comparative sur le support des principaux constructeurs associés au modèle de données défini dans la norme SQL92 par les SGBD considérés. Suivant les notations utilisées précédemment, le symbole • indique qu'un constructeur est supporté. Lorsque ce n'est pas le cas, le symbole - est utilisé. Ce tableau montre qu'actuellement les SGBD supportent les principaux constructeurs de ce modèle de données. Ainsi, la majorité des SGBD permettent de créer des tables relationnelles composées d'attributs de type primitif. Seul le type booléen est peu supporté. Dans les SGBD ne le supportant pas, il peut cependant être simulé avec un type entier ou énuméré. D'autre

[15] `http://www.postgresql.org/docs/8.2/static/index.html`
[16] `http://dev.mysql.com/doc/refman/5.0/fr/index.html`
[17] `http://www.oracle.com/pls/db10g/` (SQL Reference)
[18] `http://www-1.ibm.com/support/docview.wss?rs=71&uid=swg27009552` (SQL Reference)
[19] `http://msdn2.microsoft.com/fr-fr/library/ms130214.aspx`

Constructeurs	PostgreSQL	MySQL	Oracle	DB2	SQL Server
Table	•	•	•	•	•
Vue	•	•	•	•	•
Types primitifs					
_chaînes de caractères	•	•	•	•	•
_numériques exactes	•	•	•	•	•
_numériques approchés	•	•	•	•	•
_booléen	•	•	-	-	-
_date/heure	•	•	•	•	•
Contraintes					
_unicité	•	•	•	•	•
_référentielles	•	•	•	•	•
_générales (*check*)	•	-	•	•	•

TAB. 3.1 – Support du modèle de données de la norme SQL92 par les SGBD usuels

part, afin de maintenir la cohérence des données, les SGBD permettent de définir des contraintes d'intégrité sur ces tables. Les SGBD permettent de définir des contraintes d'unicité (clé primaire et mot clé `UNIQUE`), référentielles (clés étrangères) et, mis à part MySQL, générales (mot clé `CHECK` suivi d'une expression logique). En MySQL, ces contraintes peuvent cependant être représentées via des déclencheurs (*trigger*). Enfin, les SGDB permettent de définir des vues à partir de ces tables pour permettre notamment à chaque utilisateur de représenter sa propre perception de la base de données.

Par contre, comme le montre le tableau 3.2, les SGBD actuels supportent peu les extensions du modèle relationnel proposées dans les normes SQL99 et SQL2003. Ainsi, peu de SGBD permettent de définir des *types utilisateurs* (*User-Defined types*). En particulier, ce n'est pas le cas du SGBD MySQL et ce n'est possible en PostgreSQL et SQL Server qu'en les implantant en dehors du SGBD. En conséquence, peu de SGBD permettent de définir des *tables typées*, c'est-à-dire créées à partir d'un type utilisateur. De même, les *références d'objets* ainsi que l'*héritage de type et de table* très liés au concept de type utilisateur sont peu supportés. Enfin, peu de SGBD permettent d'utiliser les types collections.

Constructeurs	PostgreSQL	MySQL	Oracle	DB2	SQL Server
Type utilisateur	-	-	•	•	-
Héritage de type	-		•	•	-
Table typée	-	-	-	•	-
Héritage de table	•	-	-	•	-
Référence d'objet	•	-	•	•	-
Type collection					
_array	•	-	•	-	-
_multiset	-	-	•	-	-

TAB. 3.2 – Support du modèle de données de la norme SQL2003 par les SGBD usuels

2.1.2 Implantation du modèle de données de SQL au niveau logique de l'architecture ANSI/SPARC étendue

A partir de l'étude du support du modèle de données de SQL par les principaux SGBD, nous avons décidé d'inclure dans le modèle de données du niveau logique de l'architecture ANSI/SPARC étendue les principaux éléments du modèle relationnel défini dans la norme SQL92. Ainsi, comme le montre le tableau 3.3, ce modèle comprend des tables (ou relations) définies à partir des types primitifs : chaîne de caractères, nombres, booléens et date. A ces tables, des contraintes telles que les clés primaires, étrangères et générales peuvent être associées. Enfin, à partir des tables, des vues peuvent être construites. Notons que nous avons choisi d'inclure dans ce modèle de données les types booléens et les contraintes générales (`CHECK`) même si ils ne sont pas supportés par tous les SGBD. Nous avons fait ce choix car, comme nous l'avons indiqué, des représentations alternatives sont possibles pour ces éléments dans les SGBD ne les supportant pas. Par contre, nous n'avons pas inclus les constructeurs orientés-objets de la norme SQL99, c'est-à-dire les types utilisateurs et les différents éléments liés à ce constructeur (table typée, héritage de type et de table ainsi que les références d'objets).

Constructeurs	Modèle de données du niveau logique
Table	•
Vue	•
Types primitifs	•
Contraintes	•
Type utilisateur	-
Héritage de type	-
Table typée	-
Héritage de table	-
Référence d'objet	-
Type collection	-

Tab. 3.3 – Constructeurs du modèle de données de l'architecture ANSI/SPARC étendue au niveau logique

Le modèle de données du niveau logique de l'architecture ANSI/SPARC étendue étant précisé, nous indiquons maintenant le sous-ensemble du langage SQL supporté par le langage OntoQL pour manipuler des données au niveau logique.

2.2 Langage de définition, de manipulation et d'interrogation de données

Pour déterminer les opérateurs supportés par OntoQL au niveau logique, nous avons étudié le support des opérateurs de SQL par les SGBD usuels.

2.2.1 Implantation des opérateurs de SQL dans les SGBD usuels

Le tableau 3.4 présente les résultats d'une étude comparative sur le support des principaux opérateurs relationnels par les SGBD considérés. Il montre que les SGBD usuels supportent les principaux

Opérateurs	**PostgreSQL**	**MySQL**	**Oracle**	**DB2**	**SQL Server**
Fonctions prédéfinies	•	•	•	•	•
Fonctions de calculs	•	•	•	•	•
Sous-requêtes					
_clause `FROM`	•	•	•	•	•
_clause `WHERE`	•	•	•	•	•
Quantificateurs	•	•	•	•	•
Opérateurs ensemblistes					
_`UNION`	•	•	•	•	•
_`INTERSECT`	•	-	•	•	•
_`EXCEPT`	•	-	•	•	•
Jointures					
_interne	•	•	•	•	•
_externe	•	•	•	•	•
_naturelle	•	•	•	-	-
Requêtes récursives	-	-	•	•	•

Tab. 3.4 – Support des principaux opérateurs relationnels par les SGBD usuels

opérateurs définis sur le modèle relationnel. Ainsi, ils permettent d'utiliser les fonctionnalités suivantes :

– les fonctions prédéfinies comme par exemple `UPPER` qui permet de transformer une chaîne de caractères en majuscules ;
– les fonctions de calcul (`AVG`, `MIN`, `MAX`, `SUM` et `COUNT`) et d'agrégats (clauses `GROUP BY` et `HAVING`) ;
– l'expression de sous-requêtes (*requêtes imbriquées*) dans la clause `WHERE` et dans la clause `FROM` ;
– les quantificateurs (`ANY`, `ALL` et `EXISTS`) ;
– les opérateurs ensemblistes (`UNION`, `INTERSECT` et `EXCEPT`). MySQL ne supporte actuellement pas les opérateurs `INTERSECT` et `EXCEPT` mais ils peuvent être remplacés par des jointures [20] ;
– les jointures internes, externes et naturelles. DB2 et SQL Server ne supportent pas les jointures naturelles mais elles peuvent être remplacées par des jointures internes en utilisant les informations sur le schéma contenu dans la métabase.

Par contre, l'opérateur permettant d'exprimer des requêtes récursives (`WITH RECURSIVE`), introduit dans la norme SQL99, n'est pas supporté par PostgreSQL et MySQL. De plus, l'implantation de cet opérateur avec les opérateurs disponibles ne peut pas se faire de façon simple.

Considérons maintenant les opérateurs relationnels-objets introduits dans la norme SQL99. Ces opérateurs ont été définis pour permettre d'exploiter les constructeurs du modèle relationnel-objet introduits dans la norme SQL99. Ces opérateurs sont présentés dans le tableau 3.5. Ainsi, SQL propose des opérateurs pour manipuler les types utilisateurs. Ils permettent d'appeler les méthodes qui peuvent accompagner la définition d'un tel type, de tester si un objet appartient à un type utilisateur et de convertir le type

[20] cf. `http://infolab.stanford.edu/~ullman/fcdb/oracle/my-nonstandard.html`

Constructeurs relationnels-objets	Opérateurs correspondants
Type utilisateur	Appel de méthode Test du type d'une objet (`IS OF`) Transtypage (*cast*) d'un objet (`TREAT AS`)
Héritage de table	Requête polymorphe (`ONLY`) Mise à jour polymorphe (`ONLY`)
Référence d'objets	Expression de chemin Récupération de l'identifiant d'un objet Récupération d'un objet à partir de son identifiant (`DEREF`)
Type collection	Constructeur de collection à partir d'une requête Accès à un élément d'une collection indexée (`[]`) Test d'appartenance d'un élément à une collection (`= ANY`) Dégroupage d'une collection (`UNNEST`)

Tab. 3.5 – Principaux opérateurs correspondant aux constructeurs relationnels-objets introduits dans la norme SQL99

d'un objet. L'héritage de table peut être exploité en utilisant des opérateurs permettant de réaliser des requêtes ou des mises à jour qui portent non seulement sur les instances d'une table mais aussi sur ces sous-tables (*mises à jour et requêtes polymorphes*). Concernant les références d'objets, des opérateurs permettent d'exprimer des expressions de chemins et d'obtenir un objet à partir de son identifiant ou vice versa. Enfin, les opérateurs définis sur les collections permettent d'en construire une à partir d'une requête retournant une relation à une seule colonne, d'en manipuler les éléments ou de les utiliser dans la clause `FROM` d'une requête (*dégroupage*).

Comme nous l'avons vu dans la section 2.1.1, les SGBD considérés supportent peu les constructeurs du modèle relationnel-objet introduits dans la norme SQL99. En conséquence, les opérateurs correspondant à ces constructeurs présentés dans le tableau 3.5 sont également peu supportés par ces SGBD.

2.2.2 Implantation des opérateurs de SQL par OntoQL au niveau logique

Pour définir les opérateurs de SQL supportés par OntoQL, nous avons commencé par écarter les opérateurs correspondant aux constructeurs du modèle de données que nous avons choisi de ne pas inclure dans le modèle de données du niveau logique de l'architecture ANSI/SPARC étendue. N'ayant pas inclus les constructeurs du modèle relationnel-objet introduits dans la norme SQL99, les opérateurs correspondants (cf. tableau 3.5) ne sont pas supportés par les langages de définition, de manipulation et d'interrogation de données de OntoQL au niveau logique. Par contre, nous verrons que cette possibilité est offerte par les accès au niveau ontologique.

Une fois les opérateurs associés aux constructeurs non supportés dans le modèle de données du niveau logique écartés, nous avons sélectionné les principaux opérateurs restant dans la norme SQL communément supportés par les SGBD que nous avons choisis comme référence. Nous avons vu dans la section précédente que, mis à part l'opérateur permettant d'exprimer des requêtes récursives, les principaux opérateurs relationnels sont supportés par ces SGBD.

Nous avons donc décidé que les langages de définition, de manipulation et d'interrogation de données de OntoQL au niveau logique supportent ces opérateurs. Pour récapituler, le tableau 3.6 indique les opérateurs de SQL supportés par, et ceux absents du langage OntoQL au niveau logique.

Opérateurs	**Langage OntoQL au niveau logique**
Fonctions prédéfinies	•
Fonctions de calculs	•
Sous-requêtes	•
Quantificateurs	•
Opérateurs ensemblistes	•
Jointures	•
Requêtes récursives	-
Opérateurs sur les types utilisateurs	-
Opérateurs sur la hiérarchie de tables	-
Opérateurs sur les références d'objets	-
Opérateurs sur les collections	-

TAB. 3.6 – Opérateurs de SQL disponibles dans le langage OntoQL au niveau logique

Nous venons de préciser les opérateurs associés aux langages de définition, de manipulation et d'interrogation du langage OntoQL au niveau logique. Rappelons que nous avons défini ce niveau d'accès afin de fournir un seul et même langage pour accéder de façon homogène aux différents niveaux d'une BDBO et afin de garder une compatibilité avec SQL. Dans la définition des exigences, nous avons donné quelques éléments sur l'intérêt d'accéder aux données depuis ce niveau. Nous détaillons ces éléments dans la section suivante.

2.3 Utilisation

Pouvoir manipuler les données au niveau logique permet d'optimiser les traitements sur les données. En effet, comme nous le verrons ultérieurement, les requêtes sur les données exprimées à partir du niveau ontologique sont traduites en requêtes utilisant les opérateurs définis au niveau logique. Or, cette traduction n'est pas toujours optimale. Plutôt que d'exprimer cette requête au niveau ontologique, l'accès au niveau logique donne la possibilité d'exécuter une requête optimale correspondant à cette traduction pour retrouver les mêmes données.

Un second intérêt de pouvoir manipuler les données au niveau logique est la possibilité d'améliorer l'implantation du modèle de données du niveau ontologique. Comme nous le montrons dans la section suivante, ce modèle de données associe chaque classe d'une ontologie à une extension permettant d'en stocker les instances. Ces extensions sont implantées au niveau logique suivant une des représentations proposées par les BDBO pour stocker les données à base ontologique (cf. chapitre 1, section 4.3). Lorsque la représentation horizontale, qui consiste à associer une table à chaque classe, est utilisée, les traitements disponibles au niveau logique peuvent être particulièrement utiles pour :

– renforcer l'intégrité des données. Des contraintes peuvent être ajoutées sur les tables utilisées ;

– exprimer les dépendances fonctionnelles. De nouvelles tables peuvent être créées en normalisant les tables utilisées. Le lien entre les tables normalisées ainsi créées et celles dénormalisées issues de la représentation initiale peut ensuite être défini grâce au mécanisme des vues ;
– optimiser cette implantation. Lorsqu'une extension de classe est implantée comme une vue, la requête utilisée pour définir cette vue peut être modifiée pour être optimisée. Ceci est particulièrement utile lorsque cette requête est le résultat de la traduction d'une requête OntoQL sur les données exprimée au niveau ontologique car elle n'est pas forcément optimale ;
– utiliser un schéma de base de données existant. Dans le cadre de l'indexation sémantique de bases de données, l'hypothèse que le schéma de base de données est construit indépendamment de l'ontologie doit pouvoir être posée. Les extensions de classes peuvent alors être implantées comme des vues sur ce schéma pré-existant.

Manipuler les données au niveau logique permet donc de gérer les données d'une BDBO en mettant en oeuvre les principes fondamentaux des bases de données tout en permettant d'intégrer des données représentées selon un schéma indépendant d'une ontologie particulière.

Bien entendu, la modification de l'implantation du niveau ontologique n'est pas sans risque. En effet, de la même manière qu'une modification du niveau physique d'une base de données (par exemple, la suppression d'un fichier) peut entraîner l'impossibilité d'utiliser le langage SQL, une modification du niveau logique (par exemple, la suppression d'une table) peut entraîner l'impossibilité d'utiliser le langage OntoQL au niveau ontologique. En conséquence, de telles manipulations doivent être faites avec précaution et doivent être réservées à un administrateur.

Si l'exploitation des données au niveau logique présente l'avantage de l'efficacité, il présente par contre l'inconvénient d'être dépendant de leur représentation logique. Pour pallier à cet inconvénient, le langage OntoQL permet d'accéder aux données depuis les concepts d'une ontologie, indépendamment de la représentation logique employée. Dans la section suivante, nous présentons les traitements disponibles pour manipuler les données à ce niveau.

3 Exploitation des données à base ontologique au niveau ontologique. Application à la couche OCC

Le niveau ontologique ajouté à l'architecture ANSI/SPARC offre la possibilité d'accéder aux données, indépendamment de la représentation logique des données, à partir des éléments d'une ontologie organisés selon les trois couches du modèle en oignon. Dans cette section, nous présentons les énoncés (opérateurs) du langage OntoQL définis pour accéder aux données depuis les concepts primitifs d'une ontologie (couche OCC). Ces énoncés sont définis sur le modèle de données défini ci-dessous.

3.1 Modèle de données du niveau ontologique, couche OCC

An niveau ontologique, le modèle de données est constitué des classes et des propriétés des ontologies ainsi que des instances de ces classes. Conformément au noyau commun des différents modèles d'ontologies que nous avons défini au chapitre 1, section 3, ces classes et ces propriétés sont identi-

fiées indépendamment d'une BDBO en utilisant un espace de noms. Elles sont caractérisées par une description textuelle éventuellement multilingue et organisées dans une hiérarchie autorisant l'héritage multiple. Les instances de ces classes possèdent un identifiant et sont caractérisées par des valeurs de propriétés. Seules des contraintes de typage et de cardinalité peuvent être définies sur ces instances et leurs valeurs de propriétés. Nous n'avons pas inclus d'autres contraintes telles que l'unicité car peu de modèles d'ontologies permettent actuellement de définir ce type de contrainte.

Contrairement au niveau logique où les tables et les colonnes constituent un *modèle structurel* pour les instances, au niveau ontologique, les classes et les propriétés ne constituent qu'un *modèle possible* pour les instances. En effet, la structure des instances d'une classe peut n'être composée que d'un sous-ensemble des propriétés définies sur cette classe. Afin de permettre de manipuler cette structure (exigence 6), qui constitue l'aspect conceptuel des données (les éléments d'une ontologie utiles pour une application donnée), nous avons choisi de la représenter explicitement dans le modèle de données du niveau ontologique. Ainsi, chaque classe peut être liée à une *extension* qui stocke les instances de cette classe ainsi que leur caractérisation sous forme de valeurs de propriétés. L'extension d'une classe ne comprend que le sous-ensemble des propriétés qui sont utilisées pour en décrire les instances.

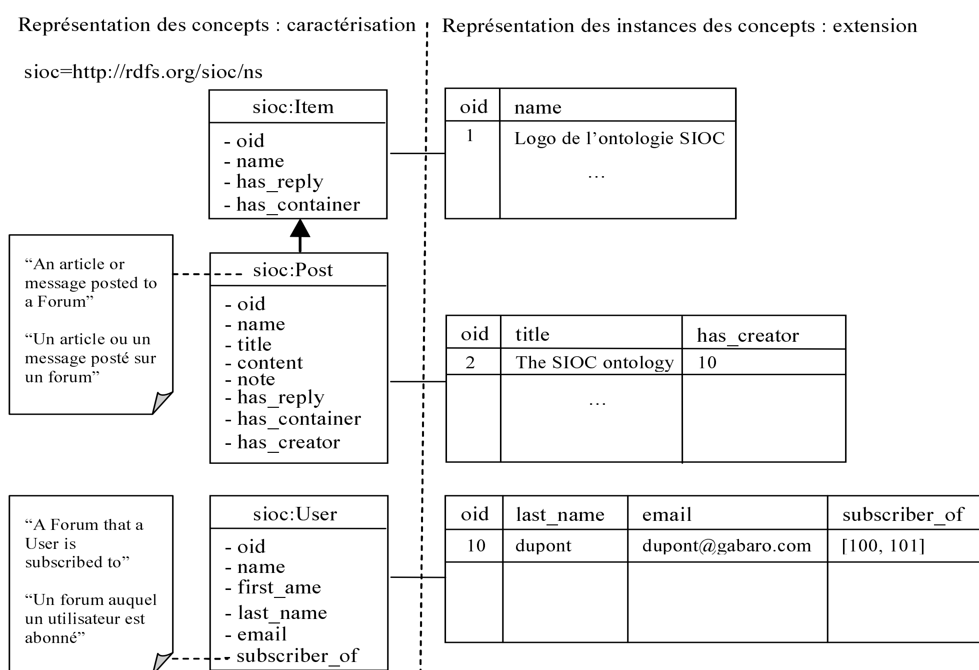


FIG. 3.1 – Illustration du modèle de données du niveau ontologique

La figure 3.1 illustre une utilisation du modèle de données du niveau ontologique sur l'exemple de l'ontologie SIOC. Sur la partie gauche, nous avons représenté les classes `Item`, `Post` (sous-classe de `Item`) et `User` avec l'ensemble des propriétés définies et héritées par ces classes. Ces classes sont définies dans l'espace de noms `http://rdfs.org/sioc/ns` dont l'alias est `sioc`. Ces classes et propriétés sont associées à une description. Sur cette figure, nous avons seulement indiqué un extrait de la description textuelle associée à la classe `Post` et à la propriété `subscriber_of`. Chacune des classes est également associée à une extension représentée sur cette figure sous la forme d'une table. Chaque extension comprend la propriété `oid` qui permet d'identifier les instances d'une classe de manière unique

ainsi que les propriétés utilisées pour décrire les instances.

Cet exemple montre que l'ensemble des propriétés applicables sur une classe n'est pas forcément utilisé pour en construire l'extension. Par exemple, l'extension de la classe `User` ne comprend pas les propriétés `name` et `first_name`. Il montre également que les propriétés des extensions ne sont pas nécessairement héritées. Par exemple, la propriété `name` utilisée dans l'extension de la classe `Item` n'est pas présente dans l'extension de la classe `Post` sous-classe de la classe `Item`.

Comme le montre le tableau 3.7, le modèle de données du niveau ontologique présente des similitudes avec le modèle relationnel-objet. Dans ce dernier, des tables typées peuvent être créées à partir d'un type utilisateur pour en stocker les instances décrites par des valeurs d'attributs. Un type utilisateur correspond donc à une classe, une table typée à une extension et un attribut à une propriété. Les types utilisateur, comme les classes, sont organisés dans une hiérarchie (même si seul l'héritage simple est autorisé pour les types utilisateur).

Modèle relationnel-objet	**Modèle de données du niveau ontologique**
Type utilisateur	Classe
Attribut	Propriété
Héritage simple entre types	Héritage multiple entre classes
Table typée (ensemble des attributs)	Extension (sous-ensemble des propriétés)
Héritage de table	
Référence d'objet	Référence d'instance
Type primitifs	Type primitifs
Type collection	Type collection
Composition	
Méthode	
	Identification universelle des classes et des propriétés
	Description des classes et des propriétés
	Type chaîne de caractères multilingues

TAB. 3.7 – Correspondances entre le modèle de données relationnel-objet et le modèle de données du niveau ontologique

Un autre point commun entre ces deux modèles est que chaque objet d'un type utilisateur, comme chaque instance d'une classe, possède un identifiant unique. L'ensemble des identifiants des objets d'un type `T` forme un type de données noté `REF(T)` (*référence d'objet*). Ces types de données peuvent également être construits à partir des classes. Une différence est que dans le modèle relationnel-objet une propriété de type `REF(T)` ne peut prendre ses valeurs que dans les tables typées correspondant au type `T` et non dans une de ses sous-tables. Sur l'exemple de l'ontologie SIOC, ceci imposerait que la propriété `has_container` définie sur la classe `Item` ne puisse prendre ses valeurs que dans la classe `Container` mais pas dans sa sous-classe `Forum`. Nous avons choisi de ne pas fixer cette contrainte sur les types références construits avec des classes d'une ontologie car elle se révélait trop contraignante pour différents projets dans lesquels nous avons utilisé le langage OntoQL[21]. Bien entendu, ne pas imposer cette

[21] notamment pour le projet EPISEM (`http://www.episem-action.org`)

contrainte engendre un coût sur le traitement des requêtes et sur la gestion de l'intégrité référentielle.

Enfin, ces deux modèles de données supportent des types primitifs et des types collection. Dans le modèle du niveau ontologique, nous avons choisi de ne supporter que le type `ARRAY` qui permet de représenter la plupart des types collection disponibles dans les modèles d'ontologies.

Ce tableau montre également des différences entre ces deux modèles. Ces différences sont une conséquence directe de la distinction que nous avons établie entre un schéma de base de données et une ontologie. Ces différences sont les suivantes.

- **Information incomplète.** Une extension d'une classe d'une ontologie est similaire à une table typée associée à un type utilisateur dans le modèle relationnel-objet. Cependant, contrairement à un schéma de base de données qui prescrit les attributs qui caractérisent les instances d'un type utilisateur, une ontologie décrit les propriétés qui peuvent être utilisées pour décrire les instances d'une classe. En conséquence, alors qu'une table typée possède une colonne pour chaque attribut d'un type utilisateur, une extension d'une classe d'une ontologie peut ne contenir qu'un sous-ensemble des propriétés définies sur cette classe.
- **Les relations de subsomption.** Dans le modèle relationnel-objet, les attributs sont hérités selon la hiérarchie de types. Les tables typées étant définies à partir des attributs de leur type utilisateur, elles héritent également des attributs de leur super-table selon la hiérarchie de types. L'héritage entre tables peut être indiqué explicitement afin de permettre d'exprimer des requêtes qui portent sur une table ainsi que sur ces sous-tables.

  Au niveau ontologique, les propriétés applicables sont distinguées des propriétés effectivement utilisées. Si les propriétés applicables sont héritées à travers la relation de subsomption comme dans le modèle de données relationnel-objet, ce n'est pas le cas de propriétés utilisées. Les classes peuvent être liées par une relation de subsomption sans exiger l'héritage des propriétés effectivement représentées. En conséquence, ce modèle ne permet pas de définir d'héritage entre les extensions. L'utilisation de classe offre ainsi beaucoup plus de flexibilité que l'utilisation de type utilisateur. C'est en particulier cette caractéristique qui est exploitée pour faciliter l'intégration d'informations [Nguyen-Xuan, 2006].
- **Identification universelle des classes et des propriétés.** Contrairement au modèle relationnel-objet où les identifiants des types utilisateurs ne sont uniques que dans le contexte d'une Base de Données Relationnelle-Objet (BDRO), les identifiants des classes et des propriétés sont définis dans le contexte d'un espace de noms permettant de référencer ces concepts à partir de n'importe quel environnement, indépendamment de la BDBO dans laquelle ils ont été définis.
- **Description précise, éventuellement multilingue des classes et des propriétés**. Les classes et les propriétés des ontologies sont associées à une description composée entre autre de noms et de définitions éventuellement définis dans plusieurs langues naturelles. Elles peuvent également être associées à de nombreux autres attributs et relations. Ces éléments permettent de définir précisément les classes et les propriétés, ce qui est essentiel pour atteindre un consensus parmi une communauté.

Notons également que le modèle de données relationnel-objet présente des constructeurs qui ne sont pas supportés par les modèles d'ontologies. D'abord, ce modèle permet de définir plusieurs tables typées à partir d'un même type utilisateur. Actuellement, le modèle de données de OntoQL ne permet de définir

qu'une seule extension par classe. Nous verrons dans les perspectives d'évolutions du langage OntoQL présentées au chapitre suivant que c'est une extension prévue au langage OntoQL car cette capacité peut être particulièrement utile dans le contexte de l'intégration de données.

Un autre constructeur supporté par le modèle relationnel-objet mais absent de notre modèle de données est la capacité de représenter des *relations de composition* en utilisant les types utilisateurs. En effet, un attribut dont le type de données est un type utilisateur `T` (et non `REF(T)`) représente une relation de composition (et non d'*agrégation*). En conséquence, les valeurs de cet attribut ne sont pas des objets possédant un identifiant. Ces valeurs sont supprimées en même temps que l'instance qu'elles décrivent. Cette relation est très peu représentée dans les modèles d'ontologies et nous ne l'avons donc pas inclue dans le modèle de données du niveau ontologique. Pour la même raison, nous n'avons pas permis la définition de méthodes sur les classes d'une ontologie ce qui est possible sur les types utilisateurs.

Dans cette section, nous avons présenté le modèle de données du niveau ontologique et montré qu'il présente des points communs avec le modèle relationnel-objet. Nous présentons maintenant des aspects syntaxiques du langage OntoQL qui permettent de manipuler les données à partir de ce niveau.

3.2 Aspects syntaxiques

Nous avons vu que le langage OntoQL permet d'accéder aux données au niveau logique (cf. section 2). Il doit également permettre d'accéder à ces données au niveau ontologique. Sa syntaxe doit donc permettre de distinguer ces deux niveaux d'accès.

3.2.1 Distinction des accès au niveau ontologique de ceux au niveau logique

Au niveau ontologique, les données sont manipulées à partir des classes et des propriétés des ontologies. Ainsi, la difficulté syntaxique est de distinguer les accès au niveau ontologique, utilisant des identifiants de classes et de propriétés, des accès au niveau logique, utilisant des identifiants de tables et de colonnes. Pour faire cette distinction, nous avons utilisé le fait que les classes et les propriétés sont définies dans une ontologie qui définit un domaine d'unicité des noms (espace de noms). Ainsi, contrairement aux identifiants de tables, l'identifiant de chaque classe et de chaque propriété comporte un espace de noms. Et donc, la distinction syntaxique entre les accès au niveau ontologique de ceux au niveau logique se fait sur la présence d'un espace de noms dans un identifiant.

Cette distinction étant établie, il reste le problème de définir des mécanismes simples pour indiquer les espaces de noms utilisés dans une instruction OntoQL. Deux mécanismes sont disponibles. Le premier consiste à ajouter une clause `USING NAMESPACE` pour indiquer le ou les espaces de noms utilisés dans une instruction OntoQL. La syntaxe de cette clause est la suivante[22] :

⟨*namespace clause*⟩ ::= `USING NAMESPACE` ⟨*namespace definition list*⟩

⟨*namespace definition*⟩ ::= [⟨*namespace alias*⟩ =] ⟨*namespace id*⟩

[22]Pour être concis, l'ensemble des éléments de la grammaire ne sont pas définis. La définition complète de cette grammaire ainsi que les notations utilisées peuvent être consultées dans l'annexe A. Par exemple, ⟨*XXX list*⟩ est un élément dont la définition est ⟨*XXX*⟩ { , ⟨*XXX*⟩ }.

Cette clause accepte la définition d'un espace de noms sans alias. Cet espace de noms est alors celui dans lequel sont recherchés par défaut les identifiants non préfixés d'une instruction OntoQL. Elle permet également d'associer un alias à un espace de noms. Pour utiliser un alias `ns` sur un identifiant `i`, la syntaxe `ns:i` doit être utilisée. Notons que ce mécanisme est également disponible dans les autres langages d'interrogation d'ontologies tels que SPARQL ou RQL.

Le second mécanisme, propre à OntoQL, consiste à définir les espaces de noms utilisés globalement et non plus localement à une instruction donnée. La syntaxe de l'instruction OntoQL correspondante est la suivante :

⟨*global namespace definition*⟩ ::= `SET NAMESPACE` [⟨*namespace alias*⟩ =] ⟨*namespace specification*⟩
⟨*namespace specification*⟩ ::= ⟨*namespace id*⟩ | `NONE`

Comme pour le mécanisme précédent, cette syntaxe permet de définir un espace par défaut et/ou de définir des alias sur des espaces de noms. Elle permet également de ne plus définir d'espace de noms par défaut en utilisant le mot clé `NONE`.

Exemple. Définir `http://rdfs.org/sioc/ns` comme étant l'espace de noms utilisé par défaut. Définir également l'alias `lisi` sur l'espace de noms `http://lisi.ensma.fr`.

```
SET NAMESPACE 'http://rdfs.org/sioc/ns'
SET NAMESPACE lisi='http://lisi.ensma.fr'
```

Explication. Une fois la première instruction exécutée, chaque instruction OntoQL sera considérée comme un traitement au niveau ontologique où les identifiants non préfixés seront recherchés dans l'espace de noms `http://rdfs.org/sioc/ns`[23]. Si la seconde instruction est exécutée, le préfixe `lisi` pourra être ajouté aux identifiants afin d'indiquer qu'ils doivent être recherchés dans l'espace de noms `http://lisi.ensma.fr`.

Le langage OntoQL permet ainsi de spécifier localement ou globalement les espaces de noms utilisés dans une instruction OntoQL afin de distinguer les accès aux données à base ontologique au niveau ontologique de ceux au niveau logique. Précisons maintenant comment sont construits les identifiants de classes et de propriétés recherchés par rapport à ces espaces de noms.

3.2.2 Identification des classes et des propriétés

OntoQL permet d'utiliser les *identifiants externes* définis dans une ontologie (attribut `ID` en OWL ou RDF-Schema, `code` en PLIB). Ces identifiants sont des chaînes de caractères. Dans notre exemple, ce sont des noms anglais dans lesquels un tiret bas remplace des espaces (par exemple, `has_creator`). Ces identifiants peuvent être utilisés sans syntaxe spécifique dans une instruction OntoQL. Ainsi, le mot `User` permet de référencer la classe du même nom.

De plus, lorsque les éléments d'une ontologie sont stockés dans un système qui propose une identification interne des données, la manipulation de ces données par cet identifiant est plus efficace que les manipulations effectuées par un identifiant externe. C'est le cas de la plupart des BDBO où une colonne de type entier est ajoutée à chaque table utilisée pour représenter les ontologies. Les *identifiants*

[23] Cet espace de noms sera celui utilisé par défaut dans les exemples de ce chapitre et des chapitres suivants.

internes de type entier (nommé `Oid`), contenus dans ces colonnes, peuvent être utilisés dans OntoQL en les préfixant par `!` afin de les différencier d'un identifiant externe. Par exemple, la syntaxe `!1016` permet d'identifier l'élément d'une ontologie ayant pour identifiant interne l'entier `1016`.

Ayant présenté les mécanismes syntaxiques que nous avons définis pour identifier les données au niveau ontologique, nous pouvons maintenant nous intéresser à la définition des langages permettant de manipuler ces données. Dans notre présentation du modèle de données du niveau ontologique (cf. section 3.1), nous avons montré que ce modèle présente de nombreux points communs avec le modèle relationnel-objet. En particulier, les classes et les propriétés des ontologies correspondent aux types utilisateurs et aux attributs. Notre objectif pour définir le langage OntoQL étant de concevoir un langage permettant d'accéder de façon homogène aux différents niveaux d'une BDBO, nous avons donc choisi de nous baser sur les opérateurs proposés par le langage SQL pour manipuler les types utilisateurs et leurs attributs. Toute la difficulté dans ce travail a été d'adapter la syntaxe et la sémantique de ces opérateurs aux particularités que présente le modèle de données du niveau ontologique par rapport au modèle relationnel-objet. Dans les sections suivantes nous présentons le résultat de ce travail pour les différents langages associés au langage OntoQL.

3.3 Langage de Définition de Données (LDD)

Le LDD doit permettre de créer les éléments du modèle de données définis précédemment, c'est-à-dire les classes et les propriétés des ontologies ainsi que les extensions de ces classes.

3.3.1 Création, modification et suppression des classes et des propriétés des ontologies

En SQL, les instructions `CREATE`, `ALTER` et `DROP TYPE` permettent de définir les types utilisateur et leurs attributs. Par contre, elles ne prennent pas en compte les spécificités suivantes des classes et des propriétés d'une ontologie :

1. un modèle d'ontologies peut permettre de créer différents types de classes. Par exemple, PLIB permet de définir des classes dont le type peut être `Item_class` ou `Model_class`. De même, OWL permet de définir des classes primitives (`Class`) ou des classes définies telles que les restrictions (par exemple, `HasValue`). De même, différents types de propriétés peuvent être définis. Par exemple le modèle PLIB permet de définir des propriétés dépendantes (`Dependent_p_det`) ou non (`Non_dependent_p_det`) du contexte ;
2. chaque classe et chaque propriété est décrite par un ensemble de valeurs d'attributs ;
3. les classes sont organisées dans une hiérarchie autorisant l'héritage multiple.

Nous avons adapté la syntaxe des instructions de création, modification et suppression d'un type utilisateur à ces spécificités.

3.3.1.1 Création de classes et de propriétés dans une ontologie

La syntaxe de création d'une classe et de ses propriétés est la suivante :

⟨*class definition*⟩ ::= CREATE ⟨*entity id*⟩ ⟨*class id*⟩ [⟨*under clause*⟩]
[⟨*descriptor clause*⟩] [⟨*properties clause list*⟩]
⟨*under clause*⟩ ::= UNDER ⟨*class id list*⟩
⟨*descriptor clause*⟩ ::= DESCRIPTOR (⟨*attribute value list*⟩)
⟨*attribute value*⟩ ::= ⟨*attribute id*⟩ = ⟨*value expression*⟩
⟨*properties clause*⟩ ::= ⟨*entity id*⟩ (⟨*property definition list*⟩)
⟨*property definition*⟩ ::= ⟨*prop id*⟩ ⟨*datatype*⟩ [⟨*descriptor clause*⟩]

L'en-tête de cette instruction débute par le mot clé `CREATE`. L'élément ⟨*entity id*⟩ permet ensuite de spécifier le type de la classe créée (spécificité 1) en indiquant le nom d'une entité. Dans le chapitre suivant, nous précisons les entités et les attributs qui peuvent être utilisés dans cette instruction. Pour la compréhension de ce chapitre, il est nécessaire d'indiquer (1) que les noms des entités et des attributs sont préfixés par le caractère # afin de distinguer les éléments d'une ontologie (par exemple, la propriété `title`) des éléments du modèle d'ontologies (par exemple, l'attribut `#name`) et (2) qu'il existe deux entités prédéfinies `#Class` et `#Property` regroupant respectivement les classes et les propriétés des ontologies.

Le nom du type de la classe créée est suivi de l'identifiant de la classe créée (⟨*class id*⟩) et de l'éventuelle liste de ses super-classes après le mot clé `UNDER` (spécificité 3). Le support de l'héritage multiple peut provoquer le problème d'*héritage répété*, c'est-à-dire qu'une propriété soit héritée via plusieurs chemins de la hiérarchie. Nous avons suivi l'approche des modèles d'ontologies supportant l'héritage multiple pour résoudre ce problème, c'est-à-dire que cette propriété ne soit héritée qu'en un seul exemplaire. Le corps de cette instruction est composé de plusieurs clauses optionnelles. La clause `DESCRIPTOR` peut être utilisée pour décrire la classe créée (spécificité 2) en permettant de spécifier des valeurs d'attributs (⟨*attribute id*⟩). D'autres clauses (⟨*properties clause*⟩) permettent de créer, en même temps que la classe, des propriétés (⟨*prop id*⟩) définies sur cette classe. Ces clauses débutent par ⟨*entity id*⟩ qui permet d'indiquer le type des propriétés créées (spécificité 1). Il est suivi par le type de données de la propriété créée (⟨*datatype*⟩) et éventuellement d'une nouvelle clause `DESCRIPTOR` qui permet d'indiquer la description de ces propriétés (spécificité 2).

Exemple. Créer la classe `Post` de l'exemple en indiquant que sa version est `001`.

```
CREATE #Class Post UNDER Item (
  DESCRIPTOR (#version = '001')
  #Property (
    title String, content String, note Int,
    has_creator REF(User) DESCRIPTOR (#version = '001'))
)
```

Explication. La classe `Post` est définie comme une classe (`#Class`) ayant pour super-classe la classe `Item`. La version de cette classe est indiquée dans la clause `DESCRIPTOR` en utilisant l'attribut `#version`. La clause `#Property` permet de créer des propriétés dont le domaine est la classe `Post`. Les propriétés `title` et `content` sont des propriétés du type primitif chaîne de caractères (`String`). La propriété `note` est de type entier (`Int`). Le type de la propriété `has_creator` est un type référence vers les instances de la classe `User`. La définition du type de données de cette propriété est suivie d'une clause `DESCRIPTOR` qui permet d'indiquer la version de cette propriété.

3.3.1.2 Modification de classes et de propriétés dans une ontologie

Nous avons étendu, de la même façon, l'instruction permettant de modifier un type utilisateur. La syntaxe obtenue est la suivante :

⟨*alter class statement*⟩ ::= `ALTER` ⟨*class id*⟩ [⟨*descriptor clause*⟩] [⟨*alter class action*⟩]
⟨*alter class action*⟩ ::= ⟨*add property definition*⟩ | ⟨*drop property definition*⟩
⟨*add property definition*⟩ ::= `ADD` [⟨*entity id*⟩] ⟨*property definition*⟩ [⟨*descriptor clause*⟩]
⟨*drop property definition*⟩ ::= `DROP` ⟨*property id*⟩

Cette instruction permet de modifier la description d'une classe dans une clause `DESCRIPTOR`. Les valeurs d'attributs indiquées dans cette clause surchargent celles qui peuvent avoir été définies précédemment. Cette instruction permet également d'ajouter (`ADD`) ou de supprimer (`DROP`) une propriété définie sur cette classe. Pour l'ajout d'une propriété, si son type n'est pas indiqué, le type `#Property` est utilisé par défaut.

La sémantique de cette instruction est similaire à celle de modification d'un type utilisateur. Ainsi, la suppression d'une propriété héritée n'est pas autorisée. De même, il n'est pas possible de supprimer une propriété si elle est la seule propriété définie sur une classe.

Exemple. Indiquer que la version de la classe `Post` est maintenant `002`. Ajouter également la propriété `size` de type entier à cette classe.
`ALTER Post DESCRIPTOR (#version='002') ADD size INT`

Explication. La clause `DESCRIPTOR` indique les nouvelles valeurs d'attributs de la classe à modifier. Après l'exécution de cette instruction, la classe `Post` aura donc la valeur `002` pour l'attribut `#version`. La clause `ADD` permet d'ajouter une propriété à la classe modifiée. La propriété `size` est ainsi ajoutée à la classe `Post`. L'entité dont cette propriété est une instance n'étant pas précisée, `#Property` est utilisé par défaut.

3.3.1.3 Suppression de classes et de propriétés dans une ontologie

Enfin, voici la syntaxe permettant de supprimer une classe et les propriétés associées :

⟨*drop class definition*⟩ ::= `DROP` ⟨*class id*⟩

Cette instruction permet de supprimer la classe d'identifiant ⟨*class id*⟩ et les propriétés définies sur cette classe. Cette suppression ne sera effectuée que si cette classe n'est pas référencée. Ainsi, cette classe ne doit pas avoir des sous-classes, ni d'extension et ne doit pas être utilisée dans un type référence. Notons que nous n'avons pas offert la possibilité de réaliser des suppressions en cascade (mot clé `CASCADE` en SQL) à cause de la complexité des traitements engendrés. Cette fonctionnalité est d'ailleurs peu implantée dans les SGBD.

Exemple. Supprimer la classe `Post`.
`DROP Post`

Explication. La classe `Post` n'ayant aucune sous-classe et n'étant pas utilisée dans un type référence, elle ne sera supprimée que si elle n'a pas d'extension. Si c'est le cas, les propriétés `title`, `content`, `note` et `has_creator`, définies sur cette classe, seront également supprimées.

3.3.2 Création, modification et suppression des extensions des classes

A partir des classes et des propriétés créées, le LDD doit permettre de créer les extensions permettant de contenir les instances de chaque classe. Une extension correspond à une table typée du modèle relationnel-objet. En SQL, une table typée est créée simplement en indiquant le type utilisateur auquel elle est associée. Ses colonnes se déduisent des attributs de ce type utilisateur. Dans le cas d'une extension, les différences suivantes sont à prendre en compte :

1. une extension peut ne contenir qu'un sous-ensemble des propriétés applicables sur une classe ;
2. il n'existe qu'une seule extension par classe ;
3. il n'y a pas d'héritage entre les extensions ;
4. une extension est implantée au niveau logique.

3.3.2.1 Création de l'extension d'une classe

La syntaxe de création de l'extension d'une classe est la suivante :

⟨*extension definition*⟩	::=	`CREATE EXTENT OF` ⟨*class id*⟩ `(` ⟨*property id list*⟩ `)` [⟨*logical clause*⟩]
⟨*logical clause*⟩	::=	`TABLE` [⟨*table and column name*⟩]
⟨*table and column name*⟩	::=	⟨*table name*⟩ [`(` ⟨*column name list*⟩ `)`]

Plusieurs tables typées pouvant être associées à un type utilisateur, l'instruction de création d'une table typée nécessite de nommer la table créée. Ceci n'est pas nécessaire pour l'instruction de création d'une extension puisqu'elle est unique pour une classe donnée. En conséquence, l'en-tête de cette instruction permet de définir une extension en indiquant seulement le nom de la classe (`OF` ⟨*class id*⟩) dont elle est l'extension (spécificité 2). Cette en-tête ne comprend pas le mot clé `UNDER` puisqu'il n'y a pas d'héritage entre les extensions (spécificité 3). Le corps de cette instruction est constitué de l'élément ⟨*property id list*⟩ qui permet d'indiquer la liste des propriétés incluses dans cette extension (spécificité 1). Par défaut, l'interpréteur d'une telle instruction implante l'extension créée au niveau logique selon la représentation horizontale, c'est-à-dire par une table (actuellement non normalisée) comprenant une colonne pour chaque propriété de l'extension (spécificité 4). Les noms de cette table et de ses colonnes sont générés automatiquement. Néanmoins, le corps de cette instruction peut être suivi d'une clause ⟨*logical clause*⟩ qui permet à un utilisateur d'indiquer les noms qu'il souhaite (⟨*table and column name*⟩). Notons que cette capacité permet de passer du niveau ontologique au niveau logique et donc de répondre en partie à l'exigence 6. Cette capacité est similaire à la possibilité offerte par SQL de passer du niveau logique au niveau physique en indiquant des informations sur le stockage physique des données d'une table dans une instruction `CREATE TABLE`.

Exemple. Créer l'extension de la classe `Post` présentée sur la figure 3.1.

```
CREATE EXTENT OF Post (title, has_creator)
       TABLE T_Post (title, has_creator)
```

Explication. L'extension de la classe `Post` est créée en indiquant que seules les propriétés `title` et `has_creator` sont utilisées pour en décrire les instances. La clause `TABLE` impose que cette extension soit créée sous la forme d'une table nommée `T_Post`. Cette table comportera trois colonnes :

la colonne `oid`, pour l'identifiant des instances, et les colonnes nommées `title` et `has_creator` correspondant aux propriétés du même nom.

3.3.2.2 Modification de l'extension d'une classe

Afin de faciliter la manipulation des extensions d'une classe, nous avons défini la syntaxe suivante pour permettre de modifier une extension :

⟨*alter extension statement*⟩ ::= `ALTER EXTENT OF` ⟨*class id*⟩ ⟨*alter extent action*⟩
⟨*alter extension action*⟩ ::= ⟨*add property definition*⟩ | ⟨*drop property definition*⟩
⟨*add property definition*⟩ ::= `ADD` [`PROPERTY`] ⟨*property id*⟩ [`COLUMN` ⟨*column name*⟩]
⟨*drop property definition*⟩ ::= `DROP` [`PROPERTY`] ⟨*property id*⟩

Cette instruction permet d'ajouter une propriété (`ADD`) à une extension en indiquant éventuellement le nom de la colonne créée dans la table du niveau logique correspondant à cette extension. Elle permet également de supprimer une propriété d'une extension (`DROP`).

Exemple. Ajouter la propriété `name` à l'extension créée précédemment pour la classe `Post`.

```
ALTER EXTENT OF Post ADD PROPERTY name COLUMN nom
```

Explication. L'exécution de cette instruction se traduit par l'ajout d'une colonne nommée `nom` dans la table utilisée pour représenter l'extension de la classe `Post` au niveau logique.

3.3.2.3 Suppression de l'extension d'une classe

Une extension peut également être supprimée par une instruction dont la syntaxe est la suivante :

⟨*drop extension statement*⟩ ::= `DROP EXTENT OF` ⟨*class id*⟩

La suppression d'une extension n'est possible que si ses instances ne sont pas référencées dans un type référence. L'interpréteur de ce type d'instruction a ainsi la charge de vérifier cette condition et de lever une erreur si elle n'est pas satisfaite.

Exemple. Supprimer l'extension de la classe `Post`.

```
DROP EXTENT OF Post
```

Explication. Cette instruction n'est valide que si les instances de la classe `Post` ne sont pas utilisées comme valeurs pour les propriétés de type `REF(Post)`. Si cette précondition est satisfaite, cette instruction supprime l'extension de la classe `Post` ainsi que la table implantant cette extension au niveau logique.

Dans cette section, nous avons montré le langage de définition de données de OntoQL qui permet de créer des conteneurs pour les instances des classes ainsi que pour leur description. Dans la section suivante, nous décrivons le langage permettant de manipuler ces instances et leur description.

3.4 Langage de Manipulation de Données (LMD)

Le LMD doit permettre de créer, de mettre à jour et de supprimer les instances des classes dans une BDBO. Ces instances ne présentent pas de différence majeure avec les objets d'un type utilisateur stockés dans une table typée d'une BDRO. En effet, elles possèdent un identifiant et ne peuvent présenter de valeur que pour les propriétés de l'extension dans laquelle elles sont stockées. En conséquence, la syntaxe du langage de manipulation de données reste conforme à celle de SQL.

3.4.1 Insertion d'instances de classe

Conformément à SQL, une instruction `INSERT` permet de créer des instances selon la syntaxe suivante :

⟨*insert statement*⟩ ::= `INSERT INTO` ⟨*class id*⟩ ⟨*insert description and source*⟩
⟨*insert description and source*⟩ ::= ⟨*from subquery*⟩ | ⟨*from constructor*⟩
⟨*from subquery*⟩ ::= [`(` ⟨*property id list*⟩ `)`] ⟨*query expression*⟩
⟨*from constructor*⟩ ::= [`(` ⟨*property id list*⟩ `)`] ⟨*values clause*⟩
⟨*values clause*⟩ ::= `VALUES (` ⟨*values expression list*⟩ `)`

Cette syntaxe permet de créer une ou plusieurs instances d'une classe. En SQL, puisque plusieurs tables typées peuvent être associées à un type utilisateur, l'instruction `INSERT` nécessite de spécifier le nom de la table typée dans laquelle les instances vont être insérées. Puisqu'une classe n'a qu'une seule extension, seul son identifiant (⟨*class id*⟩) est nécessaire pour déterminer l'extension dans laquelle les instances doivent être insérées. Les instances créées peuvent être le résultat d'une requête (⟨*from subquery*⟩). Une instance peut également être créée manuellement en indiquant ses valeurs de propriétés (⟨*insert property list*⟩). Ces propriétés doivent être présentes dans l'extension de cette classe, sinon une erreur est levée. A part cela, la sémantique de cette instruction est similaire à celle de SQL. Par exemple, il est interdit de spécifier plusieurs fois la même propriété dans la clause ⟨*insert property list*⟩.

Exemple. Ajouter l'instance de la classe `Post` présente sur la figure 3.1.

```
INSERT INTO Post (title, has_creator) VALUES ('The SIOC ontology', 10)
```

Explication. Cette instruction n'est valide que si l'extension de la classe `Post` comprend les propriétés `title` et `has_creator`. Elle suppose également qu'il existe une instance de la classe `User` dont l'identifiant est `10`. Cet identifiant aurait également pu être recherché par une requête.

3.4.2 Mise à jour d'instances de classe

L'instruction `UPDATE` permet de modifier les instances créées suivant la syntaxe suivante :

⟨*update statement*⟩ ::= `UPDATE` ⟨*class id polymorph*⟩ `SET` ⟨*set clause list*⟩
[`WHERE` ⟨*search condition*⟩]
⟨*class id polymorph*⟩ ::= ⟨*class id*⟩ | `ONLY` (⟨*class id*⟩)
⟨*set clause*⟩ ::= ⟨*property id*⟩ = ⟨*value expression*⟩

Cette syntaxe permet de modifier les instances d'une classe (⟨*class id*⟩) et de modifier ses valeurs de propriétés (⟨*set clause*⟩). A l'image de SQL qui permet de ne modifier que les tuples d'une table et non de ses sous-tables, cette instruction permet aussi de ne modifier que les instances d'une classe et non de ses sous-classes (`ONLY`).

Exemple. Mettre en majuscule les noms des instances de la classe `Item`.

```
UPDATE ONLY(Item) SET name = UPPER(name)
```

Explication. Grâce à l'opérateur `ONLY`, les instances de la classe `Post`, sous-classe de la classe `Item`, ne sont pas concernées par cette instruction. L'utilisation de cet opérateur est nécessaire dans cet exemple car l'extension de la classe `Post` ne comprend pas la propriété `name`. Et donc, une mise à jour polymorphe provoquerait une erreur.

La modification met en majuscule la valeur de la propriété `name` en utilisant la fonction prédéfinie `UPPER`. La plupart des fonctions prédéfinies de SQL peuvent également être utilisées dans une instruction OntoQL (cf. annexe A).

3.4.3 Suppression d'instances de classe

L'instruction `DELETE` permet de supprimer les instances d'une classe suivant la syntaxe suivante :

⟨*delete statement*⟩ ::= `DELETE FROM` ⟨*class id polymorph*⟩ [`WHERE` ⟨*search condition*⟩]

Comme pour l'instruction `UPDATE`, la suppression d'instances peut porter uniquement sur les instances d'une classe ou également sur celles de ses sous-classes. La suppression d'une instance n'est possible que si cette instance n'est pas référencée par un type référence.

Exemple. Supprimer les instances de la classe `Item` dont le nom n'est pas en majuscule.

```
DELETE ONLY(Item) WHERE name <> UPPER(name)
```

Explication. Comme dans l'exemple précédent, seules les instances directes de la classe `Item` sont considérées dans cette instruction. Les instances supprimées sont celles dont le nom est différent (<>) du même nom en majuscule.

3.5 Langage d'Interrogation de Données (LID)

Pour définir les langages de définition et de manipulation de données, nous avons choisi d'adapter les langages proposés par SQL car il répondait naturellement à nos besoins (créer des classes, des instances etc.). Le besoin auquel répond un langage d'interrogation est de pouvoir effectuer des requêtes sur les données. Ce besoin est beaucoup plus vague que celui exprimé pour les langages de définition et de manipulation. Il ne précise en particulier pas la puissance d'expression nécessaire pour exprimer ces requêtes. Avant même de chercher à adapter le langage SQL au modèle de données que nous avons défini, nous avons donc dû déterminer si ce langage répondait à nos besoins en terme de puissance d'expression.

3.5.1 Opérateurs supportés par OntoQL

Le langage d'interrogation de SQL permet d'interroger les données d'une table typée en utilisant non seulement les opérateurs relationnels introduits dans la norme SQL92 mais aussi les opérateurs relationnels-objets introduits dans la norme SQL99. Ces opérateurs ont été présentés brièvement à la section 2.2 et sont énumérés dans les tableaux 3.4 et 3.5. Pour déterminer si ces opérateurs répondent à nos besoins en terme de puissance d'expression, nous nous sommes basés sur les trois éléments suivants :

- les projets dans lesquels le langage OntoQL est susceptible d'être utilisé[24]. Actuellement, les besoins exprimés dans ces projets sont couverts par les opérateurs proposés par SQL ;
- le pouvoir d'expression proposé par les autres langages. Le langage le plus expressif est le langage RQL. Il se base sur le langage OQL dont la plupart des opérateurs sont communs avec SQL. Par rapport à OQL, RQL permet en plus d'exprimer des *expressions de chemins généralisées* [Christophides et al., 1994, Christophides et al., 1996] qui permettent d'exprimer des requêtes lorsque le schéma des données n'est pas complètement connu. La satisfaction de l'exigence 9 permettant de découvrir les ontologies d'une BDBO, nous avons choisi de ne pas augmenter la complexité du langage OntoQL en introduisant ces opérateurs ;
- la nécessité de valider le langage OntoQL en l'implantant sur le prototype de BDBO OntoDB. Cet élément nous a conduit à écarter l'opérateur SQL permettant d'exprimer des requêtes récursives vu la complexité de son implantation sur ce prototype.

Nous avons donc choisi de supporter les principaux opérateurs de SQL92 et SQL99 (mis à part l'opérateur `WITH RECURSIVE`). Le choix des opérateurs supportés étant fait, il reste à les adapter aux spécificités que présente le modèle du niveau ontologique par rapport au modèle relationnel-objet.

3.5.2 Mise en oeuvre des opérateurs issus de SQL

Les spécificités de notre modèle de données, à prendre en compte, sont les suivantes :

1. les classes et les propriétés ont un identifiant universel ;
2. chaque classe est associée à une et une seule extension ;
3. les classes et les propriétés ont une description textuelle éventuellement multilingue ;
4. les instances d'une classe peuvent ne présenter de valeurs que pour un sous-ensemble des propriétés applicables sur cette classe.

Précédemment, nous avons indiqué que la spécificité 1 est prise en compte par l'ajout de la clause `USING NAMESPACE` à une instruction OntoQL et que la spécificité 2 permet d'utiliser directement l'identifiant d'une classe plutôt que celle de son extension pour référencer ses instances. Nous montrons en section 5 que la spécificité 3 est prise en compte par l'ajout d'une clause `USING LANGUAGE` à une instruction OntoQL. Il reste donc à étudier les conséquences de la spécificité 4.

Une instance pouvant ne pas présenter de valeur pour une propriété donnée, il est nécessaire (1) de déterminer quelle est la valeur retournée lorsqu'une telle propriété est appliquée à une instance et (2)

[24] cf. `http://www.plib.ensma.fr/`

d'adapter la sémantique des opérateurs du langage pour qu'il puisse traiter cette valeur. Pour répondre au point (1), nous nous sommes intéressés au sens à donner au fait qu'une instance ne présente pas de valeur pour une propriété donnée. Les propriétés qui ne sont pas présentes dans l'extension d'une classe correspondent à celles qui n'ont pas d'intérêt par rapport à l'application en cours de conception. Par exemple, si l'on souhaite proposer une application fournissant des statistiques sur l'usage de forums (nombre de messages par utilisateur, par forum etc.), les propriétés `title`, `content` et `note` de la classe `Post` ne présentent pas d'intérêt. Les instances de la classe `Post` peuvent avoir une valeur pour ces propriétés, mais cette valeur n'est pas représentée dans la BDBO. Cette sémantique (valeur inconnue) correspond à l'interprétation la plus fréquente de la valeur `NULL` en base de données, même si de nombreux autres sont possibles [Date, 1982]. En conséquence, nous avons choisi qu'une projection d'instances sur une propriété non utilisée retourne la valeur `NULL`.

Les opérateurs de SQL ont été définis pour prendre en compte la valeur `NULL`. Répondre au point (2) consiste donc seulement à déterminer si cette sémantique est celle que nous voulons donner au langage OntoQL. Ayant comme objectif de rester compatible avec SQL, nous avons choisi naturellement de conserver cette sémantique. La sémantique des principaux opérateurs du langage pour la valeur `NULL` est donc la suivante :

- les opérateurs arithmétiques comme + ou / retournent la valeur `NULL` lorsqu'ils prennent en paramètre cette valeur ;
- les opérateurs de comparaison comme = ou > sont définis par rapport à une logique tri-valuée (`TRUE`, `FALSE` et `UNKNOWN`). Ils retournent la valeur `UNKNOWN` lorsqu'ils prennent en paramètre la valeur `NULL` ;
- les tables de vérités des opérateurs logiques comme `AND` ou `OR` sont définies pour prendre en compte la logique tri-valuée résultant de l'introduction de la valeur `NULL` ;
- chaque fonction prédéfinie comme `UPPER` est définie pour prendre en compte cette valeur. Par exemple, `UPPER` retourne la valeur `NULL` lorsqu'elle prend en paramètre cette valeur ;
- l'opérateur de partitionnement horizontale (`GROUP BY`) crée un groupe contenant l'ensemble des tuples lorsqu'il est appliqué à une propriété non utilisée ;
- l'opérateur de tri `ORDER BY` ne modifie pas son paramètre d'entrée lorsqu'il est appliqué à une propriété non utilisée.

Ayant précisé les opérateurs du langage OntoQL et comment nous les avons adaptés aux spécificités du modèle de données considéré, nous illustrons maintenant le langage obtenu. Afin de rester concis, nous décrivons d'abord la forme générale d'une requête puis présentons les opérateurs orientés-objets du langage.

3.5.3 Forme générale d'une requête OntoQL

La syntaxe générale d'une requête OntoQL est la suivante :

⟨*query specification*⟩ ::= ⟨*select clause*⟩ ⟨*from clause*⟩ [⟨*where clause*⟩]
[⟨*group by clause*⟩] [⟨*having clause*⟩] [⟨*order by clause*⟩]
[⟨*namespace clause*⟩] [⟨*language clause*⟩]

Comme en SQL, le paramètre d'entrée et de sortie d'une requête OntoQL est une relation. Cette relation est constituée d'attributs dont le type peut être primitif, un type collection, un type référence ou une classe d'une ontologie (ce dernier type d'attribut est construit grâce à l'opérateur DEREF présenté à la section 3.5.5). Ce paramètre d'entrée est défini dans la clause FROM. En SQL, pour interroger les instances des types utilisateur, ce paramètre est indiqué en utilisant le nom des tables typées. N'associant qu'une seule extension à une classe, OntoQL permet d'utiliser directement l'identifiant des classes pour en interroger les instances. L'identifiant d'une classe est considéré comme une relation possédant un attribut pour chaque propriété applicable sur cette classe (y compris la propriété oid). OntoQL considère que les tuples de cette relation possèdent la valeur NULL pour les propriétés non utilisées.

Mise à part la clause USING NAMESPACE que nous avons décrite précédemment et la clause USING LANGUAGE que nous présentons en section 5.2, les autres clauses de OntoQL sont similaires à celles de SQL. Ainsi, la clause SELECT permet de préciser le retour d'une requête en indiquant des propriétés, des appels de fonctions ou des expressions arithmétiques qui doivent être évalués pour chaque tuple. La clause WHERE est optionnelle, elle permet de restreindre les tuples parcourus à ceux satisfaisant des prédicats. La clause GROUP BY permet de grouper les tuples selon une valeur déterminée à partir de propriétés, de fonctions ou d'expressions arithmétiques. La clause HAVING permet de restreindre les groupes considérés à ceux satisfaisant des prédicats. Enfin, la clause ORDER BY permet de trier les résultats de la requête.

Exemple. Retourner, par ordre croissant, les noms des utilisateurs qui ont écrit plus de trois messages.

```
SELECT u.last_name
  FROM Post AS p, User AS u
 WHERE p.has_creator = u.oid
GROUP BY u.oid, u.last_name
HAVING count(p.oid) > 3
ORDER BY u.last_name ASC
USING NAMESPACE http://rdfs.org/sioc/ns
```

Explication. La clause USING NAMESPACE permet d'indiquer que cette requête est exécutée à partir d'éléments du niveau ontologique appartenant à l'ontologie SIOC. Les clauses FROM et WHERE réalisent la jointure entre les classes Post et User. Pour permettre de référencer les instances des classes Post et User ainsi jointes, les variables p et u, nommées *itérateurs*, sont introduites. Les clauses GROUP BY et HAVING permettent de regrouper les messages écrits par chaque auteur et de ne conserver que ceux en ayant écrits plus de 3. La clause SELECT permet de retourner le nom des auteurs restants. Ils sont triés dans l'ordre croissant grâce à la clause ORDER BY.

En soi, la requête SELECT présentée n'est pas nouvelle. C'est son utilisation exploitant l'architecture de bases de données définie et, comme nous le verrons ultérieurement, combinant l'accès aux concepts, aux instances et aux deux qui en fait un opérateur puissant d'interrogation de BDBO.

Une caractéristique des requêtes OntoQL est que les opérateurs orientés-objets de SQL peuvent être utilisés dans ces différentes clauses. Comme nous l'avons vu précédemment, peu de SGBD implantent ces opérateurs. Le support de ces opérateurs est donc une caractéristique importante du langage OntoQL. Ces opérateurs sont présentés dans les sections suivantes en les illustrant par des exemples.

3.5.4 Opérateurs sur les types des instances

Par défaut, lorsqu'une classe est introduite dans la clause `FROM`, une requête OntoQL porte non seulement sur les instances de cette classe mais aussi sur celles de ces sous-classes (requête polymorphe). OntoQL permet de n'exécuter une requête que sur les instances directes d'une classe, c'est-à-dire celles dont elle est la classe de base.

Exemple. Rechercher le nom des instances directes de la classe `Item`.

```
SELECT i.name FROM ONLY(Item) AS i
```

Explication. Par la présence du mot clé `ONLY`, les instances de la classe `Post`, sous-classe de `Item`, ne seront pas considérées par cette requête.

OntoQL permet également de tester le type d'une instance dans une requête et de convertir une instance dans un autre type. Ceci est particulièrement utile pour les requêtes polymorphes.

Exemple. Rechercher les noms des instances de la classe `Item`. Lorsqu'une instance est un message (`Post`), retourner également son titre.

```
SELECT i.name,
       CASE WHEN DEREF(i.oid) IS OF (Post)
            THEN TREAT (DEREF(i.oid) AS Post).title
            ELSE NULL END
  FROM Item AS i
```

Explication. Cette requête parcourt un ensemble de tuples correspondant aux instances de la classe `Item` via l'itérateur `i`. La propriété `name` peut être appliquée sur ces tuples car elle est définie sur la classe `Item`. Par contre, ce n'est pas le cas de la propriété `title` puisqu'elle ne s'applique qu'à la classe `Post`. Pour cela, un test est utilisé dans la clause `SELECT` (`CASE`). L'opérateur `IS OF` permet de déterminer si le tuple parcouru par `i` correspond à une instance de la classe `Post`. Ce test requiert de récupérer cette instance en utilisant l'opérateur `DEREF` présenté dans la section suivante. Si c'est le cas, cette instance est convertie explicitement en une instance de cette classe (opérateur `TREAT`) ce qui permet de lui appliquer la propriété `title`. Si ce n'est pas le cas, la valeur `NULL` est retournée.

3.5.5 Parcours de références

OntoQL permet de traverser des associations représentées par des propriétés de type référence (expressions de chemins). En SQL, la notation `->` est utilisée pour construire une expression de chemin. Par exemple, `p.has_reply->has_creator` permet de rechercher l'auteur d'une réponse au message `p`. SQL distingue donc la notation pointée qui permet d'accéder à une propriété depuis une instance donnée, de la notation `->` qui permet d'accéder à une propriété depuis une référence d'instance. Cette distinction permet d'éviter des ambiguïtés syntaxiques. En effet, sans l'utilisation de la notation `->`, l'expression `p.has_reply.has_creator` peut être ambiguë si `p` est à la fois un itérateur introduit dans la clause `FROM` et une propriété. Pour simplifier la syntaxe de OntoQL, nous avons choisi de suivre l'approche d'Oracle qui, plutôt que d'introduire la notation `->`, impose qu'une expression de chemin débute toujours par l'utilisation d'un itérateur.

Exemple. Rechercher le nom et le prénom des personnes qui ont répondu à un message de Dupont.

```
SELECT p.has_reply.has_creator.last_name,
       p.has_reply.has_creator.first_name
  FROM Post AS p
 WHERE p.has_creator.last_name = 'Dupont'
```

Explication. L'expression de chemin p.has_reply.has_creator permet de réaliser une jointure implicite entre les classes Post et User, facilitant ainsi l'écriture de la requête. Pour pouvoir utiliser cette expression de chemin, l'introduction de l'itérateur p sur la classe Post est obligatoire.

Dans la requête précédente, seules les valeurs des propriétés nom et prénom des instances de la classe User sont retournées. L'opérateur DEREF permet de retourner les instances complètes d'une classe.

Exemple. Retourner les instances de la classe User qui ont répondu à un message de Dupont.

```
SELECT DEREF(p.has_reply.has_creator)
  FROM Post AS p
 WHERE p.has_creator.last_name = 'Dupont'
```

Explication. Cette requête est similaire à la précédente à part que l'opérateur DEREF, qui retourne une instance à partir de son identifiant, est utilisé. Ainsi, le résultat sera une relation à une colonne dont voici un extrait possible :

DEREF(p.has_reply.has_creator)
User(1, NULL, NULL, 'Dupont', 'dupont@gabaro.com', [#100, #101])
User(2, NULL, NULL, 'Durand', 'durand@sidora.fr', [#103, #104])

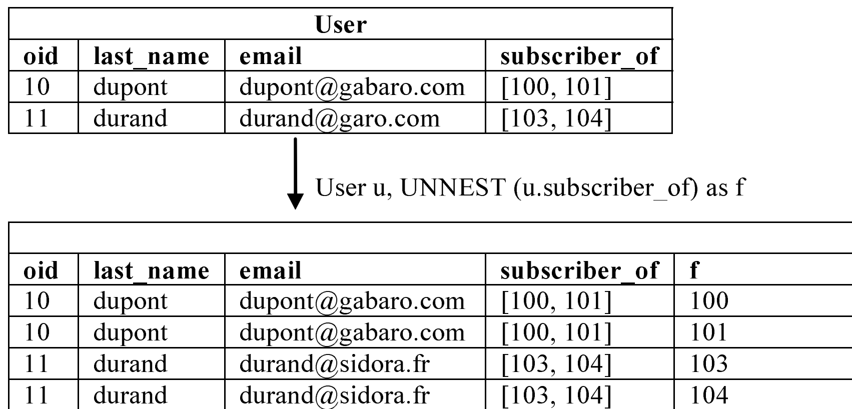
3.5.6 Manipulation des collections

L'opérateur UNNEST permet de transformer une collection en une relation pour qu'elle puisse être utilisée dans la clause FROM. Cet opérateur est souvent utilisé pour dégrouper une relation par rapport à un attribut de type collection. La figure 3.2 illustre le dégroupage de la relation correspondant à la classe User pour l'attribut de type collection subscriber_of. Le résultat est une relation mise à plat où un tuple a été créé pour chaque élément de la collection en dupliquant la valeur des autres attributs. Voici un exemple d'utilisation de cet opérateur dans le langage.

Exemple. Rechercher, pour chaque utilisateur, les noms des sites qui hébergent les forums auxquels il est abonné.

```
SELECT u.name, f.has_host.name
  FROM User AS u, UNNEST(u.subscriber_of) AS f
```

Explication. L'expression UNNEST(u.subscriber_of) AS f applique l'opérateur UNNEST à la collection subscriber_of. Elle introduit un itérateur f sur les forums auxquels l'utilisateur courant (itérateur u) est abonné. La clause SELECT permet de retrouver le nom du site qui héberge le forum f grâce à une expression de chemin. Le résultat de cette requête est le suivant :



User			
oid	**last_name**	**email**	**subscriber_of**
10	dupont	dupont@gabaro.com	[100, 101]
11	durand	durand@garo.com	[103, 104]

User u, UNNEST (u.subscriber_of) as f

oid	**last_name**	**email**	**subscriber_of**	**f**
10	dupont	dupont@gabaro.com	[100, 101]	100
10	dupont	dupont@gabaro.com	[100, 101]	101
11	durand	durand@sidora.fr	[103, 104]	103
11	durand	durand@sidora.fr	[103, 104]	104

FIG. 3.2 – Exemple d'opération UNNEST

u.name	f.has_host.name
'Dupont'	'Gabaro'
'Dupont'	'Sidora'
'Durant'	'Sidora'

Le résultat de la requête précédente est une relation contenant une ligne pour chaque nom de site. Ces noms peuvent également être retournés comme une collection.

Exemple. Retourner, pour chaque utilisateur, la liste des noms des sites qui hébergent les forums auxquels il est abonné.

```
SELECT u.name, ARRAY(SELECT f.has_host.name
                       FROM UNNEST(u.subscriber_of) AS f)
  FROM User AS u
```

Explication. La requête principale recherche tous les utilisateurs. Pour chaque utilisateur, la requête imbriquée dans la clause SELECT recherche le nom des sites qui hébergent les forums auxquels il est abonné. Le résultat de cette requête (une relation) est retourné comme une collection en utilisant le constructeur (ARRAY()). Ce constructeur ne peut être utilisé que pour une relation à une seule colonne. Le résultat de cette requête est le suivant :

u.name	f.has_host.name
'Dupont'	{'Gabaro', 'Sidora'}
'Durant'	{'Sidora'}

Plutôt que d'utiliser l'opérateur UNNEST dans la requête précédente, il est également possible de faire une jointure explicite.

Exemple. Écriture équivalente à la requête précédente réalisant une jointure explicite entre les classes User et Forum.

```
SELECT u.name, ARRAY(SELECT f.has_host.name
                       FROM Forum AS f
                      WHERE f.oid = ANY(u.subscriber_of))
```

```
 FROM User AS u
WHERE u.last_name = 'Dupont'
```

Explication. Pour réaliser une jointure entre les classes `User` et `Forum`, l'opérateur `= ANY()` est utilisé. Il permet de tester si un élément appartient à une collection.

Nous venons de présenter les différents langages permettant de manipuler les données à base ontologique au niveau ontologique, couche OCC. La contrainte posée par ces langages est qu'un utilisateur doit utiliser les concepts canoniques représentés dans cette couche pour manipuler les données. Il ne peut pas représenter sa propre perception du domaine et utiliser cette représentation pour manipuler les données. Ce problème est considéré dans la section suivante.

4 Exploitation des données à base ontologique au niveau ontologique. Application à la couche OCNC

La couche OCNC permet de représenter des points de vue particuliers sur un domaine décrit par une OCC. Elle est constituée de concepts non canoniques définis à partir des concepts de la couche OCC et/ou d'autres concepts de la couche OCNC. Pour créer des concepts non canoniques, différents constructeurs ont été proposés dans les modèles d'ontologies. A notre connaissance, aucune BDBO actuelle ne permet de représenter les concepts définis avec ces différents constructeurs. Ce problème de recherche fait l'objet d'une thèse au laboratoire LISI [Fankam, 2007]. Compte tenu de l'envergure du sujet, notre objectif dans le contexte du langage OntoQL a été, dans une étape préliminaire à ces travaux, de définir des mécanismes nécessaires pour manipuler les concepts issus des OCNC.

4.1 Problématique

Le principal problème posé par la définition des concepts non canoniques est que les constructeurs fournis par les modèles d'ontologies sont divers. Les constructeurs des modèles d'ontologies considérés permettant de définir des concepts non canoniques sont présentés dans le tableau 3.8. Ces constructeurs peuvent être regroupés en trois catégories selon leur origine :

Constructeurs	Modèle d'ontologies
Construction de classes par des expressions booléennes (Union, Intersection, Complément)	OWL
Construction de classes comme des restrictions (de domaine, de valeur ou de cardinalité)	OWL
Définition des propriétés inverses	OWL
Règles logiques	F-Logic
Expression algébrique	PLIB (en cours)

TAB. 3.8 – Constructeurs non canoniques proposés par les modèles d'ontologies

– les constructeurs de classes et de propriétés définis. Ces constructeurs sont issus de la logique de description. Les expressions logiques liées à ces constructeurs (par exemple, les expressions booléennes) peuvent être exploitées par un moteur d'inférence, aussi appelé raisonneur (par exemple, Racer [Haarslev and Möller, 2001]), afin, notamment, d'organiser les concepts canoniques et non canoniques dans une seule et même hiérarchie ;
– les règles logiques. Ces constructeurs sont issus de la logique des frames. Un moteur de règles (par exemple, Jess[25]) les exploite pour déduire de nouveaux faits à partir des faits connus ;
– les expressions algébriques (exemple, diamètre = rayon * 2). Ces constructeurs sont issus de la communauté du traitement des données. Ils nécessitent un interpréteur d'expressions algébriques.

Cette diversité d'approches soulève une question : quels types de constructeurs doivent être utilisés pour définir les concepts non canoniques d'une ontologie ? Cette question a soulevé un débat entre les différentes communautés [Motik et al., 2006, de Bruijn et al., 2005, Patel-Schneider and Horrocks, 2006]. De notre point de vue, ce débat a montré que les différents constructeurs proposés ont chacun leur utilité. Ils sont plus ou moins adaptés selon le domaine d'application dans lesquels ils sont utilisés. L'objectif est donc maintenant de proposer une architecture de BDBO suffisamment flexible pour permettre d'utiliser ces différents constructeurs.

Une des approches envisagées pour représenter ces constructeurs est d'utiliser le mécanisme de vue. Nous avons donc étudié l'enrichissement d'OntoQL par un langage de définition de vues.

4.2 Langage de Définition de Vues (LDV)

Pour définir ce langage nous avons suivi la même approche que pour les autres langages qui constituent OntoQL, c'est-à-dire s'inspirer du langage de vues défini par SQL pour les types utilisateurs. Dans ce cas, il est nécessaire d'adapter ce langage par rapport au besoin de la problématique présentée dans la section précédente.

4.2.1 Modèle de données du niveau ontologique, couche OCNC

En SQL, les instances d'un type utilisateur peuvent être stockées, non seulement dans des tables typées, mais également dans des *vues typées* construites à partir d'une requête SQL. Comme pour les tables typées, l'héritage entre vues typées est autorisé. Une vue typée doit être associée à un type utilisateur, sous-type direct de celui associé à sa super-vue. Les instances des vues typées ont un identifiant qui peut être soit celui d'une instance à partir de laquelle elle a été calculée, soit un identifiant spécifié par l'utilisateur. Les mêmes règles que pour les vues traditionnelles sont utilisées pour déterminer si une vue typée peut être mise à jour.

SQL impose les contraintes suivantes sur les vues typées :

1. la requête associée à une vue typée ne doit porter que sur une seule table (ou vue) typée nommée sa *table de base*. Ceci implique donc d'utiliser l'opérateur `ONLY` dans cette requête ;
2. une vue typée doit être associée au même type utilisateur que sa table de base ;

[25] http://herzberg.ca.sandia.gov/jess/

3. une vue typée ne peut pas hériter d'une table typée. La hiérarchie de tables typées et de vues typées est ainsi clairement distinguée.

Les contraintes 1 et 2 ont été définies pour assurer la cohérence des vues typées définies. En particulier, imposer qu'une requête d'une vue typée porte sur un seul niveau de la hiérarchie, a pour but que les instances d'une vue typée et de chacune de ses super-vues soient disjointes. La contrainte 3 a été définie pour respecter le principe d'indépendance logique défini dans la norme ANSI/SPARC qui requiert principalement que les vues soient indépendantes du schéma à partir duquel elles sont définies.

Cependant, ces contraintes sont très contraignantes par rapport aux besoins exprimés dans la problématique (cf. section 4.1) :

– les instances des classes non canoniques peuvent être dérivées à partir des instances de plusieurs classes. C'est le cas des classes non canoniques construites à partir d'une expression booléenne. Les contraintes 1 et 2 empêchent de telles définitions ;
– la hiérarchie de classes d'une ontologie peut être constituée de classes canoniques et non canoniques. La contrainte 3 impose de définir deux hiérarchies distinctes.

Ces contraintes doivent donc être levées. Sans les contraintes 1 et 2, les vues peuvent être définies avec beaucoup plus de liberté, le risque étant de définir des vues incohérentes. Nous avons choisi de laisser ce degré de liberté en notant que si un moteur d'inférence est couplé à une BDBO, ce dernier pourra vérifier automatiquement la cohérence de l'ontologie obtenue après enrichissement par les concepts OCNC.

La contrainte 3 a été définie pour respecter le principe d'indépendance logique défini dans la norme ANSI/SPARC. Or, dans notre cas, nous ne considérons plus le mécanisme de vue au niveau logique mais au niveau ontologique. A ce niveau, notre but est d'utiliser des vues pour représenter les concepts non canoniques d'une ontologie. Ces concepts non canoniques ne doivent pas être indépendants des concepts canoniques. En conséquence, nous avons décidé de ne pas imposer la contrainte 3.

Sans ces contraintes, le modèle de données du langage de définition de vues de OntoQL est le suivant. Une classe non canonique peut être définie comme une classe canonique. Elle peut ainsi être le domaine de définition de nouvelles propriétés et peut être une super et/ou sous-classes d'autres classes (canoniques ou non canoniques). La seule différence avec une classe canonique est que son extension est définie comme une vue, c'est-à-dire à partir d'une requête OntoQL explicitement décrite par l'utilisateur. Ainsi, contrairement à une classe canonique dont les instances sont celles contenues dans son extension et dans celles de ses sous-classes canoniques, les instances d'une classe non canonique sont celles retournées par une requête.

Nous présentons maintenant les énoncés du langage permettant de définir, manipuler et interroger des vues conformes à ce modèle de données.

4.2.2 Définition, manipulation et interrogation des vues

Le LDV permet de définir, mettre à jour et interroger des classes dont l'extension est créée comme une vue.

4.2.2.1 Définition de vues

Pour permettre de préciser qu'une classe est non canonique, nous avons étendu la syntaxe de création d'une classe canonique (cf. section 3.3.1) de la manière suivante :

⟨*class definition*⟩ ::= `CREATE` ⟨*entity id*⟩ ⟨*class id*⟩ [⟨*view clause*⟩] [⟨*under clause*⟩]
[⟨*descriptor clause*⟩] [⟨*properties clause list*⟩]

⟨*view clause*⟩ ::= `AS VIEW`

Nous avons ajouté la clause optionnelle ⟨*view clause*⟩ à l'instruction de création d'une classe. Lorsque cette clause est présente, la classe créée est considérée comme non canonique et doit donc être associée à une extension dont les instances sont déterminées par une requête OntoQL. La syntaxe permettant de définir une telle extension est la suivante :

⟨*view definition*⟩ ::= `CREATE VIEW OF` ⟨*class id*⟩ [⟨*property id list*⟩] `AS` ⟨*query expression*⟩

Cette instruction permet de créer l'extension de la classe ⟨*class id*⟩ à partir de la requête ⟨*query expression*⟩. La liste des propriétés utilisées dans cette extension peut être indiquée explicitement (⟨*property id list*⟩) ou déduite de la requête.

Exemple. Créer la classe `PostDupont` qui regroupe les messages écrits par l'utilisateur `Dupont`.

```
CREATE #Class PostDupont AS VIEW UNDER Post

CREATE VIEW OF PostDupont AS
  SELECT * FROM Post AS p WHERE p.has_creator.last_name = 'Dupont'
```

Explication. La première instruction permet de créer la classe `PostDupont`. Le placement d'une classe non canonique dans la hiérarchie de classes est à la tâche de l'utilisateur. Compte tenu de sa définition, la classe `PostDupont` est une classe non canonique subsumée par la classe `Post`. Ceci est indiqué par les expressions `AS VIEW` et `UNDER Post`.

La seconde instruction permet de créer l'extension de cette classe à partir d'une requête OntoQL. Cette requête retourne les instances de la classe canonique `Post` ayant pour auteur `Dupont`. Ces instances sont celles de la classe non canonique `PostDupont`.

4.2.2.2 Mise à jour de vues

La mise à jour des instances d'une classe non canonique ramène au problème de la mise à jour des vues. Le langage de manipulation de données peut donc être utilisé pour les classe non canoniques si et seulement si la vue correspondante à son extension peut être mise à jour.

Exemple. Supprimer les messages de l'utilisateur `Dupont` dont le titre commence par `SIOC`.

```
DELETE FROM PostDupont WHERE title LIKE 'SIOC%'
```

Explication. Cette instruction porte sur la classe non canonique `PostDupont`. Elle supprime des instances de la classe canonique `Post` à partir de laquelle la classe `PostDupont` est construite. Les instances supprimées sont celles dont l'auteur est `Dupont` (condition de la définition de la classe `PostDupont`) et dont le titre ne commence par `SIOC` (condition de l'instruction).

4.2.2.3 Interrogation de vues

Au niveau de la syntaxe du langage, les requêtes portant sur des classes non canoniques sont identiques à celles portant sur des classes canoniques. Par contre, l'interprétation de ces requêtes est différente. Le résultat d'une requête polymorphe portant sur une classe non canonique est déterminé en remplaçant le nom de cette classe par la requête permettant d'en calculer l'extension (*réécriture de requête*). Pour les requêtes non polymorphes, la différence (`EXCEPT`) entre les instances retournées par la requête de définition de la classe non canonique et les instances de ses sous-classes doit être réalisée.

Exemple. Rechercher les messages écrits par l'utilisateur `Dupont`.

```
SELECT * FROM PostDupont
```

Explication. Le résultat de cette requête est déterminé en exécutant la requête :

```
SELECT *
  FROM (SELECT * FROM Post AS p WHERE p.has_creator.last_name = 'Dupont')
```

Le LDV permet ainsi de calculer l'extension d'une classe non canonique à partir de celles des autres classes. Nous présentons maintenant ses capacités et ses limitations par rapport à notre problématique de départ.

4.3 Utilisation

Le langage de définition de vues peut être utilisé pour représenter des classes et des propriétés non canoniques définies à partir de certains constructeurs décrits dans le tableau 3.8.

4.3.1 Définition de classes non canoniques

Des classes non canoniques peuvent être définies par des expressions booléennes ou des restrictions de OWL.

4.3.1.1 Définition de classes non canoniques par des expressions booléennes

Les classes non canoniques définies par des expressions booléennes peuvent être créées en utilisant les opérateurs `UNION`, `INTERSECT` et `EXCEPT`. Nous montrons ceci dans les exemples suivants. Pour simplifier ces exemples, nous ne présentons pas les instructions de création des classes non canoniques. Nous indiquons uniquement l'instruction de création de leur extension. Les classes non canoniques créées dans ces exemples sont représentées par un diagramme UML sur la figure 3.3. La hiérarchie de classes est présentée en haut de cette figure. Nous avons utilisé un stéréotype pour distinguer les classes canoniques («`OCC`») des classes non canoniques («`OCNC`»). Enfin, les définitions OWL des classes non canoniques créées dans ces exemples sont indiquées en bas de la figure en utilisant la syntaxe OWL décrite au chapitre 1, section 3.2.1.

Exemple. Supposons l'existence de la classe `PostNonDupont`, sous-classe de la classe `Post`. Créer l'extension de cette classe définie comme étant le complémentaire de la classe `PostDupont` (définie dans la section précédente) par rapport à la classe `Post`.

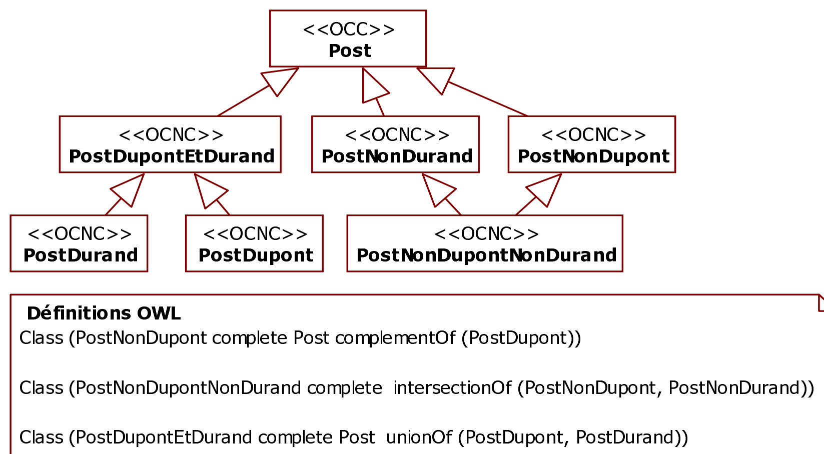


FIG. 3.3 – Exemples de création de classes non canoniques par des expressions booléennes de OWL

```
CREATE VIEW OF PostNonDupont AS
  (SELECT * FROM Post) EXCEPT (SELECT * FROM PostDupont)
```

Explication. Cette instruction crée l'extension de la classe `PostNonDupont` en réalisant la différence (`EXCEPT`) entre les instances de la classe `Post` et celles de la classe `PostDupont`.

Exemple. Supposons l'existence d'une classe `PostNonDurand` dont la définition est identique à celle de la classe `PostNonDupont`, de l'exemple précédent, pour l'utilisateur `Durand`. Soit `PostNonDupont-NonDurand` une classe, sous-classe des classes `PostNonDupont` et `PostNonDurand`.

Créer l'extension de cette classe définie comme étant les messages qui n'ont pas été écrits ni par l'utilisateur `Dupont`, ni par l'utilisateur `Durand`.

```
CREATE VIEW OF PostNonDupontNonDurand AS
  (SELECT * FROM PostNonDupont) INTERSECT (SELECT * FROM PostNonDurand)
```

Explication. Cette instruction définit l'intersection (`INTERSECT`) des instances des classes `PostNon-Dupont` et `PostNonDurand` comme étant les instances de la classe `PostNonDupontNonDurand`.

Exemple. Supposons l'existence d'une classe `PostDupontEtDurand`, sous-classe de la classe `Post` et super-classes des classes `PostDupont` et `PostDurand` créées comme dans les exemples précédents. Créer l'extension de cette classe définie comme étant les messages qui ont été écrits, soit par l'utilisateur `Dupont`, soit par l'utilisateur `Durand`.

```
CREATE VIEW OF PostDupontEtDurand AS
  (SELECT * FROM PostDupont) UNION (SELECT * FROM PostDurand)
```

Explication. L'opérateur `UNION` est utilisé pour définir les instances de `PostDupontEtDurand` à partir de celles de ces deux sous-classes.

4.3.1.2 Définition de classes non canoniques par des restrictions

Le langage de définition de vues permet également de représenter les classes non canoniques définies par des restrictions OWL. Trois types de restriction sont disponibles : les restrictions de valeur, les

restrictions de domaine et les restrictions de cardinalité. Les exemples suivants présentent la création de classes non canoniques en utilisant ces constructeurs. Ces exemples sont illustrés sur la figure 3.4 de la même manière que pour les exemples de la section précédente.

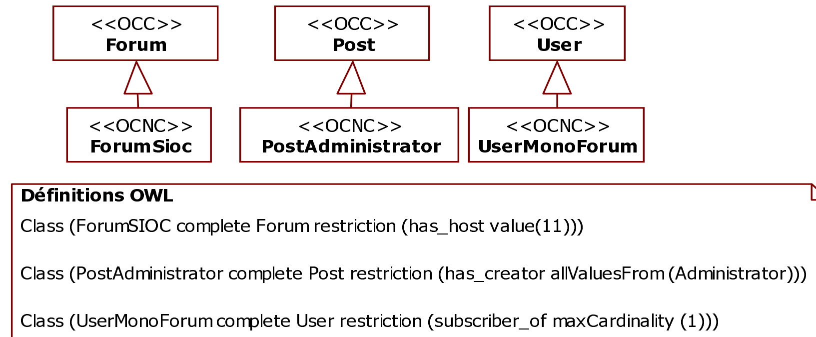


FIG. 3.4 – Exemples de création de classes non canoniques par des restrictions OWL

Exemple. Supposons l'existence de la classe `ForumSioc`, sous-classe de la classe `Forum`. Créer l'extension de cette classe définie comme étant les forums qui sont hébergés par le site de nom `Sioc` (son identifiant interne est `11`).

```
CREATE VIEW OF ForumSioc AS
  SELECT * FROM Forum WHERE has_host = 11
```

Explication. La classe `ForumSioc` est une restriction de valeur portant sur la propriété `has_host` dont la valeur doit être `11`.

Exemple. Supposons l'existence de la classe `PostAdministrator`, sous-classe de la classe `Post`. Créer l'extension de cette classe définie comme étant les messages qui ont été écrits par un administrateur (nous supposons une classe `Administrator` sous-classe de `User`).

```
CREATE VIEW OF PostAdministrator AS
  SELECT * FROM Post AS p WHERE p.has_creator IS OF REF(Administrator)
```

Explication. La classe `PostAdministrator` est une restriction de domaine portant sur la propriété `has_creator` dont les valeurs doivent être prises dans la classe `Administrator`. La requête qui permet d'en construire l'extension utilise l'opérateur `IS OF` pour tester que la référence retournée par la propriété `has_creator` porte sur une instance de la classe `Administrator`.

Le dernier type de restriction présenté est celui de restriction de cardinalité.

Exemple. Supposons l'existence de la classe `UserMonoForum`, sous-classe de la classe `User`. Créer l'extension de cette classe définie comme étant les utilisateurs qui ne sont abonnés qu'au plus à un forum.

```
CREATE VIEW OF UserMonoForum AS
  SELECT * FROM User AS u WHERE CARDINALITY(u.subscriber_of) <= 1
```

Explication. La classe `UserMonoForum` est une restriction de cardinalité portant sur la propriété `subscriber_of` qui ne doit prendre qu'au maximum une valeur pour une instance donnée. La requête qui permet d'en construire l'extension utilise la fonction SQL prédéfinie `CARDINALITY` qui permet de retourner le nombre d'éléments d'un tableau.

4.3.2 Définition de propriétés non canoniques

Considérons maintenant le cas des propriétés non canoniques. La valeur d'une propriété définie comme étant l'inverse d'une autre peut être calculée dans une vue.

Exemple. Supposons l'existence de la classe V_USER, sous-classe de la classe User qui définit la propriété creator_of indiquant les messages dont un utilisateur est le créateur (inverse de la propriété has_creator). Créer l'extension de cette classe.

```
CREATE VIEW OF V_USER AS
  SELECT *, ARRAY (SELECT p.oid
                     FROM Post AS p
                    WHERE p.has_creator = u.oid)
  FROM User AS u
```

Explication. Un utilisateur pouvant créer plusieurs Post, la propriété creator_of est de type tableau de références vers des instances de la classe Post. Pour en calculer la valeur, le constructeur de tableau prenant une requête en paramètre est utilisé. Pour chaque utilisateur, cette requête recherche les identifiants des messages dont il est l'auteur (p.has_creator = u.oid).

Le langage OntoQL supportant les opérateurs arithmétiques, il permet également de représenter les propriétés dont la valeur peut être calculée par une expression utilisant ces opérateurs.

Exemple. Supposons l'existence de la classe V_Post, sous-classe de la classe Post. Cette classe définit la propriété note_sur_5 qui peut être calculée à partir de la propriété note de la classe Post. Créer l'extension de cette classe.

```
CREATE VIEW OF V_POST AS
  SELECT *, note / 4 FROM Post
```

Explication. Cette instruction suppose que les valeurs de la propriété note sont données sur 20. Elle utilise l'opérateur de division pour retourner la note sur 5.

Dans cette section, nous venons de montrer que le LDV permet de représenter des classes et des propriétés non canoniques définies avec différents constructeurs des modèles d'ontologies. Dans la section suivante, nous présentons les limites de ses capacités.

4.3.3 Limitations du langage OntoQL pour définir des concepts non canoniques

Par rapport à la problématique globale, décrite dans la section 4.1, le langage OntoQL présente les limitations suivantes :

– le placement des classes non canoniques dans la hiérarchie de classes doit être réalisé par l'utilisateur. Dans les exemples présentés, nous avons précisé la place de la classe non canonique créée dans la hiérarchie de classes en considérant sa requête de définition. Actuellement, ceci n'est pas automatisé ;
– les propriétés non canoniques ne peuvent pas être définies sur des classes canoniques ;

– la mise à jour des classes non canoniques n'est pas toujours possible.

Des travaux, consistant à étendre ces mécanismes ou à en introduire de nouveaux pour proposer des solutions à ces limitations, sont en cours. Notons que ces travaux pourront se baser sur les résultats obtenus dans le contexte des BDOO [Guerrini et al., 1997, Abiteboul and Bonner, 1991, Rundensteiner, 1992, dos Santos et al., 1994] où les problèmes de placement automatique des vues et de mise à jour de vues ont été abordés. Un autre travail a considéré est celui proposé dans [Volz et al., 2003] qui aborde le problème du placement automatique des classes non canoniques dans le cas où les opérateurs utilisés sont ceux de OWL.

5 Exploitation des données à base ontologique au niveau ontologique. Application à la couche OL

Jusqu'à présent, nous avons montré que les classes et les propriétés d'une ontologie étaient désignées et manipulées dans les instructions OntoQL en utilisant leur identifiant. Si dans nos exemples, ces identifiants sont proches des mots du langage naturel, ce n'est pas le cas dans de nombreuses ontologies. Par exemple, dans l'ontologie PLIB IEC décrivant des composants électroniques [IEC61360-4, 1999], l'identifiant de la classe des résistances est `AAA089`. En conséquence, écrire une requête OntoQL avec ces codes n'est pas aisé et va à l'encontre de l'exigence 12. En conséquence, nous avons décidé que le langage OntoQL devait permettre d'utiliser les noms définis dans la couche linguistique d'une ontologie. Ceci nécessite d'étendre le modèle de données du niveau ontologique défini précédemment ainsi que les énoncés du langage.

5.1 Modèle de données du niveau ontologique, couche OL

Étendre le modèle de données du niveau ontologique pour prendre en compte la couche OL, consiste à déterminer les noms, associés à chaque classe et à chaque propriété, qui vont permettre de les référencer sans ambiguïté dans une instruction OntoQL. Pour une langue naturelle donnée, les modèles d'ontologies permettent d'associer plusieurs noms à chaque classe et à chaque propriété. Dans cet ensemble de noms, le modèle PLIB distingue un *nom préféré* des autres noms qualifiés de *noms synonymes*. Permettre d'utiliser l'ensemble des noms associés à une classe pour la référencer évite de devoir connaître le nom préféré qui a été choisi parmi un ensemble de noms synonymes possibles. Cependant, ceci a un coût engendré par le besoin de rechercher une classe non seulement par son nom préféré mais aussi par ses noms synonymes. Par exemple, dans la BDBO OntoDB, les noms préférés et synonymes sont stockés dans des tables distinctes. Et donc, rechercher une classe, par un de ses noms synonymes, nécessite d'effectuer des jointures supplémentaires. De plus, pour que cette recherche soit efficace, les classes et les propriétés doivent être indexées par cet ensemble de noms synonymes ce qui engendre un coût de stockage supplémentaire. En conséquence, seuls les noms préférés des classes et des propriétés peuvent être utilisés dans une instruction OntoQL. Pour préciser ces noms, nous avons introduit l'attribut `#name` défini sur les entités `#Class` et `#Property`. Ces noms pouvant être donnés dans plusieurs langues naturelles, nous avons défini un type de données *chaîne de caractères multilingue* (`MULTILINGUAL STRING`) pour représenter le codomaine de cet attribut. Dans la plupart des modèles d'ontologies, les valeurs de propriétés peuvent

également être représentées dans différentes langues naturelles. Le type MULTILINGUAL STRING peut également être utilisé comme codomaine d'une propriété.

L'utilisation des noms des classes et des propriétés dans une requête OntoQL ne doit pas introduire d'ambiguïté. Le point d'entrée d'une requête est la clause FROM dans laquelle sont précisés des noms de classes associés à des espaces de noms. Si un de ces noms fait référence à plusieurs classes dans son espace de noms, alors cette requête est ambiguë. Les noms des classes doivent donc être uniques pour un espace de noms donné. Cependant, si on considère le multilinguisme, cette contrainte est très forte. En effet, une classe peut avoir le même nom dans différentes langues naturelles (par exemple, la classe Site). De plus, un même nom peut exister dans différentes langues naturelles et faire référence à des classes différentes (*faux-amis*). Par exemple, le mot Location peut représenter deux classes différentes suivant que ce mot est considéré en anglais ou en français. En conséquence, nous avons affaibli la contrainte précédente en exigeant que les noms des classes ne soient uniques que par rapport à un espace de noms et à une langue naturelle donnés. Pour qu'une requête ne soit pas ambiguë malgré l'affaiblissement de cette contrainte, la langue naturelle utilisée doit pouvoir être précisée dans une instruction OntoQL (cf. section 5.2). Si cette contrainte n'est pas respectée, une erreur, à l'exécution de l'instruction, est levée.

Dans une requête OntoQL, les propriétés utilisées doivent être applicables à une des classes de la clause FROM. Si c'est le cas pour plusieurs de ces classes, la requête est ambiguë. Comme pour le langage SQL, cette ambiguïté doit être levée en indiquant explicitement la classe par rapport à laquelle la propriété ambiguë doit être considérée. Cependant, si plusieurs propriétés de même nom pouvaient être applicables sur une classe, l'ambiguïté pourrait subsister. En conséquence, pour un espace de noms, une langue naturelle et une classe donnés, les noms des propriétés applicables sur cette classe doivent être uniques.

Notons que si nous avions permis l'expression de requêtes non typées (permises dans les langages comme SPARQL ou RQL), c'est-à-dire qui ne précise pas les classes dans lesquelles les instances doivent être recherchées, nous aurions dû imposer sur les propriétés la même contrainte que sur les classes. Cette contrainte aurait empêchée que deux propriétés différentes soient définies avec le même nom dans une ontologie donnée. C'est pourtant nécessaire dans bien des cas. Par exemple, dans le domaine de la mécanique, la longueur d'une vis ne se calcule pas comme la longueur d'un vérin. En conséquence, dans une ontologie, il est nécessaire de définir deux propriétés dont le nom est longueur afin de représenter ce qu'est la longueur de ces deux composants. De cette manière, ces deux propriétés peuvent être associées à des descriptions (document, graphique, etc.) différentes et même éventuellement à des unités de mesure différentes. Le langage OntoQL permet de telles définitions, fréquentes dans nos domaines d'application cibles, au détriment de l'expression de requêtes non typées peu fréquentes dans ces mêmes domaines.

Ainsi, nous avons étendu le modèle de données du niveau ontologique pour permettre de manipuler les classes et les propriétés en utilisant la couche linguistique. Nous présentons maintenant les mécanismes syntaxiques nécessaires pour que le langage OntoQL permette d'exploiter cette couche.

5.2 Aspects syntaxiques

Précédemment, nous avons indiqué que les classes et les propriétés pouvaient être manipulées directement en utilisant leur identifiant. La syntaxe du langage doit maintenant permettre d'utiliser les noms préférés de ces classes et de ces propriétés. Afin de distinguer les identifiants des noms préférés,

nous avons utilisé le fait que ces derniers ne sont définis que par rapport à une langue naturelle donnée. Ainsi, si une langue naturelle est indiquée, les éléments d'une requête sont considérés comme des noms préférés, sinon, ils sont considérés comme des identifiants.

Pour indiquer la langue naturelle utilisée, nous avons utilisé des mécanismes similaires à ceux définis pour les espaces de noms. Ainsi, une langue naturelle peut être indiquée localement à une instruction (clause `USING LANGUAGE`) ou globalement (`SET LANGUAGE`). La syntaxe de ces deux mécanismes est la suivante :

⟨*language clause*⟩ ::= `USING LANGUAGE` ⟨*language id*⟩
⟨*global namespace definition*⟩ ::= `SET LANGUAGE` ⟨*language specification*⟩
⟨*language specification*⟩ ::= ⟨*language id*⟩ | `NONE`

Dans ces clauses, ⟨*language id*⟩ est un code à deux caractères identifiant la langue naturelle utilisée. Ces codes sont répertoriés dans la norme ISO 639-1 [ISO639-1, 2002]. Contrairement aux mécanismes introduits pour les espaces de noms, une seule langue naturelle peut être utilisée par instruction. En effet, si exprimer des requêtes utilisant des éléments de différentes ontologies est particulièrement utile, nous avons considéré qu'exprimer une requête en utilisant plusieurs langues naturelles augmenterait inutilement la complexité du langage.

Nous avons montré comment le langage OntoQL permet d'utiliser à la fois les identifiants et les noms pour référencer les classes et les propriétés dans une instruction. Précisons maintenant la syntaxe utilisée pour les noms. La particularité d'un nom par rapport à un identifiant est qu'il peut comporter des espaces. Pour permettre d'utiliser un nom présentant cette caractéristique, nous avons suivi l'approche proposée par SQL qui consiste à l'entourer par des apostrophes (").

Les mécanismes syntaxiques permettant d'utiliser les noms préférés des classes et des propriétés étant précisés, nous présentons maintenant comment cette capacité est exprimée et exploitée dans les langages.

5.3 Langage de définition, de manipulation et d'interrogation de données

Le langage de définition de données doit permettre de définir des éléments conformes au modèle de données présenté précédemment. Il doit donc permettre d'associer des noms préférés aux classes et aux propriétés dans différentes langues naturelles en vérifiant les contraintes que nous avons définies sur ce modèle de données. Les instructions du LDD permettent d'indiquer ces noms dans les `DESCRIPTOR` en indiquant la valeur de l'attribut `#name`. Cet attribut est de type `MULTILINGUAL STRING`. Ce type de données est associé à une opération permettant de retrouver la chaîne de caractère correspondant à une valeur de ce type pour une langue naturelle donnée. Dans le langage, la syntaxe employée pour représenter cette opération est [⟨*language id*⟩]. L'exemple suivant illustre l'utilisation de cette syntaxe.

Exemple. Créer la classe `User` de l'ontologie SIOC en indiquant son nom préféré ainsi que celui de ses propriétés en anglais et en français.

```
CREATE #Class User (
  DESCRIPTOR (
    #name[FR] = 'Utilisateur'
```

```
    #definition = 'An online account (User) in an online community site.'
    #definition[FR] = 'Un compte utilisateur d''une communauté en-ligne')
  #Property (
    "first name" String DESCRIPTOR (#name[FR] = 'prénom'),
    "last name" String DESCRIPTOR (#name[FR] = 'nom'),
    "email" String DESCRIPTOR (#name[FR] = 'email'),
    "subscriber_of" REF(Forum) ARRAY  DESCRIPTOR (#name[FR] = 'abonné à'))
)
USING LANGUAGE EN
```

Explication. Le nom anglais de la classe `User` est indiqué dans l'en-tête de l'instruction. Son nom en français (`FR`) est indiqué dans la clause `DESCRIPTOR`. Nous avons également indiqué les valeurs de l'attribut `#definition` qui permet de décrire cette classe (attribut `comment` en RDF-Schema, `definition` en PLIB). Cet attribut est également du type `MULTILINGUAL STRING`. Pour définir sa valeur en anglais, il n'est pas nécessaire de préciser cette langue car la clause `USING LANGUAGE` la définit comme celle par défaut. Les noms préférés des propriétés sont définis de la même façon en utilisant une clause `DESCRIPTOR` pour chacune d'elles.

Au niveau du langage de manipulation et d'interrogation de données, l'utilisation de la couche linguistique permet d'exprimer des instructions dans différentes langues naturelles comme le montre l'exemple suivant.

Exemple. Rechercher le prénom, le nom et l'adresse email des utilisateurs en utilisant une requête écrite avec les identifiants externes, une autre avec les noms en anglais et une dernière avec les noms en français.

```
SELECT first_name,   <=>   SELECT "first name",   <=>   SELECT prénom,
       last_name,                 "last name",                 nom,
  FROM User                  FROM User                   FROM Utilisateur
                            USING LANGUAGE EN           USING LANGUAGE FR
       (A)                        (B)                          (C)
```

Explication. La requête `(A)` n'utilise pas la couche linguistique d'une ontologie. Elle est écrite avec les identifiants des classes et propriétés. La requête `(B)` est équivalente à la requête `(A)` mais est écrite en utilisant les noms préférés anglais (`EN`) des classes et propriétés. Les apostrophes sont utilisées pour les noms comportant un espace comme par exemple `"first name"`. La requête `(C)` est également équivalente aux deux précédentes mais elle est écrite en utilisant les noms français (`FR`).

Le type `MULTILINGUAL STRING` pouvant également être utilisé comme codomaine d'une propriété, les valeurs des propriétés peuvent être recherchées dans différentes langues naturelles.

Exemple. Rechercher les titres en anglais et en français des messages.

```
SELECT title[FR], title[EN]
  FROM Post
```

Explication. Cet exemple suppose que la propriété `title` est de type `MULTILINGUAL STRING`. La valeur de cette propriété est recherchée en français (`[FR]`) et en anglais (`[EN]`).

6 Conclusion

Dans ce chapitre, nous avons présenté les traitements offerts par le langage OntoQL pour permettre d'exploiter les données à base ontologique d'une BDBO. Ces traitements peuvent s'exprimer par rapport à quatre niveaux de référence de l'architecture ANSI/SPARC étendue.

Le langage OntoQL permet d'abord d'exploiter les données à base ontologique au niveau logique (c'est-à-dire, au niveau des relations). En effet, ce langage est compatible avec un sous-ensemble de SQL communément supporté par les SGBD usuels. De cette manière, il reste simple à appréhender. Les traitements au niveau logique peuvent être utiles pour optimiser les traitements effectuées au niveau ontologique, pour améliorer l'implantation du niveau ontologique ou pour prendre en compte un schéma de base de données existant (indexation sémantique).

Le langage OntoQL permet également d'exploiter les données à base ontologique au niveau ontologique en faisant référence seulement à ses concepts canoniques, et donc, indépendamment de la représentation logique des données. Souhaitant proposer une syntaxe uniforme pour les différents niveaux d'accès à l'architecture ANSI/SPARC étendue, nous nous sommes appuyés, pour définir le modèle de données et les opérateurs de cette couche, sur ceux fournis par SQL. Nous les avons cependant adaptés aux particularités du niveau ontologique. Le langage OntoQL résultant conserve une syntaxe proche de SQL pour accéder au niveau ontologique et intègre les principaux opérateurs de SQL, y compris des opérateurs orientés-objets introduits dans la norme SQL99.

Les traitements au niveau ontologique des données à base ontologique peuvent également être réalisés à partir de la couche OCNC d'une ontologie. En effet, le langage OntoQL permet de définir une classe dont l'extension est définie comme une vue, c'est-à-dire calculée à partir des instances des autres classes. De cette manière, ces classes non canoniques peuvent être interrogées comme des classes canoniques, le traitement de ces requêtes étant réalisées par réécriture de requêtes.

Enfin, les traitements au niveau ontologique des données à base ontologique peuvent être réalisés à partir de la couche OL d'une ontologie. Cette couche associant aux différents concepts ontologiques des noms éventuellement donnés dans différentes langues naturelles, le langage OntoQL permet de les utiliser pour exprimer des requêtes. Ainsi, une requête peut être exprimée dans différentes langues naturelles. La langue naturelle utilisée peut être spécifiée dans une requête OntoQL ou définie comme un paramètre du langage.

Ainsi, par rapport aux langages de bases de données traditionnelles, le langage OntoQL permet d'exploiter les données à base ontologique indépendamment de leur représentation au niveau logique. Si les langages de bases de données traditionnelles se focalisent essentiellement sur l'exploitation des données d'une base de données, nous avons vu lors de la définition des exigences que les BDBO nécessitent également (1) de pouvoir exploiter les ontologies qu'elles contient et (2) de pouvoir interroger simultanément les ontologies et les données à base ontologique. Le langage OntoQL permet d'effectuer ces traitements spécifiques aux BDBO. Nous les présentons dans le chapitre suivant.

Chapitre 4

Traitements des ontologies et simultanément des ontologies et des données d'une BDBO

Sommaire

1 Introduction 135

2 Exploitation des ontologies d'une BDBO 135

2.1 Modèle de données permettant de représenter les ontologies 136

2.2 Aspects syntaxiques 139

2.3 Langage de Définition des Ontologies (LDO) 140

2.4 Langage de Manipulation des Ontologies (LMO) 142

2.5 Langage d'Interrogation des Ontologies (LIO) 145

3 Interrogation conjointe des ontologies et des données 149

3.1 Des ontologies vers les données à base ontologique 149

3.2 Des données à base ontologique vers les ontologies 152

4 Analyse critique du langage OntoQL et perspectives d'évolution 153

4.1 Analyse du langage OntoQL par rapport aux exigences définies 153

4.2 Perspectives d'évolution du langage OntoQL 157

5 Conclusion 159

Résumé. Dans le chapitre précédent, nous avons présenté un premier aspect du langage OntoQL, celui qui permettait l'exploitation des données à base ontologique lorsque le modèle logique de ces données et/ou les ontologies qui les décrivent étaient connus de l'utilisateur. Dans ce chapitre, nous présentons un second aspect de ce langage, à savoir la possibilité d'exploiter les ontologies stockées dans une BDBO ainsi que celle de réaliser des requêtes portant à la fois sur les ontologies et les données à base ontologique. En effet, le langage OntoQL permet de définir, manipuler et interroger les ontologies d'une BDBO sans être spécifique d'un modèle d'ontologies particulier grâce au fait qu'il est basé sur un modèle d'ontologies noyau et que celui-ci peut être étendu par le langage OntoQL lui-même. De plus, pour permettre d'interroger à la fois les ontologies et les données à base ontologique d'une BDBO, le langage OntoQL permet d'utiliser des classes et des propriétés déterminées dynamiquement à l'exécution de la requête. Ceci permet de rechercher simultanément

les classes et les propriétés ainsi que leurs instances et leurs valeurs de propriétés (ontologies vers données). Il introduit également un opérateur permettant d'obtenir la description ontologique d'une instance (données vers ontologies).

1 Introduction

Les langages d'exploitation de bases de données usuelles se focalisent sur l'exploitation des données lorsque leur modèle logique est connu de l'utilisateur. Dans le chapitre précédent, nous avons montré que le langage OntoQL permet d'exploiter les données à base ontologique au niveau logique d'une BDBO comme les langages usuels de bases de données mais également au niveau ontologique lorsque les ontologies qui décrivent les données sont connues de l'utilisateur. Ces traitements, s'ajoutant à ceux proposés par les langages usuels de bases de données, constituent un premier aspect du langage OntoQL.

Dans ce chapitre, nous présentons un second aspect de ce langage, à savoir la possibilité d'exploiter les ontologies stockées dans une BDBO ainsi que celle de réaliser des requêtes portant à la fois sur les ontologies et les données à base ontologique. Ces traitements, qui s'éloignent de ceux proposés par les langages usuels de bases de données, permettent de découvrir les ontologies stockées dans une BDBO, d'extraire des ontologies en même temps que leurs instances et d'obtenir la description ontologique d'une instance.

Pour que le langage OntoQL permette d'exploiter les ontologies d'une BDBO sans être spécifique d'un modèle d'ontologies particulier (exigence 4), il a été fondé sur un modèle d'ontologies noyau contenant les constructeurs communs aux différents modèles d'ontologies identifiés au chapitre 1. Et, pour permettre de prendre en compte les spécificités d'un modèle d'ontologies particulier, ce modèle d'ontologies noyau peut être étendu en utilisant des instructions du langage de définition de données lui-même.

De plus, pour que le langage OntoQL permette d'interroger à la fois les ontologies et les données à base ontologique d'une BDBO (exigence 10), nous avons introduit des mécanismes permettant de rechercher les instances de classes et les valeurs de leurs propriétés lorsque ces classes et ces propriétés sont déterminées dynamiquement, à l'exécution de la requête (*ontologies vers données*). Ces mécanismes sont inspirés de ceux proposés par les langages conçus pour les bases de données fédérées tels que SchemaSQL [Lakshmanan et al., 2001] et MSQL [Litwin et al., 1989]. Nous avons également introduit un opérateur permettant d'obtenir la description ontologique d'une instance (*données vers ontologies*).

Ce chapitre est organisé comme suit. Les traitements proposés par le langage OntoQL pour définir, manipuler et interroger les ontologies d'une BDBO sont décrits dans la section suivante. Nous présentons ensuite, dans la section 3, les mécanismes que nous avons introduits pour permettre d'interroger à la fois les ontologies et les données d'une BDBO. Cette section termine la présentation du langage OntoQL. Nous présentons alors une analyse du langage OntoQL par rapport aux exigences définies dans le chapitre 2 ainsi que les perspectives d'évolution de ce langage dans la section 4. Elle est suivie par une conclusion générale sur la proposition du langage OntoQL.

2 Exploitation des ontologies d'une BDBO

Le langage OntoQL doit permettre d'exploiter les ontologies d'une BDBO sans être spécifique d'un modèle d'ontologies donné. Dans cette section, nous présentons le modèle de données ainsi que les langages que nous avons définis à cette fin.

2.1 Modèle de données permettant de représenter les ontologies

Le modèle de données permettant d'exploiter les ontologies stockées au niveau ontologique de l'architecture ANSI/SPARC étendue est constitué de l'ensemble des constructeurs du modèle d'ontologies utilisé. Or, ce modèle de données ne doit pas être spécifique d'un modèle d'ontologies particulier (exigence 4). Deux approches sont possibles. La première consiste à définir un modèle de données figé contenant l'ensemble des constructeurs des différents modèles d'ontologies. Le langage SOQA-QL a suivi cette approche. Cependant, vu la quantité et la diversité des constructeurs fournis par les modèles d'ontologies, seuls les constructeurs importants ont été inclus dans le modèle de données de SOQA-QL. En conséquence, cette approche ne permet pas de prendre en compte les constructeurs spécifiques d'un modèle d'ontologies donné (cf. chapitre 2, section 4.2.2).

La seconde approche est duale à la première. Elle consiste à définir un modèle de données extensible ne contenant initialement que les constructeurs partagés par les modèles d'ontologies. Cette approche n'est intéressante que lorsque l'ensemble des constructeurs partagés est important. Or, nous avons constaté dans le chapitre 1, section 3, que c'est le cas pour les modèles d'ontologies. Nous nous proposons donc de construire le langage OntoQL sur un *modèle d'ontologies noyau* contenant cet ensemble de constructeurs puis de définir des mécanismes permettant des extensions de ce modèle.

2.1.1 Le modèle d'ontologies noyau du langage OntoQL

La figure 4.1 présente les éléments de ce modèle d'ontologies noyau sous la forme simplifiée d'un modèle UML. Voici une description des éléments de ce modèle.

- Une ontologie (`Ontology`) définit un domaine d'unicité des noms (`namespace`). Elle regroupe la définition d'un ensemble de concepts : des classes et des propriétés.
- Une classe (`Class`) possède un identifiant interne à la BDBO (`oid`) et un identifiant indépendant de celle-ci (`code`). Sa définition comporte une partie textuelle (`name`, `definition`), éventuellement donnée dans plusieurs langues naturelles (`MultilingualString`). Ces classes sont organisées selon une structure de graphe acyclique qui représente les relations d'héritage multiple (`directSuperclasses`).
- Les propriétés (`Property`) possèdent également un identifiant (interne et externe) et une partie textuelle éventuellement définie dans plusieurs langues naturelles. Chacune des propriétés doit être définie sur la classe ou sur l'une des super-classes des instances qu'elle décrit (`scope`). Chaque propriété a un codomaine (`range`) qui permet de contraindre son domaine de valeurs.
- Le type de données (`Datatype`) d'une propriété peut être un type simple (`PrimitiveType`) tel que le type entier (`IntType`) ou le type chaînes de caractères (`StringType`). Une propriété peut également prendre ses valeurs dans une classe en faisant référence à ses instances (`RefType`). Il n'y a donc pas de composition. Enfin, cette valeur peut être une collection dont les éléments sont soit de type simple soit des références à des instances d'une classe (`CollectionType`).

Ce modèle noyau est soumis aux contraintes suivantes :

- le domaine et codomaine d'une propriété doivent toujours être définis et doivent être uniques ;

– les cycles dans la hiérarchie de subsomption sont interdits.

Nous avons introduit ces contraintes car elles sont en particulier définies dans le modèle PLIB. Notons qu'elles sont également introduites dans le modèle de données exploité par le langage RQL.

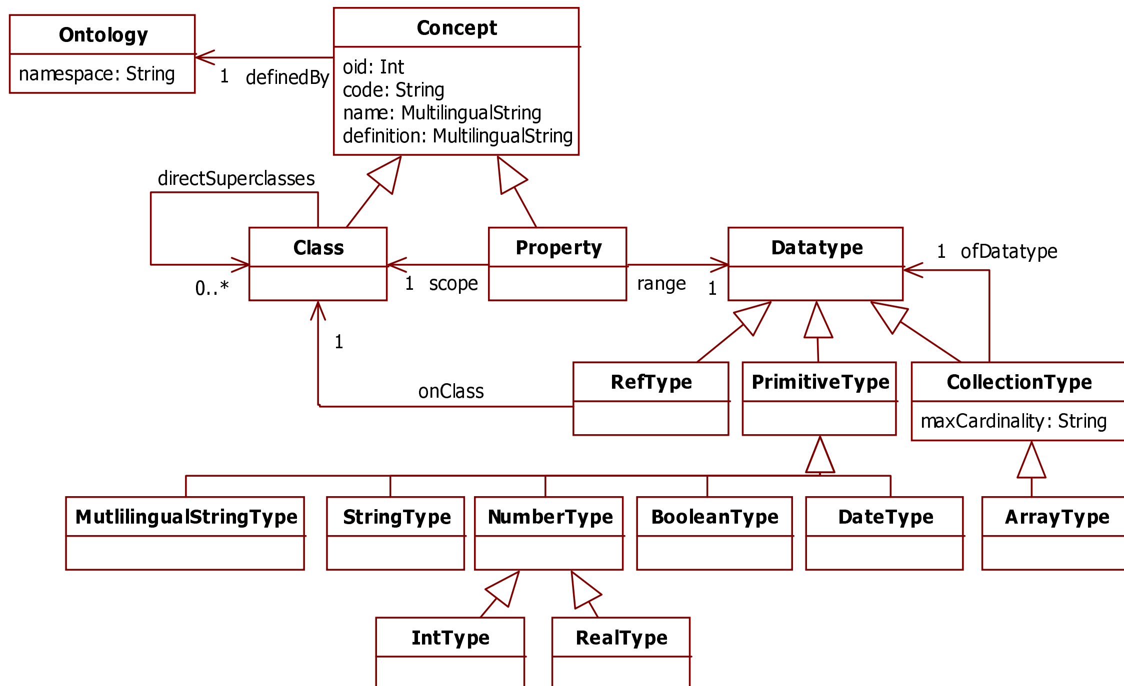


FIG. 4.1 – Le modèle d'ontologies noyau du langage OntoQL

2.1.2 Extensions du modèle d'ontologies noyau

L'extension du modèle noyau doit permettre d'ajouter des constructeurs proposés par un modèle d'ontologies particulier (dimension structurelle) et, autant que possible, permettre d'associer une sémantique à ces nouveaux constructeurs (dimension sémantique). Ces extensions doivent également préserver les contraintes que nous avons imposées sur le modèle noyau.

2.1.2.1 Dimension structurelle de l'extension du modèle noyau

Les constructeurs du modèle noyau étant représentés par des entités et des attributs, l'ajout de nouveaux constructeurs consiste à étendre le modèle noyau par de nouvelles entités et de nouveaux attributs.

Exemple. La figure 4.2 présente l'ajout des constructeurs de restrictions OWL `AllValuesFrom` et `SomeValuesFrom` (partie grisée de la figure) au modèle d'ontologies noyau de OntoQL (partie non grisée).

Explication. Les constructeurs de restriction OWL permettent de définir une classe dont les instances

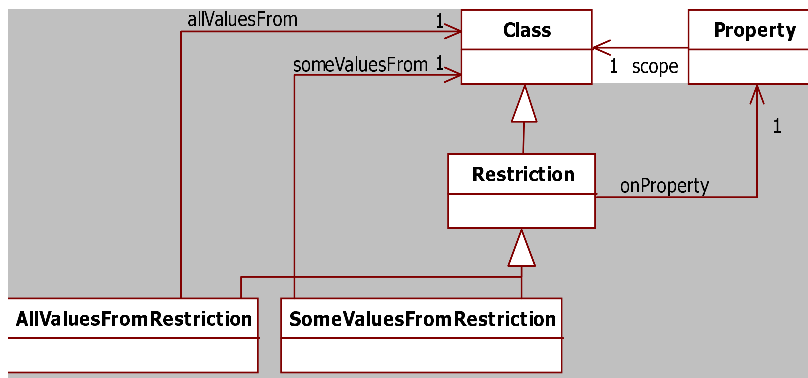


FIG. 4.2 – Exemple d'extensions du modèle d'ontologies noyau du langage OntoQL

sont caractérisées par les valeurs qu'elles prennent pour une propriété donnée. Pour représenter la notion de restriction dans le modèle d'ontologies noyau, nous avons introduit l'entité `Restriction`. Cette entité hérite de l'entité `Class`. De cette manière, chaque instance de cette entité sera considérée comme une classe. Elle possède un attribut `onProperty` qui indique la propriété sur laquelle porte une restriction. Les constructeurs `AllValuesFrom` et `SomeValuesFrom` sont deux constructeurs de restriction. Ils ont donc été introduits dans le modèle noyau comme deux nouvelles entités héritant de `Restriction` (et donc également de l'attribut `onProperty`). Les instances d'une restriction `AllValuesFrom` (respectivement `SomeValuesFrom`) ne doivent prendre comme valeurs de la propriété indiquée par l'attribut `onProperty` que des instances (respectivement au moins une instance) d'une classe donnée. Cette classe est indiquée par l'attribut `allValuesFrom` (respectivement `someValuesFrom`).

Les constructeurs d'un modèle d'ontologies peuvent ainsi être ajoutés au modèle noyau. Cette tâche peut même être automatisée pour les modèles d'ontologies pour lesquels un méta-modèle a été défini. C'est le cas pour le modèle PLIB dont le méta-modèle est défini en EXPRESS dans les normes ISO 13584 et également pour les modèles RDF-Schema et OWL dont un méta-modèle en UML a été défini par l'OMG dans [ODM, 2006]. Ainsi, la capacité d'extension du modèle noyau permet de représenter dans une BDBO l'ensemble des descriptions données dans une ontologie et ceci, quel que soit le modèle d'ontologies avec lequel elles sont définies.

2.1.2.2 Dimension sémantique de l'extension du modèle noyau

Considérons maintenant le problème plus délicat d'associer une sémantique aux extensions introduites. Lorsque cette extension est obtenue par spécialisation, les nouvelles entités héritent du comportement de leurs super-entités. Ce comportement est défini dans la sémantique opérationnelle du modèle noyau. Ainsi, toute spécialisation de l'entité `Class` définit une nouvelle catégorie de classes, mais celles-ci conservent, par héritage, le comportement usuel d'une classe. Toute instance de l'entité `Class` (ou d'une quelconque spécialisation) définit un conteneur susceptible d'être associé à des instances. Le nom de ce conteneur est généré par une fonction dite de concrétisation qui permet d'y accéder à partir de sa représentation dans le modèle d'ontologies. De même, toute spécialisation de l'entité `Property` définit des relations associant des instances de classes à des domaines de valeurs. Le nom de ces relations est égale-

ment dérivé des instances de propriétés qui les définissent. Enfin, les spécialisations de l'entité `Datatype` définissent des domaines de valeurs permettant de typer des propriétés. L'héritage permet donc d'associer la sémantique définie dans le modèle d'ontologies noyau aux nouveaux constructeurs. Par contre, il ne permet pas de leur en associer une nouvelle. Ainsi, une restriction OWL `AllValuesFrom` pourra être associée à des instances (héritage du comportement d'une classe). Par contre, ceci n'indique pas que ces instances peuvent être dérivées à partir d'une requête OntoQL. Nous verrons cependant que les différents langages proposés par OntoQL offrent une solution pour définir cette sémantique (cf. section 4.1).

Ce n'est, par contre, pas le cas pour les attributs qui peuvent être ajoutés au modèle noyau. Par exemple, la sémantique des qualificatifs de propriétés introduits par OWL (inverse, symétrique et transitive) ne peut pas être définie. OntoQL ne permet pas non plus d'associer une nouvelle sémantique aux types de données ajoutés (entité héritant de `Datatype`). Ainsi les types de données numériques PLIB associés à des unités de mesure (par exemple, `int_measure_type`) seront traités comme des types numériques classiques alors qu'ils nécessitent d'en redéfinir les opérateurs pour prendre en compte l'unité de mesure. Proposer une solution à ces limitations fait partie de nos perspectives de travail (cf. conclusion finale).

2.2 Aspects syntaxiques

Le langage OntoQL doit permettre d'exploiter les ontologies d'une BDBO en se basant sur le modèle de données que nous venons de présenter. La syntaxe de ce langage doit donc prendre en compte le fait que ce modèle de données n'est pas figé tout en permettant de distinguer les traitements portant sur les données à base ontologique de ceux portant sur les ontologies.

2.2.1 Distinction des traitements portant sur les ontologies de ceux portant sur les données

Puisque le modèle d'ontologies noyau sur lequel le langage OntoQL est basé n'est pas statique, mais qu'il peut être étendu, les éléments de ce niveau de modélisation ne peuvent pas être codés comme des mots clés de sa grammaire. Donc, nous avons dû définir une convention permettant de reconnaître dans la grammaire un élément du modèle d'ontologies à partir de son seul nom. La convention que nous avons choisie est de préfixer chaque élément de ce modèle par le caractère `#`. Ce préfixe permet de savoir que la définition de cet élément doit être insérée ou recherchée dans le modèle d'ontologies éventuellement étendu du langage OntoQL. Ceci permet donc de distinguer la manipulation des données de la manipulation des ontologies. Notons que, bien que la même convention soit utilisée pour les éléments du modèle noyau, les noms de ces éléments (par exemple, `#Class`) sont en fait des mots-clés qui ne peuvent pas être changés car ils font l'objet d'une interprétation sémantique pré-définie.

Par contre, le préfixe `#` peut être jugé peu commode. Pour ne pas imposer l'utilisation de ce préfixe, le langage OntoQL permet d'utiliser un espace de noms prédéfini `http://lisi.ensma.fr/ontoql`. Un élément de cet espace de noms est considéré comme un élément du modèle d'ontologies. Et donc, lorsque cette espace de noms est défini par défaut ou qu'il est précisé dans une requête, il n'est pas nécessaire de préfixer les éléments du modèle d'ontologies par `#`.

2.2.2 Utilisation des identifiants des éléments ontologiques

Lorsque l'on observe la figure 4.1, on constate que de nombreux attributs du modèle d'ontologies noyau sont de type référence. Les valeurs de ces attributs sont des identifiants internes d'ontologies, de classes, de propriétés ou de types de données. Ces identifiants étant des entiers, attribuer une valeur à ces attributs nécessite d'exprimer des requêtes pour les rechercher. Or, les ontologies, les classes, les propriétés et les types de données peuvent être identifiés plus simplement :

- les ontologies par leur espace de noms ;
- les classes et les propriétés par leur identifiant externe et/ou leur nom préféré ;
- les types de données par leur nom.

Il serait donc particulièrement utile que ces identifiants puissent être utilisés pour attribuer une valeur aux attributs de type référence.

Les identifiants internes sont des entiers (syntaxe sans `' '`) tandis que les identifiants externes évoqués précédemment sont des chaînes de caractères (syntaxe avec `' '`). Ces deux types d'identifiants peuvent donc être distingués dans la syntaxe OntoQL. En conséquence, ils peuvent tous deux être utilisés pour définir la valeur d'attributs de type référence.

Les mécanismes syntaxiques définis dans cette section sont utilisés par les langages qui permettent de manipuler les ontologies d'une BDBO. Notre but étant que la syntaxe du langage OntoQL soit uniforme pour les différents niveaux d'accès à une BDBO, nous nous sommes à nouveau basés sur les instructions de manipulation des types utilisateurs de SQL pour définir ces langages. Mais, cette fois-ci, le modèle de données considéré ne présente pas de différence majeure avec le modèle relationnel-objet. En effet, la seule différence entre une entité et un type utilisateur est que les entités peuvent être organisées dans une hiérarchie supportant l'héritage multiple (nécessaire, en particulier, pour le modèle PLIB). Les instructions proposées par OntoQL pour manipuler les ontologies restent donc similaires à celles de manipulation des types utilisateurs.

2.3 Langage de Définition des Ontologies (LDO)

Le LDO permet de créer, modifier et supprimer les entités et attributs dans le modèle d'ontologies considéré par OntoQL. Ce langage permet donc de modifier le modèle d'ontologies utilisé.

2.3.1 Création des entités et des attributs

La syntaxe permettant de créer une nouvelle entité est la suivante :

⟨*entity definition*⟩ ::= `CREATE ENTITY` ⟨*entity id*⟩ [⟨*under clause*⟩] ⟨*attribute clause*⟩

⟨*under clause*⟩ ::= `UNDER` ⟨*entity id list*⟩

⟨*attribute clause*⟩ ::= ⟨*attribute definition list*⟩

⟨*attribute definition*⟩ ::= ⟨*attribute id*⟩ ⟨*datatype*⟩ [⟨*derived clause*⟩]

⟨*derived clause*⟩ ::= `DERIVED BY` ⟨*function name*⟩

Cette instruction permet de créer une nouvelle entité dans le modèle d'ontologies noyau de OntoQL de manière similaire à la création d'un type utilisateur. Ainsi, cette nouvelle entité peut être créée comme sous-entité d'une ou de plusieurs autres entités (UNDER). Elle est définie avec la liste des attributs qui permettent de caractériser ses instances. Par défaut, les valeurs de ces attributs sont définies par un utilisateur. Cependant, comme nous le verrons lorsque nous définirons le langage d'interrogation des ontologies (cf. section 2.5), il est également nécessaire que ces valeurs puissent être calculées à partir des valeurs des autres attributs afin de permettre d'exprimer des requêtes récursives typiques, telles que rechercher l'ensemble des sous-classes d'une classe donnée. Cette instruction permet donc de définir des *attributs dérivés* grâce à la clause ⟨*derived clause*⟩. La fonction de dérivation d'un tel attribut est indiquée par le nom d'une fonction utilisateur (⟨*function name*⟩). Elle doit être définie avec le langage de programmation (SQL/PSM) associé au SGBD sur lequel la BDBO est implantée.

Exemple. Ajouter le constructeur de restriction OWL `AllValuesFrom` au modèle d'ontologies noyau.

```
CREATE ENTITY #Restriction UNDER #Class (
  #onProperty REF(#Property)
)

CREATE ENTITY #AllValuesFrom UNDER #Restriction (
  #allValuesFrom REF(#Class)
)
```

Explication. Ces instructions permettent d'ajouter le constructeur de restriction OWL `AllValuesFrom` en suivant l'exemple d'extensions du modèle noyau présenté sur la figure 4.2. La première instruction permet de créer l'entité `#Restriction` regroupant les différents types de restrictions OWL. Une restriction étant une classe, ce constructeur est ajouté au modèle d'ontologies comme une nouvelle entité héritant de `#Class`. Cette entité est associée à l'attribut `#onProperty` indiquant la propriété sur laquelle porte la restriction. Son type est donc un type référence vers les propriétés (`REF(#Property)`).

La seconde instruction permet de créer l'entité `#AllValuesFrom` comme sous-classe de `#Restriction` pour ajouter le constructeur du même nom. Cette entité est associée à l'attribut `#allValuesFrom`. Cet attribut indique la classe dans laquelle les instances de la restriction prennent leurs valeurs pour la propriété définie par l'attribut `#onProperty`. Son type est donc un type référence vers une classe (`REF(#Class)`).

2.3.2 Modification des entités et des attributs

La modification des entités et des attributs du modèle d'ontologies de OntoQL peut être réalisée selon la syntaxe suivante :

⟨*alter entity statement*⟩ ::= `ALTER ENTITY` ⟨*entity id*⟩ ⟨*alter entity action*⟩
⟨*alter entity action*⟩ ::= ⟨*add attribute definition*⟩ | ⟨*drop attribute definition*⟩
⟨*add attribute definition*⟩ ::= `ADD` [`ATTRIBUTE`] ⟨*attribute definition*⟩
⟨*drop attribute definition*⟩ ::= `DROP` [`ATTRIBUTE`] ⟨*attribute id*⟩

Une entité peut ainsi être modifiée en ajoutant (ADD) ou en supprimant DROP un attribut. Le modèle d'ontologies noyau étant la base sur laquelle la sémantique du langage OntoQL est fondée, il n'est pas possible de supprimer un attribut de ce modèle.

Exemple. Modifier le constructeur de restriction `#AllValuesFrom` pour permettre de retrouver les restrictions portant sur la même propriété qu'une restriction donnée.

```
ALTER ENTITY #AllValuesFrom
      ADD ATTRIBUTE #restrictionsOnTheSameProperty REF(#Restriction) ARRAY
                    DERIVED BY restrictions_on_the_same_property(#oid)
```

Explication. Cette instruction ajoute l'attribut `#restrictionsOnTheSameProperty` à l'entité `#AllValuesFrom`. Cet attribut indique les restrictions portant sur la même propriété. Le type de cet attribut est donc un type référence vers d'autres restrictions OWL quelles que soient leur type (`REF(#Restriction) ARRAY`). Les valeurs de cet attribut peuvent être calculées en parcourant les autres restrictions afin de déterminer celles qui portent sur la même propriété. Il est donc défini comme un attribut dérivé (`DERIVED BY`) dont la fonction de calcul est `restrictions_on_the_same_property`. Cette fonction prend en paramètre l'identifiant interne d'une restriction (`#oid`) et retourne une collection d'identifiants internes de restrictions (`REF(#Restriction) ARRAY`). Elle devra être codée avec le langage de programmation associé au SGBD sur lequel la BDBO est implantée.

2.3.3 Suppression des entités et des attributs

La suppression d'entités et d'attributs dans le modèle d'ontologies est réalisée par la syntaxe suivante :

⟨*drop entity statement*⟩ ::= `DROP ENTITY` ⟨*entity id*⟩

Cette instruction permet de supprimer l'entité ⟨*entity id*⟩ ainsi que l'ensemble de ses attributs. Une entité ne peut être supprimée que si elle n'appartient pas au modèle noyau et si elle n'est pas référencée par une autre entité. C'est le cas si elle a des sous-entités ou si elle est utilisée pour définir le codomaine d'un attribut. Elle ne peut pas non plus être supprimée si cette entité est une sous-entité de `#Class` et qu'une de ses instances est associée à une extension.

Exemple. Supprimer le constructeur de restriction `#AllValuesFrom`.

```
DROP ENTITY #AllValuesFrom
```

Explication. Cette instruction supprime l'entité `#AllValuesFrom`. Elle ne sera acceptée que dans le cas où les restrictions créées ne sont pas associées à des extensions permettant de stocker leurs instances. Elle provoque, en même temps, la suppression de l'attribut `#allValuesFrom` défini sur cette entité.

2.4 Langage de Manipulation des Ontologies (LMO)

Le LMO doit permettre de créer, modifier et supprimer les éléments d'une ontologie tels que ses classes et ses propriétés. Ayant déjà défini une syntaxe permettant de créer les classes et les propriétés d'une ontologie (LDD pour les données à base ontologique), ceci doit être pris en compte lors de

l'analyse des besoins restants à couvrir. Notons que la syntaxe proposée par le LDD est pratique car elle permet de créer dans une seule et même instruction une classe et les propriétés dont elle est le domaine de définition. Utiliser un LMO pour créer ces classes et ces propriétés nécessiterait plusieurs instructions `INSERT`. Notons que cela reviendrait, dans un SGBD usuel, à créer une table en réalisant des insertions dans les tables de la métabase. Par contre, les instructions du LDD ne sont pas aussi puissantes que celles d'un LMO. En effet, ce dernier devrait permettre de créer, modifier et supprimer un ensemble de classes en se basant sur le résultat d'une requête. Ceci peut être utile, par exemple, pour importer un ensemble de classes d'une ontologie dans une autre ontologie. En conséquence, il apparaît souhaitable que ces deux langages existent et permettent tous les deux de manipuler les éléments d'une ontologie. Des équivalences syntaxiques sont définies de façon systématique entre les insertions (`INSERT`) du niveau manipulation d'ontologies et la création de classe (`CREATE`) de niveau définition de données. Les deux sont licites, la seconde écriture s'avérant en général plus synthétique.

2.4.1 Ajout d'éléments dans une ontologie

Le LMO permet de créer des éléments dans une ontologie selon la syntaxe suivante :

⟨*insert statement*⟩	::=	`INSERT INTO` ⟨*entity id*⟩ ⟨*insert description and source*⟩
⟨*insert description and source*⟩	::=	⟨*from subquery*⟩ | ⟨*from constructor*⟩
⟨*from subquery*⟩	::=	[`(` ⟨*attribute id list*⟩ `)`] ⟨*query expression*⟩
⟨*from constructor*⟩	::=	[`(` ⟨*attribute id list*⟩ `)`] ⟨*values clause*⟩
⟨*values clause*⟩	::=	`VALUES (` ⟨*values expression list*⟩ `)`

Cette opération permet d'ajouter des instances à l'entité ⟨*entity id*⟩. Ces instances peuvent être définies en précisant l'ensemble de leurs valeurs d'attributs (⟨*from constructor*⟩). Elles peuvent également être le résultat d'une requête OntoQL (⟨*from subquery*⟩). Les entités et attributs utilisés sont soit ceux du modèle d'ontologies noyau (récapitulés dans le tableau 4.1 pour les principales entités), soit ceux introduits avec le langage de définition de données qui ne sont pas des attributs dérivés.

Entités	**Attributs applicables sur cette entité**
`#Ontology`	`#oid`, `#namespace`
`#Concept`	`#oid`, `#code`, `#name`, `#definition`, `#definedBy`
`#Class`	`#oid`, `#code`, `#name`, `#definition`, `#directSuperclasses`, `#definedBy`
`#Property`	`#oid`, `#code`, `#name`, `#definition`, `#scope`, `#range`, `#definedBy`
`#Datatype`	`#oid`
`#RefType`	`#oid`, `#onClass`
`#PrimitiveType`	`#oid`
`#CollectionType`	`#oid`, `#ofDatatype`, `#maxCardinality`

Tab. 4.1 – Attributs applicables sur les principales entités du modèle noyau

Exemple. Ajouter la propriété `title` à la classe `Post`.

```
INSERT INTO #Property (#name, #scope, #range, #definedBy)
  VALUES('title', 'Post', 'String', 'http://rdfs.org/sioc/ns')
```

Explication. L'insertion de cette propriété est réalisée en indiquant la valeur de chacun de ces attributs obligatoires, c'est-à-dire son nom (`#name`), son domaine (`#scope`), son codomaine (`#range`) et l'ontologie dans laquelle elle est créée (`#definedBy`). Pour donner une valeur aux attributs de type référence, cette instruction utilise l'identifiant externe des éléments référencés plutôt que leur identifiant interne, généré par le SGBD (cf. section 2.2.2). Ainsi, cette instruction utilise l'identifiant `Post` et l'espace de noms `http://rdfs.org/sioc/ns` pour référencer la classe et l'ontologie correspondantes. Le type de données est indiqué par son nom. L'interpréteur de cette instruction est chargé de rechercher si une instance de ce type existe et, si ça n'est pas le cas de la créer.

Cet exemple montre que les instructions du LMO pour créer des éléments d'une ontologie sont moins synthétiques que leur équivalent dans le LDD. En effet, si on devait utiliser ce langage pour créer la classe `Post` de l'ontologie SIOC, il serait nécessaire d'utiliser cinq instructions du LMO : une pour créer cette classe et quatre autres pour créer ses propriétés. Or, dans l'exemple du chapitre 3, section 3.3.1.1, nous avons vu que seule une instruction du LDD est nécessaire pour réaliser le même traitement.

2.4.2 Modification des éléments d'une ontologie

Les éléments créés dans une ontologie peuvent être modifiés en utilisant la syntaxe suivante :

⟨*update statement*⟩ ::= `UPDATE` ⟨*entity id polymorph*⟩ `SET` ⟨*set clause list*⟩
[`WHERE` ⟨*search condition*⟩]
⟨*entity id polymorph*⟩ ::= ⟨*entity id*⟩ | `ONLY` (⟨*entity id*⟩)
⟨*set clause*⟩ ::= ⟨*attribute id*⟩ = ⟨*value expression*⟩

Cette syntaxe permet de mettre à jour les instances directes (`ONLY`) ou également indirectes (sans le mot clé `ONLY`) d'une entité en leur attribuant de nouvelles valeurs d'attributs.

Exemple. Préfixer le nom des restrictions OWL par `R_` et incrémenter leur numéro de version.

```
UPDATE #Restriction SET #name = 'R_' || #name, #version = #version+1
```

Explication. Le mot clé `ONLY` n'étant pas utilisé, toutes les restrictions OWL (`AllValuesFrom`, `SomeValuesFrom`, etc.) seront mises à jour. Le préfixe `R_` est ajouté à leur nom en utilisant l'opérateur de concaténation (`||`). Cette instruction permet en même temps d'incrémenter le numéro de version de ces restrictions (attribut `#version`).

L'exemple précédent montre qu'il peut être plus intéressant d'utiliser le LMO plutôt que le LDD lorsque plusieurs éléments d'une ontologie doivent être modifiés en même temps.

2.4.3 Suppression des éléments d'une ontologie

Les éléments des ontologies peuvent être supprimés en utilisant la syntaxe suivante :

⟨*delete statement*⟩ ::= `DELETE FROM` ⟨*entity id polymorph*⟩ `WHERE` ⟨*search condition*⟩

Cette instruction permet de supprimer les instances directes ou également indirectes d'une entité qui respectent un prédicat donné (⟨*search condition*⟩).

Exemple. Supprimer les classes obsolètes.

```
DELETE FROM #Class WHERE #deprecated = true
```

Explication. Seules les classes obsolètes sont supprimées dans cette instruction en utilisant le fait qu'elles sont caractérisées par la valeur `true` pour l'attribut `#deprecated`.

2.5 Langage d'Interrogation des Ontologies (LIO)

Le LIO permet d'exprimer des requêtes sur les ontologies d'une BDBO. Ces requêtes sont construites de manière identique aux requêtes portant sur les données sauf que, dans ce cas, les entités et les attributs (noms précédés par # sauf utilisation de l'espace de noms particulier défini en section 2.2.1) sont utilisés à la place des classes et des propriétés.

2.5.1 Exemples de requêtes

Les entités et attributs utilisés dans les requêtes sont soit ceux du modèle d'ontologies noyau du langage OntoQL (cf. tableau 4.1), soit ceux ajoutés avec le LDO. Nous illustrons le langage obtenu par des exemples.

Exemple. Rechercher les espaces de noms des ontologies stockées dans la BDBO.

```
SELECT #namespace FROM #Ontology
```

Explication. Les ontologies de la BDBO sont des instances de l'entité `#Ontology`. Les espaces de noms qu'elles définissent sont donnés par l'attribut `#namespace`. Pour retrouver les espaces de noms des ontologies de la BDBO, il suffit donc de réaliser la projection de l'entité `#Ontology` sur l'attribut `#namespace`.

Les opérateurs orientés-objets, introduits dans le langage permettant d'interroger les données au niveau ontologique, peuvent également être utilisés pour interroger les ontologies. L'exemple suivant montre l'utilisation d'une expression de chemin.

Exemple. Rechercher l'identifiant externe des propriétés de type référence vers les instances de la classe `Post`.

```
SELECT p.#code
  FROM #Property AS p, #RefType AS t
 WHERE p.#range = t.#oid
   AND t.#onClass.#code = 'Post'
```

Explication. Pour retrouver les propriétés de type référence, une jointure est réalisée entre les entités `#Property`, contenant les propriétés des ontologies, et `#RefType`, contenant les types référence. La classe utilisée par un type référence est indiquée par l'attribut `#onClass`. Une expression de chemin est construite à partir de cet attribut afin de ne garder que les types références portant sur la classe dont l'identifiant externe (`#code`) est `Post`.

Les valeurs des attributs peuvent être indiquées dans plusieurs langues naturelles. Une syntaxe identique à celle utilisée pour les propriétés permet d'obtenir la valeur d'un attribut dans une langue naturelle donnée.

Exemple. Rechercher le nom en français et en anglais des classes et propriétés de l'ontologie SIOC.

```
SELECT #name[FR], #name[EN]
  FROM #Concept
 WHERE #definedBy.#namespace = 'http://rdfs.org/sioc/ns'
```

Explication. L'entité `#Concept` regroupe les classes et les propriétés des ontologies. Pour ne retourner que celles appartenant à l'ontologie SIOC, une expression de chemin est utilisée à partir des attributs `#definedBy`, qui indique l'ontologie dans laquelle une classe ou une propriété est définie, et `#namespace` qui indique l'espace de noms d'une ontologie. Le nom de chaque classe et de chaque propriété est retourné en français (`[FR]`) et en anglais (`[EN]`).

Ces exemples montrent que le langage OntoQL permet de découvrir les ontologies conservées dans une BDBO. Cependant, les attributs disponibles dans le modèle noyau ne sont pas toujours suffisants pour certaines requêtes, comme celles décrites dans la section suivante.

2.5.2 Attributs dérivés prédéfinis

Une requête typique portant sur les ontologies d'une BDBO consiste à rechercher l'ensemble des sous-classes (directes et indirectes) pour une classe donnée. Le langage OntoQL ne supportant pas la récursivité présente dans SQL mais qui n'est pas implantée dans tous les SGBD, cette requête ne peut être exprimée. De même, une autre requête usuelle consiste à rechercher les propriétés applicables (héritées et définies sur une classe donnée). Les requêtes de ce type ne peuvent être exprimées que très difficilement avec OntoQL. Pour permettre d'exprimer ce type de requêtes, nous avons ajouté des attributs prédéfinis à ceux issus du modèle d'ontologies noyau de OntoQL. Ces attributs sont *dérivés* à l'aide de fonction utilisateur. L'implantation du langage OntoQL sur une BDBO doit donc comprendre la définition des fonctions utilisateur permettant de calculer ces attributs. Par exemple, pour implanter le langage OntoQL sur la BDBO OntoDB qui utilise le SGBD PostgreSQL, nous avons défini ces fonctions avec le langage PL/PgSQL.

La liste des attributs dérivés prédéfinis, que nous avons décidé d'ajouter, est présentée dans le tableau 4.2. Il indique, pour chaque attribut, l'entité sur laquelle il est défini ainsi qu'une description informelle de cet attribut. Nous avons ajouté des attributs facilitant le parcours de la hiérarchie de classes. Ainsi, l'attribut `#superclasses` permet d'obtenir les identifiants de toutes les super-classes d'une classe donnée et les attributs `#directSubclasses` et `#subclasses`, les identifiants de ses sous-classes directes et indirectes rangés dans un tableau. Nous avons également ajouté des attributs permettant de récupérer les identifiants des propriétés définies (`#scopeProperties`), applicables (`#properties`) et utilisées (`#usedProperties`) associées à une classe, également rangés dans un tableau.

Ces attributs sont utiles pour découvrir les ontologies d'une BDBO comme le montre les exemples suivants.

Exemple. Rechercher les noms des super-classes de la classe `Post`.

Attributs	Domaine	Définition informelle
#superclasses	#Class	Retourne les identifiants internes des super-classes (directes et indirectes) d'une classe donnée.
#directSubclasses	#Class	Retourne les identifiants internes des sous-classes directes d'une classe donnée (inverse de #directSuperclasses).
#subclasses	#Class	Retourne les identifiants internes des sous-classes (directes et indirectes) d'une classe donnée (inverse de #superclasses).
#scopeProperties	#Class	Retourne les identifiants internes des propriétés définies sur une classe donnée (inverse de #scope).
#properties	#Class	Retourne les identifiants internes des propriétés applicables (définies et héritées) sur une classe donnée.
#usedProperties	#Class	Retourne les identifiants internes des propriétés utilisées pour décrire les instances d'une classe donnée (propriétés présentes dans l'extension de la classe).

Tab. 4.2 – Les attributs dérivés prédéfinis

```
SELECT csup.#name
  FROM #Class AS c,
       UNNEST(c.#superclasses) AS csup
 WHERE c.#name = 'Post'
```

Explication. L'opérateur UNNEST permet de dégrouper la relation contenant les classes par rapport à l'attribut #superclasses. Ceci permet d'obtenir un itérateur csup sur les super-classes de la classe sur laquelle l'itérateur c se trouve. Pour obtenir le nom de la super-classe courante il suffit donc de projeter la classe csup sur l'attribut #name.

Exemple. Rechercher le nom des propriétés utilisées sur la classe Post, ainsi que le nom de la classe sur laquelle ces propriétés ont été définies.

```
SELECT prop.#name, prop.#scope.#name
  FROM #Class AS c,
       UNNEST(c.#usedProperties) AS prop
WHERE c.#name = 'Post'
```

Explication. L'opérateur UNNEST est utilisé comme dans l'exemple précédent pour obtenir un itérateur (prop) sur les propriétés utilisées pour construire l'extension de la classe c. Cet itérateur est utilisé pour obtenir le nom de ces propriétés ainsi que le nom de leur domaine en utilisant une expression de chemin.

Exemple. Rechercher le nom des propriétés ainsi que le nom de leur type de données.

```
SELECT p.#name,
       CASE WHEN p.#range IS OF (REF(#StringType)) THEN 'String'
            WHEN p.#range IS OF (REF(#IntType)) THEN 'Int'
            WHEN p.#range IS OF (REF(#RefType)) THEN 'RefType'
            ELSE 'Unknown' END
```

```
  FROM #Property AS p
```

Explication. L'attribut `#range` retourne une référence vers le type de données d'une propriété. Cette requête teste le type de cette référence (opérateur `IS OF`) et retourne une chaîne de caractères représentant le type de données. Par exemple, si le type de données de la propriété `p` est un type référence (`IS OF (REF(#RefType))`), la chaîne de caractères `'RefType'` sera retournée.

2.5.3 Paramétrage du langage

Nous avons vu que les instructions OntoQL sont paramétrées par les espaces de noms et la langue naturelle utilisés. Ces paramètres sont également utilisés pour exploiter les ontologies.

2.5.3.1 Paramétrage par la langue naturelle utilisée

Commençons par la langue naturelle. Pour permettre d'exprimer une requête multilingue non seulement sur les données mais aussi sur les ontologies, les noms des entités et des attributs peuvent être définis dans plusieurs langues naturelles en utilisant la clause `DESCRIPTOR`.

Exemple. Rechercher le nom des propriétés utilisées sur la classe `Post` (`Message` en français), ainsi que le nom de la classe sur laquelle ces propriétés ont été définies en exprimant une requête en français.

```
SELECT prop.#nom, prop.#domaine.#nom
  FROM #Classe AS c,
       UNNEST(c.#propriétésUtilisées) AS prop
WHERE  c.#nom = 'Message'
USING LANGUAGE FR
```

Explication. Cette requête a été exprimée en anglais dans la section précédente. Dans, cette exemple, la clause `USING LANGUAGE` permet d'indiquer que la requête est écrite en français. Les entités et attributs de cette requête sont ainsi identifiés par leur nom en français. Par exemple, l'attribut qui indique le domaine d'une propriété, nommé `#scope` en anglais, est indiqué par le nom `#domaine`. Ce langage s'applique également aux valeurs des attributs définis par une chaîne multilingue (`'Message'`).

2.5.3.2 Paramétrage par les espaces de noms utilisés

Considérons maintenant le cas des espaces de noms. Pour l'exploitation des données à base ontologique, ces espaces de noms sont utilisés pour identifier les éléments d'une requête. Une possibilité serait que ces espaces de noms influencent également le résultat d'une requête. Cependant, donner deux sens à ce mécanisme nous a semblé une source de confusion. Donc, ce mécanisme peut être utilisé pour exploiter les ontologies seulement pour éviter de devoir utiliser le préfixe #.

Exemple. Rechercher les noms des classes de l'ontologie SIOC sans utiliser le préfixe #.

```
SELECT c.name
  FROM class AS c
 WHERE c.definedBy.namespace = 'http://rdfs.org/sioc/ns'
USING NAMESPACE 'http://lisi.ensma.fr/ontoql'
```

Explication. L'espace de noms utilisé étant `http://lisi.ensma.fr/ontoql`, les éléments de cette requête sont considérés comme des entités et des attributs. Cette requête nécessite d'utiliser l'attribut `#definedBy` pour ne rechercher que les classes de l'ontologie `SIOC`.

La présentation du LIO termine la description des traitements disponibles pour exploiter les ontologies d'une BDBO. Le langage OntoQL permet donc d'interroger d'un côté les données à base ontologique et de l'autre, les ontologies d'une BDBO. Ceci laisse entrevoir la possibilité de combiner ces deux niveaux d'interrogation.

3 Interrogation conjointe des ontologies et des données

L'exigence 10 requiert que le langage permette d'interroger à la fois l'ontologie et les données. Cette exigence a été définie parce que le langage doit permettre d'une part de rechercher des classes avec leurs instances (ontologies vers données) et d'autre part d'effectuer des requêtes sur les instances en récupérant en même temps leur description ontologique (données vers ontologies).

3.1 Des ontologies vers les données à base ontologique

Le LIO défini pour interroger les ontologies permet de rechercher les classes et les propriétés définies dans les ontologies.

Exemple. Retourner les classes contenues dans la BDBO avec leurs propriétés applicables[26].

```
SELECT c.#name, p.#name
  FROM #Class AS c, UNNEST(c.#properties) AS p
```

Explication. Pour chaque classe `c`, un itérateur `p` est introduit sur les propriétés applicables sur cette classe. Les noms des classes `c` et des propriétés `p` sont retournés. Le résultat de cette requête pour les classes `Item` et `Forum` est ainsi :

c.#name	p.#name
'Item'	'name'
'Item'	'has reply'
'Item'	'has container'
'Forum'	'name'
'Forum'	'has host'
'Forum'	'has moderator'

Nous pouvons alors utiliser le LID défini pour interroger les données à base ontologique afin de rechercher les instances des classes retournées et leurs valeurs de propriétés.

Exemple. Retourner les instances des classes `Item` et `Forum` avec leurs valeurs pour les différentes propriétés retournées dans le résultat précédent.

[26] Les exemples de cette section suppose que la langue naturelle par défaut est l'anglais.

```
SELECT name, "has reply", "has container" FROM Item
```

```
SELECT name, "has host", "has moderator" FROM Forum
```

Explication. Le résultat de la requête précédente indique que les propriétés applicables sur la classe `Item` sont `name`, `has reply` et `has container`. La première requête retourne les valeurs de des propriétés des instances de la classe `Item` en utilisant une projection impliquant ces propriétés. La seconde requête fait de même pour retourner les valeurs de propriétés des instances de la classe `Forum`.

Pouvoir interroger à la fois les ontologies et les données dans le sens ontologie vers données consiste à combiner les deux types de requêtes précédentes. Une même requête doit permettre de rechercher les classes et leurs propriétés ainsi que leurs instances et leurs valeurs de propriétés. Dans les deux sections suivantes, nous présentons les mécanismes que nous avons introduits dans le langage OntoQL à cette fin.

3.1.1 Itérateurs et liaisons dynamiques

Dans les requêtes portant sur les données à base ontologique, suivant l'approche des langages traditionnels des bases de données, un itérateur `i` sur les instances d'une classe `C` est introduit par la syntaxe `C AS i`. Dans les deux requêtes présentées dans la section précédente, `C` représente respectivement la classe `Item` et la classe `Forum`. Ainsi, dans ces requêtes, la classe `C` est connue avant l'exécution de la requête. L'interrogation conjointe des ontologies et des données nécessite donc que cette liaison soit dynamique. Nous avons donc étendu ce mécanisme pour permettre d'introduire un itérateur sur les instances d'une classe déterminée dynamiquement pendant l'exécution de la requête. L'introduction de ce type d'itérateur dans la clause `FROM` d'une requête OntoQL est réalisée à l'aide du constructeur `AS` (qui peut être implicite), selon la syntaxe suivante :

⟨*dynamic iterator*⟩ ::= ⟨*identifier polymorph*⟩ [`AS`] ⟨*alias name*⟩
⟨*identifier polymorph*⟩ ::= ⟨*identifier*⟩ | `ONLY (` ⟨*identifier*⟩`)`

Cette syntaxe est similaire à l'introduction d'un itérateur classique à part que la classe dont les instances (directes ou également indirectes) sont parcourues par l'itérateur est une variable (⟨*identifier*⟩). Ce mécanisme permet d'obtenir les classes et leurs propriétés ainsi que les instances de cette classe. Notons que ce mécanisme a été introduit dans les langages pour les bases de données fédérées tels que SchemaSQL [Lakshmanan et al., 2001] ou MSQL [Litwin et al., 1989] afin de permettre l'interrogation des données d'une base de données, indépendamment du schéma logique de ces données, en utilisant la métabase.

Exemple. Retourner les classes contenues dans la BDBO avec leurs propriétés applicables et les identifiants de leurs instances.

```
SELECT c.#name, p.#name, i.oid
  FROM #Class AS c, UNNEST(c.#properties) AS p, c AS i
```

Voici un extrait du résultat de cette requête où nous n'avons représenté qu'une instance par classe :

c.#name	p.#name	i.oid
'Item'	'name'	1
'Item'	'has reply'	1
'Item'	'has container'	1
'Forum'	'name'	11
'Forum'	'has host'	11
'Forum'	'has moderator'	11

Explication. L'itérateur `c` parcourt les classes. Un itérateur dynamique `i` est introduit sur les instances de ces classes par la syntaxe `c AS i`. La valeur de la propriété `oid` est retournée pour chacune de ces instances.

Dans l'exemple précédent, la propriété `oid` peut être projetée car elle est définie pour chaque classe. Nous présentons maintenant le mécanisme que nous avons introduit pour permettre de projeter des instances sur les autres propriétés qui les caractérisent.

3.1.2 Projection utilisant des propriétés déterminées à l'exécution d'une requête

Grâce au mécanisme précédent (opérateur `AS`), nous pouvons introduire un itérateur sur les instances de classes déterminées à l'exécution de la requête. Les propriétés applicables sur ces classes, qui caractérisent leurs instances, peuvent également être recherchées. Par contre, aucun mécanisme introduit jusqu'à présent ne permet de les utiliser dans une projection. Nous avons donc permis qu'une propriété déterminée à l'exécution de la requête soit utilisée dans la clause `SELECT` en utilisant la syntaxe suivante :

⟨*project dynamic property*⟩ ::= ⟨*identifier*⟩ `.` ⟨*identifier*⟩

La syntaxe permettant de réaliser une projection utilisant une propriété déterminée à l'exécution de la requête est similaire à celle définie pour les propriétés statiques à part que l'utilisation d'un préfixe est obligatoire et que la propriété dynamique est une variable (⟨*identifier*⟩).

Exemple. Retourner les classes contenues dans la BDBO avec leurs propriétés applicables ainsi que les identifiants de ces instances et leurs valeurs de propriétés.

```
SELECT c.#name, p.#name, i.oid, i.p
  FROM #Class AS c, UNNEST(c.#properties) AS p, c AS i
```

c.#name	p.#name	i.oid	i.p
'Item'	'name'	1	'message concernant l'ontologie SIOC'
'Item'	'has reply'	1	2
'Item'	'has container'	1	11
'Forum'	'name'	11	'forum sur les ontologies'
'Forum'	'has host'	11	111
'Forum'	'has moderator'	11	1111

Explication. Pour chaque classe `c`, l'itérateur `p` parcourt ses propriétés applicables. Et, la valeur de cette propriété est retournée pour chacune des instances `i` de `c` par la syntaxe `i.p`. Le résultat de cette requête est une relation où l'attribut correspondant à la projection `i.p` est une union des types chaînes

de caractères (pour la propriété `name`) et de plusieurs types références (pour les propriétés `has reply`, `has container` etc.).

Le langage OntoQL permet ainsi de rechercher un ensemble de classes ainsi que leurs instances et leurs valeurs de propriétés. Afin de ne pas rendre complexe l'implantation de ce langage, nous avons seulement permis d'utiliser une propriété déterminée dynamiquement dans l'opérateur de projection et non dans les autres opérateurs du langage (sélection, agrégat, etc.).

Les mécanismes introduits dans cette section permettent de retourner les instances de classes recherchées par une requête. Nous présentons maintenant les mécanismes introduits pour permettre l'opération inverse, c'est-à-dire retourner des informations sur la notion ontologique qui décrit des instances recherchées par une requête.

3.2 Des données à base ontologique vers les ontologies

Le LID permettant d'interroger les données à base ontologique depuis le niveau ontologique permet d'effectuer des recherches sur les instances directes et indirectes d'une classe. Ces requêtes retournent ainsi des instances dont les classes de base peuvent être différentes. La classe de base d'une instance la décrit en indiquant, par exemple, les propriétés utilisées pour la caractériser (propriétés de son extension). Cependant, aucun opérateur d'OntoQL défini jusqu'à présent ne permet de récupérer la classe de base d'une instance donnée. En effet, ce langage, comme SQL, est équipé d'opérateurs permettant de tester le type d'une instance et de convertir une instance dans un type donnée mais pas d'un opérateur permettant de récupérer le type d'une instance.

Nous avons donc décidé d'introduire un nouvel opérateur, nommé `TYPEOF` : `Oid` → `#Class`, qui permet d'obtenir la classe de base (de type `#Class`) à partir de l'identifiant d'une instance (type `Oid`). Cet opérateur peut être implanté sur une BDBO car elle conserve le lien entre les données (instances) et les éléments des ontologies (concepts) (cf. chapitre 1, section 4). La syntaxe permettant d'utiliser cet opérateur est la suivante :

⟨*typeof treatment*⟩ ::= `TYPEOF` `(` ⟨*value expression*⟩ `)`

L'opérateur `TYPEOF` peut être appelé en utilisant une notation fonctionnelle, similaire à celle de l'opérateur `DEREF`. Cette notation permet de distinguer cet opérateur d'une projection impliquant une propriété ou un attribut.

Exemple. Rechercher les noms des instances de la classe `Item` en retournant également le nom de leur classe de base ainsi que la définition de ces classes.

```
SELECT i.name, TYPEOF(i.oid).#name, TYPEOF(i.oid).#definition,
  FROM Item AS i
```

Explication. L'expression `TYPEOF(i.oid)` retourne la classe de base des instances `i` de la classe `Item` à partir de leur identifiant. Afin d'obtenir le nom et la définition des classes de base ainsi retournées, les attributs `#name` et `#definition` sont appliqués. Voici un extrait du résultat de cette requête :

i.name	TYPEOF(i.oid).#name	TYPEOF(i.oid).#definition
'The SIOC Ontology'	'Post'	'An article or message'
'SIOC Ontology RDF File'	'Item'	'A content Item'

L'opérateur TYPEOF peut également être utilisé pour réaliser des jointures entre des éléments des ontologies et des instances des classes.

Exemple. Même requête que dans l'exemple précédent, mais, en retournant les propriétés utilisées pour décrire les instances retournées.

```
SELECT i.name, p.#name
  FROM Item AS i, #Property AS p
 WHERE p.#oid = ANY (TYPEOF(i.oid).#usedProperties)
```

Explication. Cette requête effectue la jointure entre les instances de la classe Item (niveau données) et les propriétés des ontologies (niveau ontologie). La condition de jointure est que la propriété p soit l'une de celles utilisées (#usedProperties) dans l'extension de la classe de base de l'instance i. Si la propriété name est la seule propriété utilisée pour décrire les instances de la classe Item et que pour la classe Post ce sont les propriétés title et has creator, alors, le résultat de cette requête est le suivant :

i.name	p.#name
'The SIOC Ontology'	'name'
'SIOC Ontology RDF File'	'title'
'SIOC Ontology RDF File'	'has creator'

L'opérateur TYPEOF permet ainsi de combiner les langages d'interrogation définis pour rechercher les ontologies et les données d'une BDBO.

4 Analyse critique du langage OntoQL et perspectives d'évolution

Le langage OntoQL étant présenté, nous pouvons à présent l'évaluer par rapport aux exigences définies au chapitre 2.

4.1 Analyse du langage OntoQL par rapport aux exigences définies

Exigence 1 (Expression de requêtes niveau ontologique)

Le langage OntoQL permet d'exprimer des requêtes sur les données à partir des ontologies, indépendamment du schéma des données, avec une puissance d'expression proche de SQL99 (cf. chapitre 3, section 3).

Une comparaison du pouvoir d'expression de plusieurs langages définis pour RDF a été proposée dans [Haase et al., 2004] sur un échantillon de requêtes. L'expression de ces requêtes en OntoQL est présentée dans l'annexe B. Cette étude montre que le langage OntoQL propose les opérateurs suivants, peu supportés par les autres langages considérés dans l'étude :

- les opérateurs d'agrégat (GROUP BY) et de tri (ORDER BY) ;
- les opérateurs sur les collections (par exemple, l'accès à un élément donné d'une collection) ;
- les opérateurs permettant d'exploiter le multilinguisme (rechercher une valeur dans une langue naturelle donnée) ;

– les opérateurs sur les types de données (par exemple, les opérateurs arithmétiques sur les entiers).

Par contre, il ne permet pas d'exprimer des requêtes récursives ainsi que les requêtes qui nécessitent de prendre en compte certaines particularités de RDF telles que la réification où le fait que tout élément manipulé constitue une ressource en RDF. Nous avons également vu qu'il ne permet pas d'exprimer des requêtes non typées.

Exigence 2 (Définition de concepts non canoniques)

Des classes non canoniques peuvent être créées en utilisant le LDD. L'extension de cette classe est définie à partir d'une requête OntoQL à l'image des vues en SQL (cf. chapitre 3, section 4).

Par rapport au langage RVL, la principale différence est que le langage OntoQL permet d'organiser les classes canoniques et non canoniques dans la même hiérarchie. Dans RVL, ces deux types de concepts sont clairement distingués pour respecter le principe d'indépendance logique. Pour nous, les classes canoniques et les classes non canoniques ne forment qu'un seul et même niveau : le niveau ontologique. Notons que ne pas distinguer ces deux types de concepts évite de devoir importer les concepts canoniques, nécessaires à la définition des concepts non canoniques, dans un nouvel espace de noms.

Une autre différence est que dans le langage OntoQL, lorsque les concepts non canoniques peuvent être mis à jour, ces mises à jour sont répercutées sur les concepts canoniques correspondants. Nous avons fait ce choix pour rester conforme à la sémantique des modèles d'ontologies qui offrent des constructeurs de concepts non canoniques comme par exemple OWL. En RVL, les concepts non canoniques peuvent toujours être mis à jour. Par contre, ces mises à jour ne sont pas répercutées sur les concepts à partir desquels ces concepts non canoniques ont été définis.

Enfin, par rapport à la problématique de définir la couche OCNC d'une ontologie, ces deux langages présentent les limitations évoquées à la fin de la section 4 du chapitre 3, c'est-à-dire essentiellement de ne pas permettre le placement automatique des classes non canoniques dans la hiérarchie de classes.

Exigence 3 (Exploitation linguistique)

Les modèles de données définis pour exploiter les ontologies et leurs instances prennent en compte le multilinguisme. En effet, les ontologies sont exploitées par rapport à un modèle d'ontologies noyau dont les noms d'entités et d'attributs ainsi que les valeurs d'attributs peuvent être définis dans plusieurs langues naturelles (cf. section 2.5). De même, les instances sont exploitées par rapport aux ontologies dont les noms des classes et des propriétés ainsi que les valeurs de propriétés peuvent également être définis dans plusieurs langues naturelles (cf. chapitre 3, section 5). Les instructions OntoQL peuvent ainsi être exprimées dans différentes langues naturelles et traiter des données caractérisées dans différentes langues naturelles. Cette capacité n'est cependant possible que si (1) les noms des classes sont uniques pour un espace de noms et une langue naturelle donnés et si (2) les noms des propriétés sont uniques pour une classe, un espace de noms et une langue naturelle donnés (cf. section 5 du chapitre 3).

SPARQL est un des rares langages qui permet également d'exploiter le multilinguisme. En effet, comme OntoQL, il permet de rechercher une valeur de propriétés ou d'attributs dans différentes langues naturelles. Il propose également les fonctions `lang` et `lang-matches` qui permettent de retrouver la langue naturelle dans laquelle une chaîne de caractères est définie. Ces fonctions sont particulièrement

utiles lorsque l'on utilise une propriété multilingue dans une projection et que l'on souhaite connaître la langue naturelle dans laquelle chaque valeur retournée est définie. Ces fonctions ne sont pas proposées par OntoQL car il ne permet de réaliser une projection impliquant une propriété multilingue que par rapport à une langue naturelle donnée. Par contre, contrairement à OntoQL, SPARQL ne permet pas d'exprimer des requêtes multilingues car il est basé sur le modèle RDF qui n'impose pas les contraintes permettant de proposer cette fonctionnalité.

Exigence 4 (Indépendance par rapport à un modèle d'ontologies donné)

Le langage OntoQL est construit sur un modèle noyau contenant les constructeurs communs à différents modèles d'ontologies (cf. section 2.1). Ce modèle noyau peut être étendu en y ajoutant de nouveaux constructeurs afin de prendre en compte les particularités d'un modèle d'ontologies donné. Cette extension est principalement structurelle. Ceci permet de représenter dans une BDBO l'ensemble des descriptions fournies par une ontologie quel que soit le modèle d'ontologies selon lequel elle est définie. Niveau sémantique, les constructeurs ajoutés ne peuvent qu'hériter de la sémantique des constructeurs définie dans le modèle noyau. Actuellement, cette sémantique, cablée dans le modèle noyau, ne peut pas être étendue automatiquement.

Par contre, le langage OntoQL permet de définir manuellement cette sémantique en utilisant les différents langages qu'il propose. Par exemple, la sémantique du constructeur OWL `HasValue` est que les instances d'une telle restriction sont celles qui présentent une valeur donnée pour une propriété donnée. Nous avons vu que cette sémantique pouvait être représentée en créant une vue (cf. chapitre 3, section 4.3). De plus, comme le montre la figure 4.3, la définition de ces vues peut être automatisée. En effet, une requête OntoQL sur les ontologies peut être exprimée pour retrouver chaque restriction `HasValue` ainsi que les informations permettant d'en créer l'extension, c'est-à-dire la propriété et la valeur sur laquelle elle porte. Pour chaque ligne du résultat, une instruction `CREATE VIEW` peut être générée pour créer l'extension de la restriction correspondante.

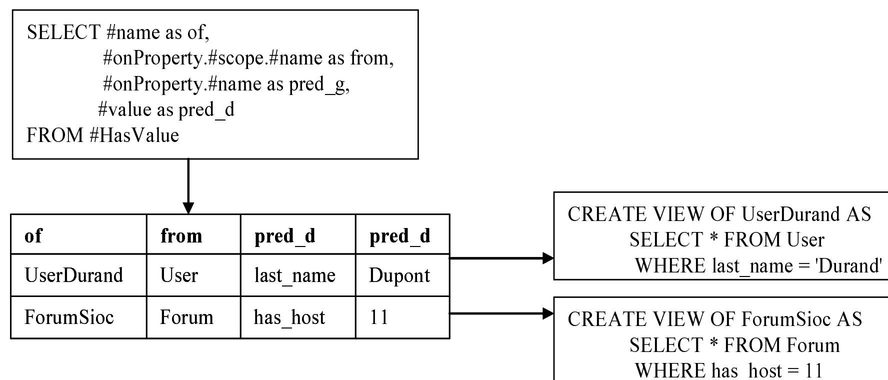


FIG. 4.3 – Automatisation du calcul de l'extension des classes non canoniques `HasValue`

Contrairement au langage SPARQL, qui est indépendant d'un modèle d'ontologies particulier mais qui les considère comme un ensemble de triplets, sans sémantique spécifique, le langage OntoQL se base sur un modèle noyau auquel une sémantique cablée est associée et qui peut être étendu via des instructions du langage.

Exigences 5 et 6 (Compatibilité avec l'architecture traditionnelle des bases de données)

OntoQL propose les principaux opérateurs de SQL92 pour permettre de manipuler les données au niveau logique d'une BDBO (cf. chapitre 3, section 2). A ce niveau, les données sont stockées dans un schéma résultant de l'implantation des extensions des classes. OntoQL permet de connaître les propriétés utilisées pour créer l'extension d'une classe (`#usedProperties`). Il permet également de forcer l'implantation de cette extension comme une représentation horizontale (cf. chapitre 3, section 3.3.2). Il ne permet pas, pour l'instant, de forcer une autre implantation.

Par rapport à CQL, la syntaxe du langage OntoQL permettant de manipuler les données au niveau logique respecte strictement celle de SQL. De plus, OntoQL distingue clairement l'accès au niveau ontologique (à partir des classes) de l'accès au niveau logique (à partir des tables), alors que dans CQL, ces deux niveaux ne sont pas distingués.

Exigence 7 (Définition et manipulation des ontologies)

Deux langages permettent de créer les éléments d'une ontologie : le LDD (cf. chapitre 3, section 3.3) et le LMO (cf. section 2.4). L'avantage du LDD est de proposer la syntaxe habituelle de création d'un conteneur. Celui du LMO est d'offrir un pouvoir d'expression supérieur, étant lié au LIO. En particulier, l'instruction `INSERT INTO SELECT` permet de créer les éléments d'une ontologie dans une autre ontologie. Elle permet également de réaliser des transformations de modèles [Jean et al., 2007b].

Comparé au LDD proposé par RVL, OntoQL permet de définir la valeur de tous les attributs qui permettent de caractériser une classe et une propriété. RVL permet seulement de créer une classe en indiquant son identifiant et ses super-classes. Pour les propriétés, il ne permet que de préciser son identifiant, ses éventuelles super-propriétés et son domaine et codomaine. De plus, OntoQL permet de créer d'autres éléments que des classes et des propriétés dans une ontologie (par exemple, les documents PLIB). Enfin, à notre connaissance, il est le seul à proposer un langage pour manipuler le modèle d'ontologies utilisé (cf. section 2.3). Ce langage permet d'ajouter de nouvelles entités (par exemple, les restrictions OWL) ou de nouveaux attributs (par exemple, les attributs `note` et `remark` de PLIB) à ce modèle d'ontologies.

Exigence 8 (Définition et manipulation des données)

OntoQL propose un langage de définition de données permettant de créer les classes et les propriétés selon une syntaxe similaire à celle de SQL (cf. chapitre 3, section 3.3). Il permet également d'associer des instances à ces classes. En OntoQL, une instance n'appartient qu'à une seule classe de base qui est canonique. Elle peut cependant appartenir à d'autres classes non canoniques via le mécanisme des vues. La manipulation de ces instances peut être faite avec un LMD similaire à celui de SQL (cf. chapitre 3, section 3.4). Ce LMD peut être utilisé pour mettre à jour une instance à partir de sa classe de base. Ces mises à jour sont alors automatiquement répercutées sur les classes non canoniques grâces au mécanisme de vue. Quand ces vues peuvent être mises à jour, le LMD peut aussi être utilisé pour mettre à jour une instance à partir des classes non canoniques (cf. chapitre 3, section 4.2). A nouveau, ces mises à jour sont répercutées sur les autres classes auxquelles elle appartient.

L'approche proposée par RUL est différente. Une instance peut appartenir à plusieurs classes de base sans que celles-ci n'aient un lien entre elles. RUL permet de manipuler une instance par rapport à une classe sans que cela n'ait de conséquence sur ses autres classes d'appartenance. Par exemple, une

instance peut être supprimée d'une classe sans qu'elle ne soit supprimée des autres classes auxquelles elle appartient. Pour cela, il propose de manipuler les instances à travers les liens d'instanciation. D'autre part, RUL sépare la manipulation d'une instance par rapport aux classes de la mise à jour de ses valeurs de propriétés en proposant deux instructions différentes.

Ces deux langages diffèrent donc sur leur approche de la multi-instanciation qui permet de représenter différents points de vue possibles sur une instance. OntoQL permet la multi-instanciation via le mécanisme des vues. Par contre, il impose l'unicité de la classe de base afin que la structure possible d'une instance soit fixée et afin de permettre un stockage efficace de ces instances. Ce n'est pas le cas de RUL qui impose, en conséquence, que les instances soient manipulées et stockées indépendamment de leurs valeurs de propriétés.

Exigence 9 (Interrogation des ontologies)

Le langage OntoQL propose un langage d'interrogation pour les ontologies d'une BDBO, le LIO (cf. section 2.5). Ce langage propose les mêmes opérateurs que le LID permettant d'interroger les données à base ontologique. En conséquence, la comparaison entre le pouvoir d'expression de ce langage avec celui des autres langages proposés dans la littérature est la même que celle que nous avons faite pour le LID. Notons cependant, qu'en plus, il permet d'interroger les ontologies en utilisant des attributs dont le calcul nécessite un pouvoir d'expression supérieur à SQL. Ceci permet d'exprimer des requêtes typiques sur les ontologies comme par exemple rechercher les sous-classes (directes ou non) d'une classe ou rechercher les propriétés (définies ou applicables) sur une classe.

Exigence 10 (Interrogation à la fois des ontologies et des données)

OntoQL permet d'exprimer des requêtes portant à la fois sur les ontologies et sur les données. Pour naviguer dans une BDBO des ontologies vers les données, il permet d'utiliser des itérateurs à liaison dynamique (`C AS i`) afin de parcourir les instances d'une classe déterminée à l'exécution de la requête. De plus, il permet d'utiliser des propriétés déterminées à l'exécution d'une requête dans une projection (`i.p`). Pour naviguer des données vers les ontologies, OntoQL propose l'opérateur `TYPEOF` qui permet d'obtenir la classe de base d'une instance à partir de son identifiant. Cet opérateur peut ainsi être utilisé pour réaliser des jointures entre les éléments des ontologies et les instances des classes.

Le pouvoir d'expression proposé par OntoQL pour interroger à la fois les ontologies et leurs instances est inférieur à celui de RQL qui permet d'utiliser des expressions de chemins généralisées. Cette capacité permet notamment de construire une expression de chemin contenant plusieurs propriétés déterminées dynamiquement à l'exécution d'une requête. Par exemple, l'expression RQL `@P.@P` permet de rechercher les valeurs d'instances pour n'importe quelle expression de chemin de taille 2. L'expression de telles requêtes en OntoQL n'est pas possible. Néanmoins, le pouvoir d'expression qu'il propose s'est montré suffisant pour les projets dans lesquels nous l'avons utilisé.

4.2 Perspectives d'évolution du langage OntoQL

Nous présentons dans cette section les principales évolutions du langage OntoQL que nous envisageons de réaliser.

Association de plusieurs extensions à une classe

Le modèle de données du niveau ontologique, couche OCC ne permet de définir qu'une seule extension par classe (cf. chapitre 3, section 3.1). Cependant, la capacité de pouvoir associer plusieurs extensions à une classe peut être particulièrement utile dans des approches d'intégration matérialisées de type entrepôt de données pour éviter d'avoir à fusionner les extensions provenant de différentes sources associées à une même classe [Nguyen-Xuan, 2006]. En conséquence, la possibilité d'associer plusieurs extensions à une même classe fait partie des extensions que nous prévoyons de définir sur ce modèle de données et le langage associé.

Choix de la représentation logique d'une extension de classe

L'extension d'une classe est implantée au niveau logique selon une des représentations proposées par les BDBO (horizontale, verticale et binaire). Le langage OntoQL permet uniquement d'indiquer dans une instruction de création d'une extension qu'elle doit être implantée selon la représentation horizontale (cf. chapitre 3, section 3.3.2). Or, les évaluations de performance réalisées entre ces différentes représentations ont montré que la représentation horizontale n'est pas toujours la meilleure représentation, notamment lorsque peu de propriétés sont utilisées dans une requête. Nous prévoyons donc de donner la possibilité de créer une extension de classe en indiquant que la représentation binaire ou verticale doit être utilisée.

De plus, les relations créées pour implanter les extensions des classes ne sont pas en troisième forme normale. En effet, elles peuvent contenir des attributs multivalués et aucune dépendance fonctionnelle n'est exprimée sur ces relations. Pour bénéficier des avantages de la normalisation (limiter la redondance, etc.), nous prévoyons de modifier le langage OntoQL pour qu'il permette de créer automatiquement les extensions des classes comme un ensemble de tables en troisième forme normale.

Création de propriétés dérivées

Le langage OntoQL permet de créer des *attributs dérivés*, c'est-à-dire dont les valeurs sont calculées à partir des valeurs d'autres attributs. Ceci permet de fournir des attributs utiles pour découvrir les ontologies stockées dans une BDBO (cf. section 2.3). Par contre, il ne fournit pas cette capacité pour faciliter la recherche des données à base ontologique. Nous prévoyons donc d'étendre le langage OntoQL pour qu'il permette de définir des *propriétés dérivées*. Ceci permettrait de représenter certaines propriétés non canoniques d'une ontologie comme par exemple les propriétés inverses.

Généralisation de l'utilisation des propriétés déterminées dynamiquement à l'exécution d'une requête

Le langage OntoQL permet d'utiliser une propriété déterminée dynamiquement à l'exécution d'une requête dans la clause `SELECT` d'une requête (cf. section 3). Il pourrait être utile de pouvoir utiliser ces propriétés pour, par exemple, filtrer les instances selon les valeurs de ces propriétés. Nous prévoyons donc de permettre l'utilisation des propriétés déterminées dynamiquement à l'exécution d'une requête dans les différentes clauses d'une requête OntoQL. Notons que cette possibilité est offerte par le langage SchemaSQL. L'extension du langage OntoQL avec cette capacité pourra donc s'appuyer sur la syntaxe

et la sémantique de ce langage [Lakshmanan et al., 2001] ainsi que sur les techniques développées pour l'implanter efficacement [Lakshmanan et al., 1999].

Interrogation des données à la fois au niveau logique et ontologique

L'opérateur TYPEOF de OntoQL permet de combiner les langages d'interrogation définis pour rechercher les ontologies et les données d'une BDBO (cf. section 3.2). Nous envisageons également de proposer un opérateur afin de permettre de combiner les langages d'interrogation définis pour rechercher les données au niveau logique et ontologique. Ceci permettrait par exemple de rechercher la table relationnelle dans laquelle une instance de classe est stockée.

Autres composants d'un langage d'exploitation

Jusqu'à présent, nous nous sommes focalisés sur la définition du LDD, LMD et LID de OntoQL. Il nous reste à définir les autres composants d'un langage de bases de données. Nous pensons notamment à la gestion des autorisations et des transactions dans une BDBO, problèmes qui, à notre connaissance, n'ont pas été abordés dans la littérature. Ces aspects sont pourtant fondamentaux pour un SGBD. De plus, pour compléter le langage OntoQL, nous avons identifié les actions suivantes :

– définir un langage de spécification de procédures stockées permettant d'exécuter des instructions OntoQL ;
– définir un langage de déclencheurs (*triggers*) ;
– définir des interfaces d'accès avec les langages orientés-objets autres que JAVA (l'interface que nous avons conçue pour JAVA est présentée dans le chapitre 6);
– définir une politique pour gérer les versions, les révisions et plus généralement l'évolution dans les BDBO (elle pourra être mise en place dans le LMD).

5 Conclusion

Dans ce chapitre, nous avons tout d'abord présenté les traitements spécifiques aux BDBO proposés par le langage OntoQL. Ce langage permet d'exploiter les ontologies d'une BDBO indépendamment d'un modèle d'ontologies particulier. Il est en effet conçu autour d'un modèle d'ontologies noyau constitué des constructeurs partagés par les différents modèles d'ontologies. Pour prendre en compte les spécificités d'un modèle d'ontologies particulier, ce noyau peut être étendu par des instructions du langage en ajoutant des entités et des attributs, éventuellement dérivés. Les instructions des langages permettant d'exploiter les ontologies ont une syntaxe qui reste proche de SQL. Le langage OntoQL propose ainsi une syntaxe uniforme pour exploiter les ontologies d'une BDBO et leurs instances.

De plus, le langage OntoQL permet d'interroger à la fois les ontologies et les instances des classes. Pour permettre de rechercher des classes et des propriétés et en même temps d'obtenir les instances de ces classes et leurs valeurs de propriétés, nous avons introduit deux mécanismes issus des langages proposés pour les bases de données fédérées. Le premier mécanisme permet d'introduire un itérateur à liaison dynamique. Le second permet d'utiliser une propriété déterminée à l'exécution d'une requête dans une projection. Ces deux mécanismes permettent ainsi de rechercher des classes puis de retourner

leurs instances. Pour permettre une navigation dans le sens inverse, nous avons introduit l'opérateur `TYPEOF`. Cet opérateur permet d'obtenir la classe de base d'une instance, permettant, ainsi, de faire le lien entre les données à base ontologique et les classes des ontologies.

Ayant terminé la présentation du langage OntoQL, nous en avons proposé une analyse critique en l'évaluant par rapport aux exigences définies au chapitre 2 et en présentant ses perspectives d'évolution. Le langage OntoQL propose une solution pour les différentes exigences définies dans le chapitre 2 (cf. partie suivante pour les exigences 11 et 12). Les principales nouveautés qu'il propose permettent :

1. d'étendre le modèle d'ontologies noyau sur lequel il est défini. Même si cette extension n'est essentiellement que structurelle et descriptive, ceci permet de stocker des ontologies dans une BDBO quel que soit le modèle d'ontologies utilisé tout en fournissant un LDO, LMO et LIO pour exploiter ces ontologies. De plus, il est possible d'associer, par programme, une sémantique opérationnelle à ces extensions ;
2. d'exprimer des requêtes dans différentes langues naturelles. Cette capacité permet à un utilisateur quelle que soit la langue qu'il utilise d'écrire des requêtes OntoQL beaucoup plus simplement qu'en utilisant les identifiants associés aux concepts d'une ontologie ;
3. d'utiliser une syntaxe et une sémantique proche (et compatible) de SQL pour accéder aux différentes couches d'une BDBO. Ceci permet de rendre l'utilisation du langage OntoQL plus naturelle pour les utilisateurs de bases de données et d'en permettre une interprétation plus aisée sur les SGBD ;
4. de manipuler les données au niveau logique d'une BDBO. Ceci autorise la modification de l'implantation logique du niveau ontologique exactement comme SQL permet de modifier l'implantation physique du niveau logique (ajouter des index etc.) ;
5. de définir et manipuler les données et les ontologies avec des langages de définition et de manipulation de données. Si les propositions de LID sont nombreuses pour les BDBO, ce n'est pas le cas pour les langages de définition et de manipulation de données. De tels langages sont pourtant essentiels dans le contexte d'une base de données car ils offrent beaucoup plus de flexibilité qu'une API ;
6. permettre de manipuler explicitement le modèle conceptuel des données en définissant les extensions associées à chaque classe d'une ontologie.

La syntaxe complète du langage OntoQL est présentée dans l'annexe A. Des démonstrations sont disponibles sous forme de vidéos à l'adresse `http://www.plib.ensma.fr/plib/demos/ontodb/index.html`. Afin de valider le langage OntoQL, nous l'avons implanté sur la BDBO OntoDB. Cette implantation est basée sur la sémantique formelle de ce langage présentée au chapitre suivant.

Troisième partie

Validation théorique et opérationnelle du langage OntoQL

Chapitre 5

Sémantique formelle du langage OntoQL

Sommaire

1 Introduction 165

2 Définition formelle du modèle de données d'une BDBO 165

2.1 Modèle de données *Encore* d'une BDOO 165

2.2 Modèle de données d'accès aux données d'une BDBO 166

2.3 Modèle de données d'accès aux ontologies d'une BDBO 169

2.4 Modèle de données d'accès aux ontologies et aux données d'une BDBO . 170

3 *OntoAlgebra* : adaptation de l'algèbre *Encore* au modèle de données d'une BDBO 170

3.1 Algèbre *Encore* permettant d'interroger les données d'une BDOO 171

3.2 Algèbre permettant d'interroger les données d'une BDBO 172

3.3 Algèbre permettant d'interroger les ontologies d'une BDBO 179

3.4 Algèbre permettant d'interroger les ontologies et les données d'une BDBO 181

4 Etude du langage OntoQL en utilisant l'algèbre *OntoAlgebra* 183

4.1 Propriétés du langage OntoQL 183

4.2 Optimisation logique de requêtes OntoQL 184

5 Conclusion 188

Résumé. Dans la partie précédente, nous avons présenté la syntaxe du langage OntoQL. Dans ce chapitre, nous présentons la sémantique formelle de ce langage [Jean et al., 2007a]. Cette sémantique consiste en une algèbre d'opérateurs basée sur une représentation formelle du modèle de données d'une BDBO. Pour construire ce modèle de données et cette algèbre d'opérateurs, nous avons adapté et étendu l'algèbre *Encore* définie pour les BDOO. Cette démarche nous garantit la *fermeture* et la *complétude relationnelle* du langage OntoQL et nous a permis de proposer des techniques d'optimisation pour les requêtes écrites dans ce langage. Cette formalisation constitue, de plus, une base formelle pour l'implantation de ce langage sur une BDBO.

1 Introduction

Dans la partie précédente, nous avons présenté le langage OntoQL au travers de sa syntaxe, en montrant les différentes utilisations offertes. Ce langage permet d'interroger les données d'une BDBO au niveau logique en étant compatible avec le langage SQL. La sémantique de ce sous-ensemble du langage OntoQL est définie formellement par l'algèbre relationnelle. Dans ce chapitre, nous présentons la sémantique formelle du sous-ensemble du langage OntoQL permettant d'interroger les données au niveau ontologique, les ontologies et à la fois les données et les ontologies d'une BDBO [Jean et al., 2007a]. Il s'agit en fait de présenter le modèle algébrique associé à la manipulation du niveau ontologique de l'architecture de bases de données ANSI/SPARC étendue.

Le modèle de données défini pour le langage OntoQL utilisant de nombreux constructeurs orientés-objets, plutôt que de définir la sémantique du langage OntoQL à partir des éléments de base, nous avons choisi de la fonder sur une algèbre proposée pour les BDOO. Notre choix s'est porté sur l'algèbre *Encore* [Zdonik and Mitchell, 1991], très citée dans la littérature, qui présente la particularité de rester proche de l'algèbre relationnelle. Ceci permet d'obtenir une sémantique uniforme pour les différents niveaux d'interrogation proposés par le langage OntoQL. Cette sémantique peut, de plus, comme nous le montrons dans ce chapitre, être utilisée pour établir la *fermeture* et la *complétude relationnelle* du langage OntoQL et pour étudier l'optimisation de requêtes écrites dans ce langage.

Ce chapitre est organisé comme suit. Dans la section suivante nous présentons la définition formelle du modèle de données d'une BDBO en montrant comment nous l'avons défini à partir de celui d'une BDOO. Dans la section 3, nous définissons l'algèbre du langage OntoQL nommée *OntoAlgebra* basée sur le modèle de données d'une BDBO et construite à partir de l'algèbre d'*Encore*. La section 4 montre comment cette algèbre peut être utilisée pour établir des propriétés du langage OntoQL et étudier l'optimisation de requêtes écrites dans ce langage. Nous concluons ce chapitre à la section 5 sur les perspectives de recherche qu'ouvre la définition de cette algèbre et sur son intérêt pour l'implantation du langage OntoQL.

2 Définition formelle du modèle de données d'une BDBO

Pour définir la sémantique du langage OntoQL, il est nécessaire de définir formellement le modèle de données d'une BDBO. Cette section présente la conception de ce modèle de données à partir de celui d'*Encore*.

2.1 Modèle de données *Encore* d'une BDOO

Formellement, dans le modèle de données *Encore*, une BDOO est définie comme un 8-uplet <ADT, O, Property, directSuperTypes, typeO, scope, range, valO>, dont chaque élément est décrit de la façon suivante :

– ADT est un ensemble de types abstraits de données. Il contient les types primitifs (Int, String, etc.), un super-type global Object et des types utilisateurs ;

- O est un ensemble d'objets contenus dans la base de données ou qui peuvent être construits par une requête. Tous les objets ont un identifiant unique ;
- Property est un ensemble de propriétés utilisées pour décrire l'état de chaque objet ;
- directSuperTypes : ADT $\rightarrow 2^{ADT}$ est une fonction partielle[27]. Elle associe un type à l'ensemble de ses super-types directs. Cette fonction définit un treillis de types. Sa relation d'ordre est l'héritage et elle respecte le principe de substitution ;
- typeO : O $\rightarrow$ ADT associe à chaque objet le type minorant pour la relation de subsomption auquel il appartient ;
- scope : Property $\rightarrow$ ADT et range : Property $\rightarrow$ ADT définissent respectivement le domaine et le codomaine de chaque propriété ;
- valO : O $\times$ Property $\rightarrow$ O donne la valeur d'un objet pour une propriété. Cette propriété doit être définie sur le type de l'objet.

Ce modèle de données supporte les collections d'objets en fournissant les types paramétriques SET[T] et MULTISET[T]. SET[T] représente un type collection d'objets sans doublon de type T. $\{o_1, \ldots, o_n\}$ référence un objet de ce type où les o_i sont des objets de type T. Le type MULTISET autorise les doublons. Un autre ADT paramétrique, nommé TUPLE, permet de créer des relations entre objets. Un type TUPLE est construit en fournissant un ensemble de noms d'attributs (A_i) et de types d'attributs (T_i). TUPLE$[< (A_1, T_1), \ldots, (A_n, T_n) >]$ représente un type tuple construit en utilisant les noms d'attributs A_i et les types d'attributs T_i. $< A_1 : o_1, \ldots, A_n : o_n >$ est un objet de ce type (un tuple) où les o_i sont des objets du type correspondant T_i. Le type TUPLE est défini avec les fonctions get_A_i_value permettant de récupérer la valeur d'un tuple o pour l'attribut A_i. L'appel de ces fonctions peut être abrégé en utilisant la notation pointée ($o.A_i$). Le type TUPLE est fondamental pour construire de nouveaux types. En particulier, il permet de décrire des types de données qui ne sont pas disponibles dans le schéma de la base de données et qui peuvent être construits à partir d'expressions de l'algèbre *Encore*.

La plupart des langages de BDOO ne permettent que d'interroger les données d'une BDOO. Ils ne permettent pas d'interroger les schémas contenus dans une BDOO. En conséquence, le modèle de données *Encore* ne concerne que les données d'une BDOO. Or, le langage OntoQL permet également d'interroger les ontologies et à la fois les ontologies et les données d'une BDBO. Il est donc nécessaire de formaliser le modèle de données permettant ces différents types d'interrogation d'une BDBO.

2.2 Modèle de données d'accès aux données d'une BDBO

Le modèle de données *Encore* ne peut pas être utilisé pour une BDBO sans adaptation. En effet, alors que le modèle *Encore* est conçu selon une approche orientée-objet afin de permettre d'accéder aux données depuis un schéma orienté-objet, le modèle de données du langage OntoQL est basé sur le modèle relationnel-objet et permet d'accéder aux données depuis une ontologie. En conséquence, le modèle de données *Encore* doit être adapté d'une part, à l'approche relationnelle-objet, et, d'autre part, aux différences entre une ontologie et un schéma orienté-objet.

Les approches relationnelles-objets et orientées-objets diffèrent sur leur façon de voir les données. Dans l'approche orientée-objet, toute donnée est considérée comme un objet ayant un identifiant géré par

[27]Nous utilisons le symbole 2^C pour représenter l'ensemble des parties de C.

le système. Cet identifiant est utilisé pour déterminer si deux objets sont égaux. L'approche relationnelle-objet est différente. Seules les instances des types utilisateur ont un identifiant qui peut être manipulé par un utilisateur. Les instances des autres types de données sont considérées comme des valeurs sans identifiant. Le modèle de données d'une BDBO doit donc distinguer les instances des classes, qui possèdent un identifiant, des instances des autres types de données, qui n'en possèdent pas.

D'autre part, nous avons vu que la principale différence entre un schéma orienté-objet et une ontologie est qu'une ontologie décrit les données alors qu'un schéma orienté-objet les prescrit. Cette distinction nous a conduit à introduire la notion d'extension. Chaque classe peut être associée à une extension qui en stocke les instances. Cette extension peut ne comprendre qu'un sous-ensemble des propriétés applicables sur cette classe. Le modèle de données d'une BDBO doit donc également représenter la notion d'extension.

En prenant en compte ces adaptations, le modèle de données qui permet d'accéder aux données d'une BDBO peut-être défini comme un 13-uplets < ADT_C, V_C, Class, I, Property, directSuperClasses, scope, range, Extent, usedProperties, nomination, typeI, valI> où :

- ADT_C contient les types primitifs (Int, String, etc.), les classes des ontologies (Class) et les types paramétriques SET, MULTISET, TUPLE, REF et ARRAY (ces deux derniers types sont décrits ultérieurement) ;
- V_C est l'ensemble des valeurs des types de ADT_C. Il contient les instances des classes (avec un identifiant) et les valeurs des types primitifs et paramétriques (sans identifiant) ;
- Class est l'ensemble des classes de l'ontologie. Comme dans la plupart des modèles d'ontologies, nous supposons l'existence d'une classe racine ObjectC ;
- I est l'ensemble des instances des classes de l'ontologie. Chaque instance possède un identifiant unique dont la valeur est donnée par la propriété oid ;
- Property est l'ensemble des propriétés de l'ontologie. Il contient la propriété oid définie sur la classe racine ObjectC. Cette propriété retourne un entier du type primitif Oid ;
- directSuperClasses : Class $\rightarrow 2^{\text{Class}}$ associe une classe à ses super-classes directes ;
- scope : Property → Class et range : Property → ADT_C définissent respectivement le domaine et le codomaine de chaque propriété ;
- Extent est l'ensemble des extensions des classes de l'ontologie ;
- usedProperties : Extent $\rightarrow 2^{\text{Property}}$ retourne les propriétés utilisées pour décrire les instances d'une classe (l'ensemble des propriétés qui ont une valeur pour ses instances) ;
- nomination : Class → Extent est une fonction partielle. Elle associe une classe à son éventuelle extension. L'ensemble des propriétés utilisées dans l'extension d'une classe doit être un sous-ensemble des propriétés applicables sur cette classe ;
- typeI : I → Extent associe à chaque instance l'extension de la classe à laquelle elle appartient ;
- valI : I × Property → V_C donne la valeur d'une instance i pour une propriété p qui doit être utilisée dans l'extension de sa classe de base ($p \in \text{usedProperties}(\text{typeI}^{-1}(i))$).

Nous avons introduit deux nouveaux types paramétriques dans ce modèle de données : les types ARRAY et REF définis dans le modèle relationnel-objet. ARRAY permet de construire des collections dont

les éléments sont indexés. Il est défini avec la fonction `get_value(index)` qui permet de retrouver l'élément d'une collection à un index donné. L'utilisation de cette fonction peut être abrégée en utilisant la notation `v[index]` où `v` est une valeur de type `ARRAY`. `REF[C]` est un type de données constitué de l'ensemble des identifiants des instances d'une classe `C`. Il est défini avec la fonction `DEREF` : `REF[C]` → `C` qui retourne une instance d'une classe à partir de son identifiant.

Ce modèle de données suivant l'approche relationnelle-objet, les tuples et les collections n'ont pas d'identifiant. Il est donc nécessaire de préciser comment est défini l'opérateur d'égalité sur ces valeurs. Deux tuples sont égaux si et seulement si ils ont les mêmes valeurs d'attributs. Deux collections sont égales si et seulement si elles ont la même cardinalité et si ses éléments sont deux à deux égaux. Pour les collections indexées (`ARRAY`), les éléments aux mêmes index doivent être égaux.

Exemple. Pour illustrer le modèle de données défini dans cette section, nous présentons la formalisation de l'exemple présenté sur la figure 5.1. Pour simplifier, nous n'avons représenté sur cette figure que les principaux éléments concernant la classe `Item` de l'ontologie SIOC. Cette classe est associée à une extension nommée par commodité `Extent_Item`. Elle est constituée des propriétés `oid` et `name` et contient une instance. La représentation de ces données en utilisant le modèle de données présenté dans cette section est la suivante :

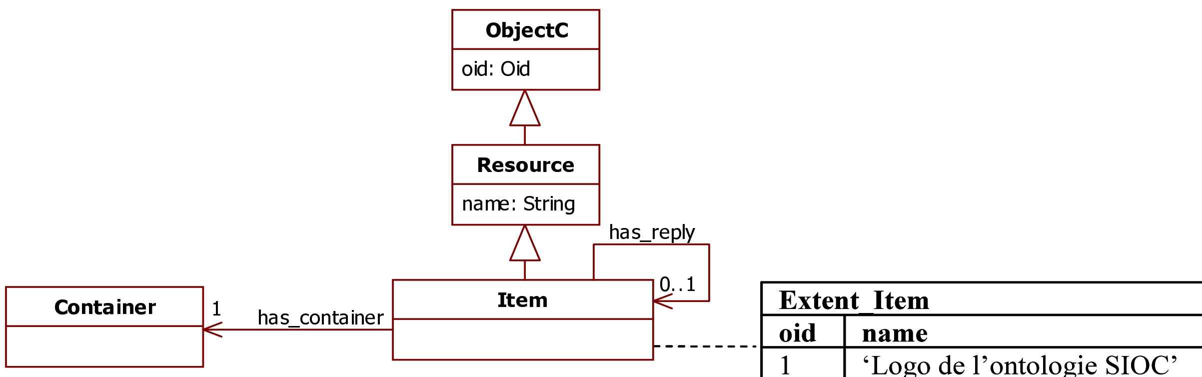


FIG. 5.1 – Exemple de formalisation de la classe `Item` de l'ontologie SIOC

- ADT_C = `{Oid, String, REF[Item], REF[Container], Item, Container, Resource, ObjectC }` ;
- V_C = `{ 1, 'Logo de l'ontologie SIOC', $1 }` où `$1` est l'instance représentée par l'ensemble des couples `(propriété, valeur)` suivant `{ (oid, 1), (name, 'Logo de l'ontologie SIOC') };`
- `Class = { Item, Container, Resource, ObjectC };I = { $1 };`
- `Property = { oid, name, has_reply, has_container };`
- `directSuperClasses(Item) = { Resource } ;`
- `scope(oid) = ObjectC;range(oid) = Oid;`
- `scope(name) = Resource;range(name) = String;`
- `scope(has_reply) = Item;range(has_reply) = REF[Item];`

– scope(has_container) = Item ; range(has_container) = REF[Container] ;
– EXTENT = { Extent_Item } ; usedProperties(Extent_Item) = { oid, name } ;
– nomination(Item) = Extent_Item ; typeI($1) = Extent_Item ;
– valI($1, oid) = 1 ; valI($1, name) = 'Logo de l'ontologie SIOC'.

2.3 Modèle de données d'accès aux ontologies d'une BDBO

Un modèle d'ontologies peut être représenté sous la forme d'un schéma orienté-objet. En conséquence, le modèle de données permettant d'accéder aux ontologies d'une BDBO ne présente pas de différence majeure avec le modèle de données *Encore*. En suivant la formalisation du modèle *Encore* adaptée à l'approche relationnelle-objet, ce modèle de données peut être défini formellement par un 10-uplet < ADT_E, V_E, E, OC, A, directSuperEntities, typeOC, attScope, attRange, valOC> où :

– ADT_E contient les types primitifs (Int, String, etc.), les entités du modèle d'ontologies et les types paramétriques SET, MULTISET, TUPLE, REF et ARRAY ;
– V_E est l'ensemble des valeurs des types de ADT_E ;
– E est l'ensemble des entités du modèle d'ontologies. Nous supposons l'existence d'une entité racine #ObjectE ;
– OC est l'ensemble des éléments des ontologies (les classes, les propriétés, etc.). Ces éléments sont des instances d'une des entités de E ;
– A est l'ensemble des attributs du modèle d'ontologies. Il contient l'attribut #oid qui associe un identifiant à chaque élément d'une ontologie ;
– directSuperEntities : $E \rightarrow 2^E$ associe une entité à ses super-entités directes ;
– typeOC : $OC \rightarrow E$ associe chaque élément de l'ontologie à l'entité dont il est instance ;
– attScope : $A \rightarrow E$ et attRange : $A \rightarrow ADT_E$ définissent respectivement le domaine et le codomaine de chaque attribut ;
– valOC : $OC \times A \rightarrow V_E$ donne la valeur d'un élément d'une ontologie pour un attribut donné.

La principale différence entre ce modèle de données et celui d'*Encore* est qu'il existe un ensemble d'entités et d'attributs prédéfinis. Ces entités et attributs sont ceux du modèle noyau sur lequel le langage OntoQL est basé. Ainsi, E contient en particulier les entités #Class et #Property. Ces entités correspondent aux ensembles du même nom sans le préfixe # que nous avons utilisés pour formaliser le modèle de données permettant d'accéder aux données d'une BDBO. De même, A contient notamment les attributs #scope, #range, #directSuperClasses et #usedProperties correspondants aux fonctions du même nom. Il contient également les attributs dérivés que nous avons présentés au chapitre 4, dans le tableau 4.2.

Exemple. Pour illustrer le modèle de données d'accès aux ontologies, voici un extrait de la formalisation concernant la classe Item de l'ontologie SIOC (cf. figure 5.1). Pour simplifier, nous avons seulement représenté les attributs #oid et #code parmi ceux décrivant la classe Item.

– ADT_E = {Oid, String, #Class, #Concept, #ObjectE }

- V_E = { 100, 'Item', CItem } où CItem est l'instance de l'entité #Class correspondant à la classe Item représentée par l'ensemble des couples (attribut, valeur) suivant {(#oid, 100), (#code, 'Item') };
- E = { #Class, #Concept, #ObjectE } ; OC = { CItem } ;
- A = { #oid, #code } ;
- directSuperEntities(#Class) = { #Concept } ;
- attScope(#oid) = #ObjectE ; attRange(#oid) = Oid ;
- attScope(#code) = #Concept ; attRange(#code) = String ;
- typeOC(CItem) = #Class ; valOC(CItem, #oid) = 100 ; valOC(CItem, #code) = 'Item'.

2.4 Modèle de données d'accès aux ontologies et aux données d'une BDBO

Le langage OntoQL permet d'interroger à la fois les données et les ontologies d'une BDBO dans le sens ontologie vers données, pour extraire des classes et leurs instances, et dans le sens données vers ontologie pour obtenir la description ontologique d'une instance de classe. Le résultat d'une requête peut donc retourner des valeurs de niveau ontologie (V_E) et de niveau données (V_C). Or, dans la formalisation définie, les types tuples ne peuvent être construits qu'à partir des types abstraits de données de niveau ontologie (ADT_E) ou de niveau données (ADT_C). Pour permettre de construire des types tuples comprenant des éléments de ces deux niveaux, nous introduisons l'ensemble $ADT = ADT_E \cup ADT_C$ comme étant l'ensemble des types abstraits d'une BDBO. Les types tuples peuvent être construits à partir de cet ensemble de types. Les valeurs de ces tuples sont dans l'ensemble $V = V_E \cup V_C$.

D'autre part, une requête OntoQL peut réaliser une projection impliquant une propriété déterminée dynamiquement à l'exécution d'une requête. Le résultat d'une telle projection peut être l'union de plusieurs types de données. Nous introduisons donc le type paramétrique UNION afin de permettre de représenter le résultat de ce type de requêtes. $UNION[T_1, \ldots, T_n]$ représente un type dont les valeurs sont l'union de celles des types T_i paramètres.

Enfin, nous avons introduit l'opérateur typeOf afin de permettre de naviguer des données vers les ontologies. Dans notre formalisation, cet opérateur est représenté par la fonction TYPEOF : REF[ObjectC] $\rightarrow$ Class qui retourne une classe (de type Class) à partir de l'identifiant d'une instance (appartient à la classe racine ObjectC).

Les modèles de données permettant d'accéder aux données, aux ontologies et à la fois aux données et aux ontologies d'une BDBO étant définis, nous pouvons maintenant définir l'algèbre d'opérateurs qui permet d'interroger ces différents éléments.

3 *OntoAlgebra* : adaptation de l'algèbre *Encore* au modèle de données d'une BDBO

Pour définir l'algèbre de requêtes d'une BDBO, nous avons adapté les opérateurs de l'algèbre *Encore*.

3.1 Algèbre *Encore* permettant d'interroger les données d'une BDOO

Chaque opérateur de l'algèbre *Encore* prend comme paramètre d'entrée une collection d'objets dont le type est un ADT T, défini en section 2.1. Il peut éventuellement prendre d'autres paramètres d'entrée. Il retourne une collection d'objets dont le type est un ADT Q. Ainsi, la signature d'un opérateur est SET[T] $\times \cdots \rightarrow$ SET[Q]. En suivant notre formalisation, nous utilisons la signature $\mathtt{ADT} \times 2^{\mathtt{O}} \times \cdots \rightarrow \mathtt{ADT} \times 2^{\mathtt{O}}$ de manière à mettre en évidence le type de données et l'ensemble des objets. Nous décrivons maintenant les opérateurs principaux de l'algèbre *Encore*. Pour définir ces opérateurs, nous utilisons les éléments suivants :

- T et R sont deux ADT ;
- $\mathtt{O_t}$ et $\mathtt{O_r}$ sont deux collections d'objets ($\in 2^{\mathtt{O}}$) de type respectif T et R ;
- $\mathtt{f}, \mathtt{f_1}, \ldots, \mathtt{f_n}$ sont des fonctions dont le domaine est le type T ;
- $\mathtt{A_1}, \ldots, \mathtt{A_n}$ sont des chaînes de caractères.

Image. L'opérateur Image retourne une collection d'objets résultant de l'application d'une fonction à une collection d'objets. Sa signature est $\mathtt{ADT} \times 2^{\mathtt{O}} \times \mathtt{Function} \rightarrow \mathtt{ADT} \times 2^{\mathtt{O}}$. Function contient en particulier toutes les propriétés de l'ensemble Property. Une fonction peut également être une composition d'autres fonctions. En étendant le domaine de scope, range et valO de Property à Function, l'opérateur Image est défini par :

$$\mathtt{Image}(\mathtt{T}, \mathtt{O_t}, \mathtt{f}) = (\mathtt{range}(\mathtt{f}), \{\mathtt{valO}(\mathtt{o}, \mathtt{f}) \mid \mathtt{o} \in \mathtt{O_t}\})$$

Project. L'opérateur Project étend Image en permettant l'application de plusieurs fonctions à un objet. Le type du résultat est un TUPLE dont les noms d'attributs sont pris en paramètre. Sa signature est $\mathtt{ADT} \times 2^{\mathtt{O}} \times 2^{\mathtt{String} \times \mathtt{Function}} \rightarrow \mathtt{ADT} \times 2^{\mathtt{O}}$. Il est défini par :

$$\begin{aligned} &\mathtt{Project}(\mathtt{T}, \mathtt{O_t}, \{(\mathtt{A_1}, \mathtt{f_1}), \ldots, (\mathtt{A_n}, \mathtt{f_n})\}) = \\ &\quad (\mathtt{TUPLE}[<(\mathtt{A_1}, \mathtt{range}(\mathtt{f_1})), \ldots, (\mathtt{A_n}, \mathtt{range}(\mathtt{f_n}))>], \\ &\qquad \{<\mathtt{A_1} : \mathtt{valO}(\mathtt{o}, \mathtt{f_1}), \ldots, \mathtt{A_n} : \mathtt{valO}(\mathtt{o}, \mathtt{f_n})> \mid \mathtt{o} \in \mathtt{O_t}\}) \end{aligned}$$

L'opérateur Project applique les fonctions $\mathtt{f_i}$ sur chaque objet de la collection $\mathtt{O_t}$. Le résultat est de type tuple (écriture $\mathtt{TUPLE}[<(\mathtt{A_1}, \mathtt{T_1}), \ldots, (\mathtt{A_n}, \mathtt{T_n})>]$). Chacune de ces valeurs (écriture $<\mathtt{A_1} : o_1, \ldots, \mathtt{A_n} : o_n>$) correspond à l'application des fonctions $\mathtt{f_i}$ à un objet de $\mathtt{O_t}$.

Select. L'opérateur Select crée, par filtration, une collection d'objets qui satisfont un prédicat. Sa signature est $\mathtt{ADT} \times 2^{\mathtt{O}} \times \mathtt{Predicate} \rightarrow \mathtt{ADT} \times 2^{\mathtt{O}}$. Il est défini par :

$$\mathtt{Select}(\mathtt{T}, \mathtt{O_t}, \mathtt{pred}) = (\mathtt{T}, \{\mathtt{o} \mid \mathtt{o} \in \mathtt{O_t} \wedge \mathtt{pred}(\mathtt{o})\})$$

OJoin. L'opérateur OJoin crée une relation entre des objets de deux collections en entrée. Sa signature est $\mathtt{ADT} \times 2^{\mathtt{O}} \times \mathtt{ADT} \times 2^{\mathtt{O}} \times \mathtt{String} \times \mathtt{String} \times \mathtt{Predicate} \rightarrow \mathtt{ADT} \times 2^{\mathtt{O}}$. Il est défini par :

$$\begin{aligned} &\mathtt{OJoin}(\mathtt{T}, \mathtt{O_t}, \mathtt{R}, \mathtt{O_r}, \mathtt{A_1}, \mathtt{A_2}, \mathtt{pred}) = \\ &(\mathtt{TUPLE}[<(\mathtt{A_1}, \mathtt{T}), (\mathtt{A_2}, \mathtt{R})>], \{<\mathtt{A_1} : \mathtt{t}, \mathtt{A_2} : \mathtt{r}> \mid \mathtt{t} \in \mathtt{O_t} \wedge \mathtt{r} \in \mathtt{O_r} \wedge \mathtt{pred}(\mathtt{t}, \mathtt{r})\}) \end{aligned}$$

La définition de cet opérateur est modifiée quand il prend un objet de type TUPLE en paramètre. En effet, il devient nécessaire de mettre à plat les tuples imbriqués résultant de manière à préserver la composition de cet opérateur.

En plus de ces opérateurs principaux, l'algèbre *Encore* inclut des opérateurs ensemblistes (Union, Difference, et Intersection) et des opérateurs sur les collections (Flatten, DupEliminate, Nest et UnNest). Leur définition est celle de la théorie des ensembles et de la manipulation des collections. Tous ces opérateurs définissent une algèbre permettant d'interroger une BDOO. Dans ce qui suit, nous montrons comment les définitions de ces opérateurs peuvent être réutilisées et étendues pour interroger les données, les ontologies et à la fois les ontologies et les données d'une BDBO.

3.2 Algèbre permettant d'interroger les données d'une BDBO

Précédemment, nous avons adapté le modèle de données *Encore* à l'approche relationnelle-objet suivie par OntoQL. Nous devons définir les opérateurs de cette algèbre par rapport à cette approche.

3.2.1 Les opérateurs de *OntoAlgebra* et leur signature

Dans l'approche relationnelle-objet, les paramètres d'entrée et de sortie des requêtes sont des relations et non des collections d'objets comme dans l'algèbre *Encore*. Cependant, les relations peuvent être représentées dans le modèle *Encore* par une collection de tuples (SET[TUPLE]). Pour suivre l'approche relationnelle-objet, il suffit donc de spécialiser les opérateurs de cette algèbre à la signature SET[TUPLE] $\times \cdots \rightarrow$ SET[TUPLE]. Ce n'est cependant pas possible pour les deux opérateurs suivants :

– Image : SET[T] $\times$ Function $\rightarrow$ SET[Q] car il retourne une collection d'objets dont le type est le codomaine de la fonction appliquée. Il ne peut donc pas retourner une relation. Cet opérateur peut cependant être remplacé par l'appel de l'opérateur Project avec une seule fonction en paramètre puisque cet opérateur est une généralisation de l'opérateur Image ;

– Flatten : SET[SET[T]] $\rightarrow$ SET[T] car il prend en paramètre une collection de collections qu'il met à plat. Cet opérateur n'est pas nécessaire pour OntoQL car la mise à plat d'une relation peut être réalisée avec l'opérateur UnNest (cf. section 3.2.2).

Mis à part ces deux opérateurs, l'algèbre *OntoAlgebra* est ainsi constituée de l'ensemble des opérateurs *Encore* spécialisés à la signature SET[TUPLE] $\times \cdots \rightarrow$ SET[TUPLE]. Nous notons OntoProject, OntoSelect, OntoOJoin, ... ces opérateurs. Suivant notre formalisation de la section 2.2, la signature d'un opérateur pour interroger les données d'une BDBO est :

$$\mathtt{TUPLE}[<(\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C})>] \times 2^{\mathtt{V_C}} \times \cdots \rightarrow \mathtt{TUPLE}[<(\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C})>] \times 2^{\mathtt{V_C}}$$

Cette signature présente l'inconvénient de ne pas préciser le type de la collection en entrée et en sortie. Comme en SQL, tous les opérateurs sont définis sur des collections autorisant les doublons (MULTISET) à l'exception des opérateurs ensemblistes (OntoUnion, OntoIntersect et OntoExcept) définis sur des ensembles sans doublon (SET).

Cette signature est adaptée aux requêtes SQL où la relation en entrée est désignée par le nom d'une table ou d'une table typée. Cependant, dans les requêtes OntoQL, cette relation n'est pas explicite. En effet, l'identifiant d'une classe peut être spécifié dans la clause FROM pour désigner la relation qui peut être construite en projetant cette classe sur ses propriétés applicables. Cette opération, qui peut être réalisée avec l'opérateur OntoProject, doit être représentée explicitement dans l'expression algébrique d'une requête OntoQL. Nous avons donc défini la signature de l'opérateur OntoProject pour que son paramètre d'entrée puisse être une classe et non une relation. Sa signature est ainsi :

$$\mathtt{ADT_C} \times 2^{\mathtt{V_C}} \times 2^{\mathtt{String} \times \mathtt{Function}} \rightarrow \mathtt{TUPLE}[< (\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C}) >] \times 2^{\mathtt{V_C}}$$

Notons que ce changement de signature n'a pas de conséquence sur le fait que les opérateurs composent puisque les types TUPLE sont inclus dans le type $\mathtt{ADT_C}$. En effet, puisque le retour d'un opérateur est de type TUPLE, il peut être pris en entrée d'un opérateur dont le domaine est $\mathtt{ADT_C}$.

Une autre particularité de l'approche relationnelle-objet qui n'est pas prise en compte par l'algèbre *Encore* est de permettre l'expression de requêtes non polymorphes (ONLY). En effet, dans l'algèbre *Encore*, une classe C référence les instances de C et les instances de toutes les sous-classes de C. Cette algèbre ne permet pas de référencer que les instances de C uniquement. C'est pourquoi nous avons introduit la fonction $\mathtt{ext} : \mathtt{Class} \rightarrow 2^{\mathtt{I}}$ qui retourne les instances d'une classe et $\mathtt{ext}^* : \mathtt{Class} \rightarrow 2^{\mathtt{I}}$ qui retourne les instances directes d'une classe ainsi que celles de toutes ses sous-classes directes et indirectes.

Si c est une classe et $\mathtt{c_1}, \ldots, \mathtt{c_n}$ sont les sous-classes directes de c, $\mathtt{ext}^*$ est dérivées récursivement[28] à partir de ext de la manière suivante sur le modèle de données d'une BDBO :

$$\mathtt{ext(c)} = \mathtt{typeI}^{-1}(\mathtt{nomination(c)})$$
$$\mathtt{ext}^*(\mathtt{c}) = \mathtt{ext(c)} \cup \mathtt{ext}^*(\mathtt{c_1}) \cup \cdots \cup \mathtt{ext}^*(\mathtt{c_n})$$

Ces fonctions peuvent être utilisées avec l'opérateur OntoProject pour construire la relation associée à une classe contenant ses instances directes ou également indirectes.

Enfin, l'algèbre *Encore* ne couvre pas l'ensemble des opérateurs relationnels-objets disponibles dans OntoQL. Nous avons donc introduit les opérateurs suivants (leur signature est précisée dans l'annexe C) :

- l'opérateur OntoSort qui permet de trier une relation ;
- les opérateurs OntoLeftOuterOJoin, OntoRightOuterOJoin et OntoFullOuterOJoin qui permettent de réaliser des jointures externes.

Les opérateurs de l'algèbre *OntoAlgebra* étant précisés, nous présentons maintenant leur définition formelle.

3.2.2 Définition des principaux opérateurs de *OntoAlgebra*

La sémantique associée aux opérateurs de l'algèbre *Encore* ne prend pas en compte les particularités du modèle de données d'une BDBO. Nous avons donc introduit de nouvelles définitions pour ces opérateurs. Dans ces définitions, nous utilisons des notations similaires à celles utilisées précédemment (T, $\mathtt{O_t}$, etc.).

[28] Pour simplifier les notations, nous étendons toutes les fonctions f par $\mathtt{f}(\emptyset) = \emptyset$.

OntoProject. La signature de l'opérateur `OntoProject` est celle de `Project` adaptée au modèle d'une BDBO (cf. section précédente) :

$$\mathtt{ADT_C} \times 2^{\mathtt{V_C}} \times 2^{\mathtt{String}\times\mathtt{Function}} \rightarrow \mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>] \times 2^{\mathtt{V_C}}$$

A l'opposé du modèle de données d'une BDOO, une des propriétés qui intervient dans l'une des fonctions prises en paramètre peut ne pas être présente dans l'extension d'une classe. Ainsi, nous ne pouvons pas utiliser la fonction `valI` pour définir la sémantique de cet opérateur. Il est nécessaire d'étendre son domaine aux propriétés définies sur une classe mais qui ne sont pas utilisées dans son extension. Ceci nécessite d'introduire la valeur `NULL`. Nous appelons `OntoValI` l'extension de `val`. Elle est définie par :

$$\begin{aligned}\mathtt{OntoValI}(\mathtt{i},\mathtt{p}) &= \mathtt{valI}(\mathtt{i},\mathtt{p}), \text{ si } \mathtt{p} \in \mathtt{usedProperties}(\mathtt{typeI}(\mathtt{i})) \\ &= \mathtt{NULL} \text{ sinon}\end{aligned}$$

Pour préserver la composition de la fonction `OntoValI`, elle retourne `NULL` lorsqu'elle est appliquée à cette même valeur (`OntoValI(NULL, p) = NULL`). En utilisant l'opérateur `OntoValI`, `OntoProject` est ainsi défini par :

$$\begin{aligned}&\mathtt{OntoProject}(\mathtt{T},\mathtt{I_t},\{(\mathtt{A_1},\mathtt{f_1}),\ldots,(\mathtt{A_n},\mathtt{f_n})\}) = \\ &\quad (\mathtt{TUPLE}[<(\mathtt{A_1},\mathtt{range}(\mathtt{f_1})),\ldots,(\mathtt{A_n},\mathtt{range}(\mathtt{f_n}))>], \\ &\quad\quad \{<\mathtt{A_1}:\mathtt{OntoValI}(\mathtt{i},\mathtt{f_1}),\ldots,\mathtt{A_n}:\mathtt{OntoValI}(\mathtt{i},\mathtt{f_n})> \mid \mathtt{i} \in \mathtt{I_t}\})\end{aligned}$$

Exemple. Construire la relation correspondant à l'expression OntoQL : `FROM Item`. Cette relation devra également contenir les tuples correspondant aux instances de la classe `Post`.

$$\begin{aligned}\mathtt{RItem}{*} := \mathtt{OntoProject}(\mathtt{Item},\mathtt{ext}^*(\mathtt{Item}),\lambda\mathtt{i}\bullet\{&(\mathtt{oid},\mathtt{i.oid}),(\mathtt{name},\mathtt{i.name}),\\ &(\mathtt{has_container},\mathtt{i.has_container}),\\ &(\mathtt{has_reply},\mathtt{i.has_reply})\})\end{aligned}$$

Explication. La relation `RItem*` associée à la classe `Item` est construite en utilisant l'opérateur `Onto-Project`. Grâce à l'utilisation de la fonction `ext*`, il est appliqué sur les instances directes et indirectes de la classe `Item`. Un itérateur `i` est introduit sur ces instances en utilisant la notation lambda (λ). L'opérateur `OntoProject` projette les instances `i` de la classe `Item` sur l'ensemble des propriétés applicables de cette classe. La relation construite est de type `SET[TUPLE<(oid:Oid), (name:String),(has_container, REF[Container]), (has_reply, REF[Item])>]`.

Remarque. Pour simplifier les exemples suivants, nous n'indiquons pas la construction des relations associées aux classes manipulées dans les expressions algébriques des requêtes. Nous utilisons directement le nom de la classe préfixé par `R` pour référencer cette relation. Nous suffixons ce nom par * lorsqu'elle contient également les tuples correspondant aux instances de ses sous-classes.

OntoSelect. La signature de l'opérateur `OntoSelect` est :

$$\begin{aligned}&\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>] \times 2^{\mathtt{V_C}} \times \mathtt{Predicate} \\ &\quad\rightarrow \mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>] \times 2^{\mathtt{V_C}}\end{aligned}$$

Sa sémantique est similaire à celle de `Select`:

$$\mathtt{OntoSelect}(\mathtt{T},\mathtt{I_t},\mathtt{pred}) = (\mathtt{T},\{\mathtt{i} \mid \mathtt{i} \in \mathtt{I_t} \land \mathtt{pred}(\mathtt{i})\})$$

Si le prédicat pris en paramètre de `OntoSelect` contient des appels de fonctions, alors la fonction `OntoValI` est utilisée. En conséquence, la valeur `NULL` peut apparaître dans un prédicat. Les opérations impliquant cette valeur ont la même sémantique que celle définie en SQL.

Exemple. Rechercher les messages créés par un utilisateur dont l'email se termine par `'@ensma.fr'`.

```
SELECT DEREF(p.oid) AS post FROM Post AS p
 WHERE p.has_creator.email LIKE '%@ensma.fr'
```

$\texttt{PostEnsma:= OntoSelect(RPost*}, \lambda \texttt{p}_\bullet \texttt{like(DEREF(p.has_creator).email, '\%@ensma.fr'))} ;$

$\texttt{Result:= OntoProject(PostEnsma}, \lambda \texttt{p}_\bullet \{(\texttt{post}, \texttt{DEREF(p.oid)})\})$

Explication. L'opérateur `OntoSelect` est utilisé sur la relation construite à partir de la classe `Post` qui contient des tuples correspondant à ces instances et à celles de ses sous-classes (`RPost*`). Un itérateur `p` est introduit sur les tuples de cette relation en utilisant la notation lambda (λ). Seuls les tuples satisfaisant le prédicat `p.has_creator.email LIKE '%@ensma.fr'` sont retournés dans la relation `PostEnsma`. Pour représenter ce prédicat dans *OntoAlgebra*, l'opérateur `LIKE` est défini comme une fonction (`like`) et l'expression de chemin est représentée en appliquant la fonction `DEREF`. L'opérateur `OntoProject` applique ensuite la fonction `DEREF` sur l'identifiant des tuples de cette relation. `Result` est donc une relation ne possédant qu'un seul attribut nommé `post` dont le type est la classe `Post`. Le type de cette relation est donc `MULTISET[TUPLE[<(post:Post)>]]`.

OntoOJoin. La signature de l'opérateur `OntoOJoin` est :

$$\texttt{TUPLE}[<(\texttt{String}, \texttt{ADT}_\texttt{C}), \ldots, (\texttt{String}, \texttt{ADT}_\texttt{C})>] \times 2^{\texttt{V}_\texttt{C}} \times \texttt{TUPLE}[<(\texttt{String}, \texttt{ADT}_\texttt{C}), \ldots, (\texttt{String}, \texttt{ADT}_\texttt{C})>] \times 2^{\texttt{V}_\texttt{C}} \times \texttt{Predicate} \rightarrow \texttt{TUPLE}[<(\texttt{String}, \texttt{ADT}_\texttt{C}), \ldots, (\texttt{String}, \texttt{ADT}_\texttt{C})>] \times 2^{\texttt{V}_\texttt{C}}$$

Sa sémantique est celle de l'opérateur `OJoin` lorsqu'il prend un type `TUPLE` en paramètre :

$$\texttt{OntoOJoin}(\texttt{TUPLE}[<(\texttt{A}_1, \texttt{T}_1), \ldots, (\texttt{A}_\texttt{m}, \texttt{T}_\texttt{m})>], \texttt{I}_\texttt{t}, \texttt{TUPLE}[<(\texttt{A}_{\texttt{m}+1}, \texttt{T}_{\texttt{m}+1}), \ldots, (\texttt{A}_\texttt{n}, \texttt{T}_\texttt{n})], \texttt{I}_\texttt{r}, \texttt{pred}) = (\texttt{TUPLE}[<(\texttt{A}_1, \texttt{T}_1), \ldots, (\texttt{A}_\texttt{n}, \texttt{T}_\texttt{n})>], \{<(\texttt{A}_1, \texttt{t.A}_1), \ldots, (\texttt{A}_\texttt{m}, \texttt{t.A}_\texttt{m}), (\texttt{A}_{\texttt{m}+1}, \texttt{r.A}_{\texttt{m}+1}), \ldots, (\texttt{A}_\texttt{n}, \texttt{r.A}_\texttt{n})> \mid \texttt{t} \in \texttt{I}_\texttt{t} \wedge \texttt{r} \in \texttt{I}_\texttt{r} \wedge \texttt{pred(t, r)}\})$$

Cet opérateur permet ainsi de joindre deux relations. Lorsqu'elles ont des attributs ayant le même nom, la relation qui résulte de cette jointure comporte plusieurs attributs portant le même nom. Pour les distinguer, l'opérateur `OntoOJoin` prend en paramètres optionnels deux chaînes de caractères qui préfixent les attributs des relations jointes. La jointure est réalisée selon le prédicat pris en paramètre. Il est traité de la même manière que pour `OntoSelect`.

Exemple. Rechercher les messages dont le titre contient le nom d'un des utilisateurs.

```
SELECT p.title, p.content FROM Post AS p, User AS u
 WHERE p.title LIKE '%' || u.name || '%'
```

$\texttt{PostName:= OntoOJoin(RPost*, RUser*}, \lambda \texttt{p} \, \lambda \texttt{u}_\bullet \texttt{like(p.title, '\%'} \,\|\, \texttt{u.name} \,\|\, \texttt{'\%'))} ;$

$\texttt{Result:= OntoProject(PostName}, \lambda \texttt{p}_\bullet \{(\texttt{title}, \texttt{p.title}), (\texttt{content}, \texttt{p.content})\})$

Explication. L'opérateur `OntoOJoin` est utilisé pour joindre les relations associées aux classes `Post` et `User`. Les itérateurs `p` et `u` sont introduits sur les tuples de ces deux relations. La jointure de ces deux relations se fait sur le prédicat `WHERE p.title LIKE '%' || u.name || '%'`. Ce prédicat utilise l'opérateur de concaténation (`||`) pour tester si le titre du message contient (encadré par le caractère `%`) le nom de l'utilisateur. Le résultat de cette jointure est la relation `PostName`. L'opérateur `OntoProject` permet ensuite de retourner le titre et le contenu des messages contenus dans la relation `PostName`. `Result` est donc du type `MULTISET[TUPLE[<(title:String), (content:String)>]]`.

OntoNest et OntoUnNest. Les opérateurs `OntoNest` et `OntoUnNest` permettent de représenter un ensemble de tuples comme une relation imbriquée ou mise à plat. La signature de l'opérateur `OntoNest` est :

$$\mathtt{TUPLE}[<(\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C})>] \times 2^{\mathtt{V_C}} \times \mathtt{String} \rightarrow \mathtt{TUPLE}[<(\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C})>] \times 2^{\mathtt{V_C}}$$

Il est défini par :

$$\begin{aligned}&\mathtt{OntoNest}(\mathtt{TUPLE}[<(\mathtt{A_1}, \mathtt{T_1}), \ldots, (\mathtt{A_i}, \mathtt{T_i}), \ldots, (\mathtt{A_n}, \mathtt{T_n})>], \mathtt{I_t}, \mathtt{A_i}) = \\ &\quad (\mathtt{TUPLE}[<(\mathtt{A_1}, \mathtt{T_1}), \ldots, (\mathtt{A_i}, \mathtt{SET}[\mathtt{T_i}]), \ldots, (\mathtt{A_n}, \mathtt{T_n})>], \\ &\quad \{<\mathtt{A_1} : \mathtt{s.A_1}, \ldots, \mathtt{A_i} : \mathtt{t}, \ldots, \mathtt{A_n} : \mathtt{s.A_n}> \mid \forall \mathtt{r} \in \mathtt{t}\ \exists \mathtt{s} \in \mathtt{I_t}.(\mathtt{s.A_i} = \mathtt{r})\})\end{aligned}$$

Cet opérateur remplace la relation d'entrée par un ensemble de tuples similaires pour les attributs autres que $\mathtt{A_i}$ mais groupant dans une collection les valeurs de $\mathtt{A_i}$ correspondant aux tuples identiques pour les autres attributs. Cet opérateur est également défini pour grouper une relation selon plusieurs attributs $\mathtt{A_i}$. Ceci permet de représenter l'opérateur de groupement (`GROUP BY`) de SQL.

Exemple. Retourner le nombre d'objets (messages, etc.) écrits sur chaque forum.

```
SELECT count(i.oid) AS nbItem FROM Item AS i GROUP BY i.has_container
```

$$\begin{aligned}&\mathtt{PostNest} := \mathtt{OntoNest}(\mathtt{RItem}*, \{\mathtt{oid}, \mathtt{name}, \mathtt{has_reply}\}) ; \\ &\mathtt{Result} := \mathtt{OntoProject}(\mathtt{PostNest}, \lambda \mathtt{i}_{\bullet}\{(\mathtt{nbItem}, \mathtt{COUNT}(\mathtt{i.oid}))\})\end{aligned}$$

Explication. L'opérateur `GROUP BY` permet de créer un groupe pour chaque valeur différente de la propriété `has_container`. Pour représenter cette opération avec l'opérateur `OntoNest`, il est nécessaire de l'appliquer à toutes les autres propriétés de la classe `Item`, c'est-à-dire `oid`, `name` et `has_reply`. Le résultat de cette opération (`PostNest`) est une relation de type `MULTISET[TUPLE[<(oid:SET[Oid]),(name:SET[String]), (has_reply:SET[REF[Item]]), (has_container:REF[Container])>]]`. L'opérateur `OntoProject` applique ensuite la fonction `COUNT` pour retourner le nombre d'éléments des collections contenues dans la colonne `oid`. `Result` est donc du type `MULTISET[TUPLE[<(count:Int)>]]`.

L'opérateur `OntoUnNest` a l'effet inverse de l'opérateur `OntoNest`. Sa signature est :

$$\mathtt{TUPLE}[<(\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C})>] \times 2^{\mathtt{V_C}} \times \mathtt{String} \times \mathtt{String} \rightarrow \mathtt{TUPLE}[<(\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C})>] \times 2^{\mathtt{V_C}}$$

Il est défini par :

$$\begin{aligned}\mathtt{OntoUnNest}(\mathtt{TUPLE}[<(\mathtt{A}_1,\mathtt{T}_1),\ldots,(\mathtt{A}_i,\mathtt{SET}[\mathtt{T}_i]),\ldots,(\mathtt{A}_n,\mathtt{T}_n)>],\mathtt{I}_t,\mathtt{A}_i,\mathtt{A}) = \\ (\mathtt{TUPLE}[<(\mathtt{A}_1,\mathtt{T}_1),\ldots,(\mathtt{A}_n,\mathtt{T}_n),(\mathtt{A},\mathtt{T}_i)>], \\ \{<\mathtt{A}_1:\mathtt{s.A}_1,\ldots,\mathtt{A}_n:\mathtt{s.A}_n,\mathtt{A}:\mathtt{t}>\mid \mathtt{s}\in\mathtt{I}_t\wedge \mathtt{t}\in\mathtt{s.A}_i\})\end{aligned}$$

Cet opérateur définit notamment la sémantique de l'opérateur OntoQL UNNEST.

Exemple. Retourner le nom des forums auxquels chaque utilisateur est abonné.

```
SELECT u.name AS uname, f.name AS fname
  FROM User AS u, UNNEST(u.subscriber_of) AS f
```

$$\begin{aligned}\mathtt{PostUnNest} &:= \mathtt{OntoUnNest}(\mathtt{RUser}*,\mathtt{subscriber_of},\mathtt{f});\\ \mathtt{Result} &:= \mathtt{OntoProject}(\mathtt{PostUnNest},\lambda \mathtt{u}_\bullet\{(\mathtt{uname},\mathtt{u.name}),\\ &\qquad (\mathtt{fname},\mathtt{DEREF(f).name})\})\end{aligned}$$

Explication. La relation RUser* contient l'attribut subscriber_of dont le type est SET[REF[Forum]]. L'opérateur OntoUnNest est utilisé pour dégrouper cette relation selon cet attribut. Le résultat (PostUnNest) est une relation contenant un attribut supplémentaire f dont le type est REF[Forum]. L'opérateur OntoProject applique l'opérateur DEREF aux identifiants correspondants à cet attribut pour retourner le nom des forums. Result est du type MULTISET[TUPLE[<(uname:String), (fname:String)>]].

Autres opérateurs. Nous ne détaillons pas les autres opérateurs de *OntoAlgebra* dont la sémantique se déduit des opérateurs définis dans cette section ou de ce ceux de l'algèbre relationnelle. Les définitions de l'ensemble des opérateurs de *OntoAlgebra* peuvent être consultées dans l'annexe C, section 1.

Nous avons défini la sémantique des différents opérateurs de *OntoAlgebra* et montré sur des exemples que des requêtes OntoQL peuvent être transformées en une expression algébrique composant ces opérateurs. Nous montrons maintenant comment toute requête OntoQL peut être transformée en une telle expression.

3.2.3 Expression algébrique d'une requête OntoQL

Le tableau 5.1 présente les principales règles de transformation[29] permettant de construire l'expression algébrique d'une requête OntoQL. Dans ce tableau, la deuxième colonne contient une expression OntoQL. Dans ces expressions algébriques, C désigne une classe, $\mathtt{p}_1,\ldots,\mathtt{p}_n$ sont des propriétés et $\mathtt{f}_1,\ldots,\mathtt{f}_n$ sont des fonctions. Une expression OntoQL peut contenir une autre expression OntoQL notée $\underline{\mathtt{exp}_i}$. Ainsi, une requête OntoQL peut être traduite à l'aide des correspondances exprimées dans ce tableau.

Exemple. Donner l'expression algébrique de la requête suivante :

```
SELECT p.title, p.content FROM Post AS p, User AS u
```

[29] La totalité des règles peut être consultée dans l'annexe C, section 2.

	Expressions OntoQL (exp)	**Expressions algébriques correspondantes**
1	C ou FROM C	OntoProject (C, ext*(C), $\{(p_1, p_1), \ldots, (p_n, p_n)\}$
2	ONLY(C) ou FROM ONLY(C)	OntoProject (C, ext(C), $\{(p_1, p_1), \ldots, (p_n, p_n)\}$
3	FROM $\underline{exp_1}$, $\underline{exp_2}$	OntoOJoin ($\underline{exp_1}$, $\underline{exp_2}$, true)
4	FROM $\underline{exp_1}$ INNER JOIN $\underline{exp_2}$ ON pred	OntoOJoin ($\underline{exp_1}$, $\underline{exp_2}$, pred)
5	FROM $\underline{exp}$ UNNEST $\underline{exp}.p_1$AS p	OntoUnNest ($\underline{exp}$, p_1, p)
6	SELECT $f_1, \ldots, f_n$ $\underline{exp}$	OntoProject ($\underline{exp}$, $\{(f_1, f_1), \ldots, (f_n, f_n)\}$)
7	DISTINCT $\underline{exp}$	OntoDupEliminate ($\underline{exp}$)
8	$\underline{exp}$ WHERE pred	OntoSelect ($\underline{exp}$, pred)
9	$\underline{exp}$ GROUP BY $p_1, \ldots, p_k$	OntoNest ($\underline{exp}$, $\{p_{k+1}, \ldots, p_n\}$)
10	$\underline{exp}$ HAVING pred	OntoSelect ($\underline{exp}$, pred)
11	$\underline{exp}$ ORDER BY $f_1, \ldots, f_n$	OntoSort ($\underline{exp}$, $\{f_1, \ldots, f_n\}$, true)

TAB. 5.1 – Extrait des règles permettant de construire l'expression algébrique d'une requête OntoQL

```
WHERE p.title LIKE '%' || u.name || '%'
```

La figure 5.2 (a) présente l'arbre syntaxique de cette requête et la figure 5.2 (b) représente l'expression algébrique de cette requête. Le tableau suivant indique comment cette expression algébrique a été construite à partir des règles du tableau 5.1 :

exp_i	$Q(exp_i)$	$A(exp_i)$
exp_1	Post AS p	OntoProject (Post, ext*(Post), {(oid, oid), ..., (email, email) }
exp_2	User AS u	OntoProject (User, ext*(User), {(oid, oid), ..., (note, note) }
exp_3	FROM $Q(exp_1)$, $Q(exp_2)$	OntoOJoin ($A(exp_1)$, $A(exp_2)$, true)
exp_4	$Q(exp_3)$ WHERE p.title LIKE '%' || u.name || '%'	OntoSelect ($A(exp_3)$, like(p.title, '%' || u.name || '%'))
exp_5	SELECT p.title, p.content $Q(exp_4)$	OntoProject ($A(exp_4)$, {(title, title), (content, content) }

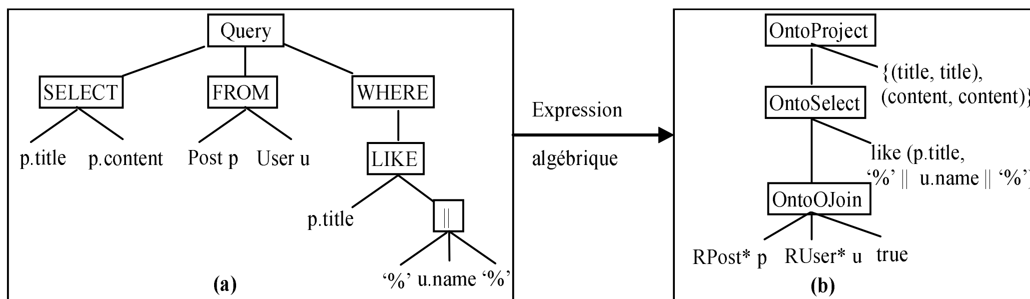


FIG. 5.2 – Exemple de génération de l'expression algébrique *OntoAlgebra* d'une requête OntoQL

Explication. Les deux premières lignes du tableau appliquent la règle 1 du tableau 5.1 afin de construire la relation associée aux classes `Post` et `User`. Ces deux lignes correspondent aux expressions `RPost*` et `RUser*` utilisées précédemment. La règle 3 est ensuite appliquée pour obtenir le résultat du produit cartésien (prédicat toujours vrai) entre ces deux relations. La clause `WHERE` de la requête est ensuite prise en compte par la règle 8. Elle permet de filtrer les tuples obtenus en appliquant le prédicat `p.title LIKE '%' || u.name || '%'` à chacun de ces tuples. Enfin l'expression finale ($\mathtt{exp}_5$) est construite en appliquant la règle 6. Cette règle indique d'utiliser l'opérateur `OntoProject` pour projeter les propriétés `title` et `content`. Notons que l'expression algébrique finale ($\mathtt{exp}_5$) est différente de celle présentée dans la section 3.2.2 pour la même requête. Cependant, ces deux expressions sont équivalentes car plusieurs règles d'équivalence peuvent être définies entre les opérateurs de *OntoAlgebra* (cf. section 4.2).

Dans les règles définies, des fonctions sont utilisées pour représenter toute expression qui peut apparaître dans la clause `SELECT` d'une requête OntoQL. En conséquence, il existe dans l'algèbre *OntoAlgebra* une fonction correspondant à chaque opérateur qui n'est pas représenté explicitement dans ces règles. C'est le cas par exemple des opérateurs `IS OF` ou `CASE`. Nous verrons dans la section suivante un exemple utilisant ces fonctions.

Les opérateurs présentés dans cette section peuvent ainsi être utilisés pour représenter l'expression algébrique d'une requête OntoQL portant sur les données d'une BDBO. La section suivante présente les opérateurs permettant d'interroger les ontologies d'une BDBO.

3.3 Algèbre permettant d'interroger les ontologies d'une BDBO

Le langage OntoQL permet d'interroger les ontologies d'une BDBO avec les mêmes opérateurs que ceux permettant d'interroger les données d'une BDBO. En conséquence, les opérateurs de *OntoAlgebra* sont également définis pour permettre d'interroger les ontologies. La signature d'un opérateur défini sur le modèle de données du niveau ontologie, défini en section 2.3, est :

$$\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_E}),\ldots,(\mathtt{String},\mathtt{ADT_E})>]\times 2^{\mathtt{V_E}}\times\cdots\rightarrow \mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_E}),\ldots,(\mathtt{String},\mathtt{ADT_E})>]\times 2^{\mathtt{V_E}}$$

Les paramètres d'entrée et de sortie de ces opérateurs sont des relations (`TUPLE`) contenant des tuples dont les valeurs sont des éléments des ontologies et/ou des valeurs d'attributs ($\mathtt{V_E}$). A ce niveau, le modèle de données ne présente pas de particularité par rapport au modèle *Encore*. En conséquence, la sémantique des opérateurs de l'algèbre *Encore* est adaptée aux opérateurs de *OntoAlgebra* permettant d'interroger les ontologies.

Pour illustrer ces opérateurs de *OntoAlgebra*, nous présentons des exemples illustrant comment une requête OntoQL sur les ontologies peut être décomposée en séquence d'appels aux opérateurs de cette algèbre. Cette décomposition peut être réalisée avec des règles similaires à celles présentées pour l'algèbre sur les données. Dans ces exemples, nous n'indiquons pas l'expression algébrique permettant de construire la relation en entrée à partir de l'identifiant d'une entité. Comme pour les données, cette opération consiste à projeter une entité sur ses attributs via l'opérateur `OntoProject`. Cette relation peut ne contenir que les tuples correspondant aux instances de cette entité en utilisant la fonction $\mathtt{ext} : \mathtt{E} \rightarrow 2^{\mathtt{OC}}$ ou également celles de ces sous-entités en utilisant la fonction $\mathtt{ext*} : \mathtt{E} \rightarrow 2^{\mathtt{OC}}$ dérivée à partir de `ext`.

Exemple. Retourner les super-classes de la classe Post.

```
SELECT c.#superClasses FROM #Class AS c WHERE c.#code = 'Post'
```

$$\texttt{ClassPost} := \texttt{OntoSelect}(\texttt{R\#Class}*, \lambda \texttt{c}_\bullet \texttt{c.\#code} = \texttt{'Post'});$$
$$\texttt{Result} := \texttt{OntoProject}(\texttt{ClassPost}, \lambda \texttt{c}_\bullet \{(\texttt{superClasses}, \texttt{c.\#superClasses})\})$$

Explication. R#Class* représente la relation construite en projetant l'entité #Class sur l'ensemble de ses attributs (#oid, #code, etc.). L'opérateur OntoSelect est appliqué sur cette relation pour ne conserver que le tuple correspondant à la classe dont l'identifiant (#code) est Post. Ensuite, l'opérateur OntoProject projette cette relation sur l'attribut #superClasses. Le type de cet attribut étant SET[REF[#Class]], Result est de type MULTISET[TUPLE[<(#superClasses, SET[REF[#Class]])>]].

Cet exemple a montré une utilisation de l'opérateur OntoSelect. L'exemple suivant montre une utilisation de l'opérateur OntoOJoin.

Exemple. Retourner les propriétés avec leur domaine.

```
SELECT DEREF(p.#oid) AS prop, DEREF(c.#oid) AS class
  FROM #Propery AS p, #Class AS c
 WHERE p.#scope = c.#oid
```

$$\texttt{JoinCP} := \texttt{OntoOJoin}(\texttt{R\#Property}*, \texttt{R\#Class}*, \texttt{P}, \texttt{C}, \lambda \texttt{p}\, \lambda \texttt{c}_\bullet \texttt{p.\#scope} = \texttt{c.\#oid});$$
$$\texttt{Result} := \texttt{OntoProject}(\texttt{JoinCP}, \lambda \texttt{cp}_\bullet \{(\texttt{prop}, \texttt{DEREF}(\texttt{cp.P\#oid})), (\texttt{class}, \texttt{DEREF}(\texttt{cp.C\#oid}))\})$$

Explication. L'opérateur OntoOJoin est utilisé pour joindre les relations contenant les propriétés et les classes des ontologies. La condition de jointure utilise l'attribut #scope pour joindre les propriétés avec leur domaine. La relation JoinCP contient ainsi les attributs applicables sur les entités #Property et #Class. Ces entités ayant des attributs en commun (par exemple, #code), les préfixes P et C, paramètres optionnels de l'opérateur OntoOJoin, sont utilisés pour les distinguer. L'opérateur OntoProject est ensuite utilisé pour appliquer la fonction DEREF afin de retourner des tuples contenant une instance de l'entité #Property et une instance de l'entité #Class. Result est donc de type SET[TUPLE[<(prop, #Property)>, (class, #Class)>]].

L'exemple suivant montre l'expression algébrique d'une requête utilisant les opérateurs CASE et IS OF de OntoQL. Ces opérateurs sont des fonctions qui peuvent s'appliquer sur les valeurs des instances ou des éléments d'une ontologie, c'est-à-dire sur l'ensemble V défini comme étant l'union de ces deux types de valeurs ($V = V_C \cup V_E$). Elles sont définies par :

- case : $\texttt{Boolean} \times V \times V \rightarrow V$. $\texttt{case}(b, v_1, v_2)$ retourne v_1 si b vaut vrai. Sinon elle retourne v_2. Cette fonction est surchargée pour permettre de réaliser plusieurs tests ;
- isOf : $V \times \texttt{ADT} \rightarrow \texttt{Boolean}$. isOf(v, T) retourne la valeur vrai si v est de type T. Cette opération est définie pour les types références, classes et entités. Elle est surchargée pour permettre de tester l'appartenance d'une valeur à plusieurs ADT.

Exemple. Retourner les propriétés avec un libellé de leur codomaine.

```
SELECT p.#name[FR] AS name,
       CASE WHEN p.#range IS OF (REF(#StringType)) THEN 'String'
            WHEN p.#range IS OF (REF(#IntType)) THEN 'Int'
            ELSE 'Unknown' END AS datatype
 FROM #Property AS p
```

$$\begin{aligned}\texttt{Result} := \texttt{OntoProject}(&\texttt{R\#Property*},\\ &\lambda \texttt{p}_\bullet\{(\texttt{\#name[FR]}, \texttt{p.\#name}),\\ &\quad(\texttt{datatype}, \texttt{case}(\texttt{isOf}(\texttt{p.\#range}, \texttt{REF(\#StringType)}), \texttt{'String'},\\ &\qquad\qquad \texttt{isOf}(\texttt{p.\#range}, \texttt{REF(\#IntType)}), \texttt{'Int'}, \texttt{'Unknown'}))\})\end{aligned}$$

Explication. L'opérateur `OntoProject` est utilisé pour retourner le nom en français des propriétés et un libellé de son type de données (`SET[TUPLE[<(name, String)>, (datatype, String)>]]`). Le nom en français est retourné en appelant la fonction qui permet de retrouver la valeur d'une chaîne multilingue pour une langue naturelle donnée (notation abrégée `[]`) sur l'attribut `#name`. Le libellé dépend du type de la propriété. La fonction `isOf` est utilisée pour faire des tests sur ce type. Elle est utilisée conjointement à la fonction `case` afin de retourner un libellé en fonction de ce type. La fonction `case` utilisée dans cet exemple permet de faire deux tests. Sa signature est ainsi `case` : $\texttt{Boolean} \times \texttt{V} \times \texttt{Boolean} \times \texttt{V} \times \texttt{V} \rightarrow \texttt{V}$.

3.4 Algèbre permettant d'interroger les ontologies et les données d'une BDBO

Afin de permettre d'interroger à la fois les ontologies et les données, les opérateurs de *OntoAlgebra* doivent pouvoir prendre en entrée et retourner en sortie ces deux types d'éléments. En conséquence, la signature de ces opérateurs est étendue à :

$$\texttt{TUPLE}[< (\texttt{String}, \texttt{ADT}), \ldots, (\texttt{String}, \texttt{ADT}) >] \times 2^{\texttt{V}} \times \cdots \rightarrow \texttt{TUPLE}[< (\texttt{String}, \texttt{ADT}), \ldots, (\texttt{String}, \texttt{ADT}) >] \times 2^{\texttt{V}}$$

Cette signature permet aux relations prises en entrée et en sortie des opérateurs de contenir des éléments correspondant aux données à base ontologique et aux ontologies en utilisant les ensembles `V` et `ADT` qui regroupent ces valeurs et ces types de données (cf. section 2.4). Dans la section suivante nous montrons que les opérateurs de *OntoAlgebra* basés sur cette signature permettent d'interroger conjointement les ontologies et les données dans le sens ontologie vers données ou inversement.

3.4.1 Ontologie vers données

Pour permettre de rechercher des classes ainsi que leurs instances, OntoQL permet d'introduire des itérateurs dynamiques. Ce mécanisme peut être représenté avec l'opérateur `OntoOJoin` de *OntoAlgebra* en effectuant la jointure entre les classes (`#Class`) et l'ensemble des instances de ces classes (instances de `ObjectC`). La condition de jointure utilise la fonction `ext` ou `ext*` afin d'obtenir les classes avec leurs instances directes ou indirectes.

Exemple. Rechercher les classes avec l'identifiant de leurs instances directes.

```
SELECT c.#name, i.oid FROM #Class AS c, ONLY(c) AS i
```

$$\texttt{ClassInst} := \texttt{OntoOJoin}(\texttt{R\#Class}*, \texttt{RObjectC}*, \lambda \texttt{c}\, \lambda \texttt{i}_{\bullet} \texttt{i} \in \texttt{ext}(\texttt{c}))\ ;$$
$$\texttt{Result} := \texttt{OntoProject}(\texttt{ClassInst}, \lambda \texttt{ci}_{\bullet} \{(\texttt{\#name}, \texttt{ci.\#name}), (\texttt{oid}, \texttt{ci.oid})\})$$

Explication. L'opérateur OntoOJoin est utilisé pour effectuer la jointure entre la relation contenant l'ensemble des classes (R#Class∗) et celle contenant toutes les instances, c'est-à-dire toutes celles de la classe racine (RObjectC∗). La condition de jointure utilise la fonction ext afin de ne joindre une classe qu'avec ses instances directes (ONLY). Le résultat de cette opération est la relation ClassInst qui contient l'ensemble des attributs de la relation R#Class∗ (#oid, #code, etc.) ainsi que l'attribut oid de la relation (RObjectC∗). L'opérateur OntoProject projette cette relation sur la propriété #name et oid. Result est ainsi du type MULTISET[TUPLE[<(#name:String),(oid:Oid)>]].

Afin de retrouver la description des instances des classes, OntoQL permet de réaliser une projection qui utilise une propriété déterminée à l'exécution de cette requête. L'opérateur de projection de *OntoAlgebra* étant OntoProject, nous avons étendu cet opérateur avec cette capacité. Ainsi cet opérateur peut prendre en paramètre une fonction résultat de l'appel d'une autre fonction. Une telle fonction peut ne pas être définie sur la relation sur laquelle OntoProject est appliquée.

Exemple. Rechercher les classes avec les valeurs de propriétés de leurs instances directes et indirectes.

```
SELECT c.#name AS cname, p.#name AS pname, i.p
  FROM #Class AS c, ONLY(c) AS i, UNNEST(c.#properties) AS p
```

$$\texttt{ClassInst} := \texttt{OntoOJoin}(\texttt{R\#Class}*, \texttt{RObjectC}*, \lambda \texttt{C}\, \lambda \texttt{i}_{\bullet} \texttt{i} \in \texttt{ext}(\texttt{C}))\ ;$$
$$\texttt{ClassPropInst} := \texttt{OntoUnNest}(\texttt{ClassInst}, \texttt{\#properties}, \texttt{p})\ ;$$
$$\texttt{Result} := \texttt{OntoProject}(\texttt{ClassPropInst}, \lambda \texttt{cip}_{\bullet} \{(\texttt{cname}, \texttt{cip.\#name}), (\texttt{pname}, \texttt{DEREF}(\texttt{cip.p}).\texttt{\#name}), (\texttt{p}, \texttt{cip.(cip.p)})\})$$

Explication. ClassInst est la relation de l'exemple précédent. Elle contient l'attribut #properties (ensemble des propriétés applicables sur une classe) dont le type est SET[REF[#Property]]. L'opérateur OntoUnNest est utilisé pour dégrouper cette relation selon cet attribut. Le résultat (ClassPropInst) est une relation ayant un attribut p de type REF[#Property]. L'opérateur OntoProject projette cette relation pour retrouver le nom des classes (#name) et le nom des propriétés applicables sur ces classes en utilisant la fonction DEREF. Cette classe est également projetée sur la propriété dont l'identifiant est une valeur de l'attribut p en utilisant l'expression cip.(cip.p). Le type de cette projection est l'union de l'ensemble des types de données des propriétés. Cette union de type comprend donc, en particulier, les types de données des propriétés de la classe Item. Le type de Result est ainsi MULTISET[TUPLE[<(cname:String),(pname:String), (p, UNION[Int,String, REF[Item], REF[Container], ...])>]].

3.4.2 Données vers les ontologies

Pour permettre de rechercher des instances d'une classe tout en obtenant sa description ontologique, OntoQL permet d'utiliser l'opérateur `TYPEOF`. Nous l'avons introduit dans le modèle de données d'une BDBO comme une fonction. Cette fonction peut être utilisée dans les différents opérateurs de *OntoAlgebra*.

Exemple. Rechercher les noms des instances de la classe `Item` en retournant également le nom de leur classe de base ainsi que le nom des propriétés utilisées pour les décrire.

```
SELECT i.name, p.#name
  FROM Item AS i, #Property AS p
  WHERE p.#oid = ANY (TYPEOF(i.oid).#usedProperties)
```

```
ItemProp:= OntoOJoin(RItem*, R#Property*,
                     λi λp•in(p.#oid, typeOf(i.oid).#usedProperties)) ;
Result:= OntoProject(ItemProp, λip•{(name, ip.name), (#name, ip.#name)})
```

Explication. L'opérateur `OntoOJoin` permet d'effectuer la jointure entre les instances de la classe `Item` et les propriétés des ontologies. La condition de jointure utilise la fonction `in` d'appartenance d'un élément à une collection. Ainsi, le prédicat en paramètre de l'opérateur `OntoOJoin` teste si l'identifiant de la propriété `p` appartient à la collection des identifiants des propriétés utilisées (`#usedProperties`) sur la classe de base de l'instance `i`. La classe de base de cette instance est retournée par la fonction `typeOf`.

Dans cette section, nous avons défini l'algèbre *OntoAlgebra* qui permet d'interroger les données, les ontologies et à la fois les ontologies et les données d'une BDBO. Nous avons montré que toute requête OntoQL peut être exprimée par une expression de cette algèbre. Ainsi, *OntoAlgebra* fournit une base formelle pour étudier le langage OntoQL.

4 Etude du langage OntoQL en utilisant l'algèbre *OntoAlgebra*

Différentes propriétés ont été définies pour caractériser un langage d'interrogation de bases de données. L'algèbre *OntoAlgebra* peut être utilisée pour établir que le langage OntoQL satisfait ces propriétés.

4.1 Propriétés du langage OntoQL

La *fermeture* et la *complétude relationnelle* sont les deux principales propriétés satisfaites par le langage SQL. Ces deux propriétés permettent d'assurer qu'un langage à un pouvoir d'expression minimum et qu'une requête de ce langage peut être construite comme composition d'autres requêtes. Ces propriétés ont non seulement été utilisées pour caractériser les langages définies pour le modèle relationnel mais aussi pour ceux définis pour d'autres modèles tels que le modèle XML [Deutsch et al., 1999] ou le modèle RDF [Haase et al., 2004]. Établir que le langage OntoQL satisfait ces deux propriétés permet ainsi de le caractériser selon un référentiel connu. Nous le montrons dans cette section en étudiant ces propriétés sur l'algèbre *OntoAlgebra*.

Une algèbre est dite *fermée*, si le résultat de tout opérateur de cette algèbre peut être le paramètre d'entrée d'un autre opérateur. Dans *OntoAlgebra*, tous les opérateurs retournent une relation dont le type est `SET[TUPLE]`. Ce type est celui du paramètre d'entrée de tous les opérateurs de *OntoAlgebra* à l'exception de `OntoProject` qui est `SET[ADT]`. Cependant, puisque le type `ADT` contient le type `TUPLE`, l'opérateur `OntoProject` accepte également une relation comme paramètre d'entrée. En conséquence, l'algèbre *OntoAlgebra* est fermée.

La notion de *complétude relationnelle* a été introduite par Codd [Codd, 1972] pour comparer le pouvoir d'expression des langages proposés pour le modèle relationnel. Une algèbre satisfait la propriété de complétude relationnelle si pour toute expression algébrique construite à partir de l'algèbre relationnelle, il existe une expression équivalente dans l'algèbre considérée. Les opérateurs de l'algèbre *OntoAlgebra* permettent de réaliser les différentes opérations de l'algèbre relationnelle :

- `OntoProject` permet de réaliser les opérations de projection ;
- `OntoSelect` permet de réaliser les opérations de sélection ;
- `OntoOJoin` appliquée avec un prédicat toujours vrai permet de réaliser le produit cartésien de deux relations ;
- `OntoUnion` et `OntoDifference` permettent de réaliser les opérations ensemblistes union et différence.

Chaque opération de l'algèbre relationnelle pouvant être réalisée avec un opérateur de l'algèbre *OntoAlgebra*, toute expression algébrique construite à partir de l'algèbre relationnelle à une expression équivalente utilisant les opérateurs de *OntoAlgebra*. Ceci établit la complétude relationnelle de cette algèbre.

Puisque toute requête OntoQL peut être exprimée comme une expression algébrique de *OntoAlgebra* qui possède les propriétés de fermeture et de complétude relationnelle, nous pouvons en déduire que le langage OntoQL possède également ces propriétés. Ce langage est même plus que complet au sens relationnel puisque *OntoAlgebra* présente des opérateurs qui permettent d'appliquer des fonctions sur les tuples d'une relation ou de grouper/dégrouper une relation. Une nouvelle définition de la complétude serait donc nécessaire pour caractériser le pouvoir d'expression du langage OntoQL et des autres langages de BDBO.

4.2 Optimisation logique de requêtes OntoQL

Une requête OntoQL est transformée en une séquence d'appels aux opérateurs de *OntoAlgebra* (composition). L'optimisation logique d'une requête OntoQL consiste ainsi à rechercher la séquence d'appels dont le coût d'évaluation risque d'être minimum. Ayant défini l'algèbre *OntoAlgebra* à partir de l'algèbre *Encore*, nous pouvons utiliser les techniques d'optimisation développées dans le contexte de cette algèbre.

4.2.1 Techniques d'optimisation traditionnelles

Les techniques d'optimisation proposées pour l'algèbre *Encore* sont basées sur la notion d'équivalence de requêtes. Des règles d'équivalence sont définies entre des expressions algébriques construites à partir de cette algèbre. Elles peuvent être utilisées pour réécrire une expression algébrique en une

nouvelle expression algébrique dont le coût d'évaluation risque d'être inférieur. Ayant défini l'algèbre *OntoAlgebra* à partir de l'algèbre *Encore*, la plupart des règles d'équivalence établies pour cette algèbre peuvent être utilisées pour optimiser une expression algébrique de *OntoAlgebra*. Ces règles sont données dans l'annexe C, section 3.

De plus, les opérateurs de l'algèbre *Encore* étant proches de ceux de l'algèbre relationnelle, ceci permet de réutiliser les techniques d'optimisation définies pour cette algèbre. Par exemple, une stratégie d'optimisation définie pour l'algèbre relationnelle consiste à réaliser les opérations de sélection avant celles de jointure. Cette stratégie peut également être suivie en utilisant la règle d'équivalence suivante :

$$\texttt{OntoSelect}(\texttt{OntoOJoin}(T, I_t, R, I_r, pred), pred_t) \Leftrightarrow \texttt{OntoOJoin}(\texttt{OntoSelect}(T, I_t, pred_t), R, I_r, pred) \quad (1)$$

Dans cette expression, $pred_t$ est un prédicat qui porte sur la relation de type T. Cette règle permet de restreindre les tuples de la relation T grâce au prédicat $pred_t$ avant d'effectuer la jointure entre les relations R et T.

Exemple. Considérons la requête suivante qui permet de rechercher le contenu des messages dont le titre contient l'adresse email des utilisateurs dont le nom est Dupont :

```
SELECT p.content FROM Post AS p, User AS u
 WHERE p.title LIKE '%' || u.email || '%' AND u.name = 'Dupont'
```

Une expression algébrique possible de cette requête est présentée sur la figure 5.3 (a) sous la forme d'un arbre. Une expression algébrique équivalente dont l'évaluation peut être plus efficace est présentée sur la figure 5.3 (b).

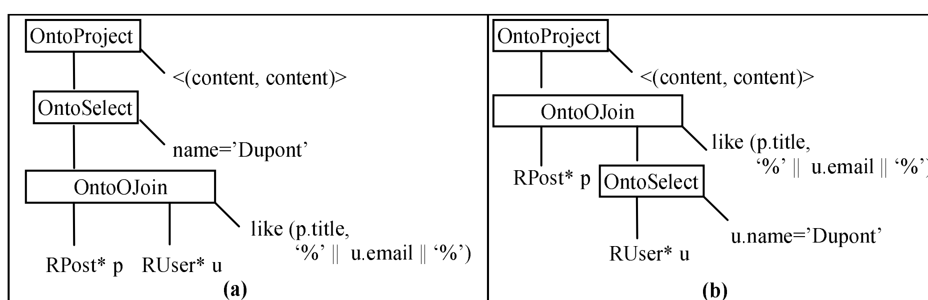


FIG. 5.3 – Exemple d'optimisation traditionnelle d'une requête OntoQL

Explication. Le prédicat `name = 'Dupont'` porte uniquement sur la relation RUSer*. En conséquence, la règle d'équivalence (1) peut être appliquée pour filtrer la relation RUser* avec ce prédicat avant de joindre cette relation avec la relation RPost*.

En plus des techniques d'optimisation définies pour l'algèbre relationnelle, les particularités du modèle de données sur lequel se base *OntoAlgebra* peuvent être utilisées pour en définir de nouvelles.

4.2.2 Techniques d'évaluation partielle appliquées aux requêtes OntoQL

Comme nous l'avons identifié lors de la définition du modèle de données d'une BDBO, certaines propriétés définies dans une ontologie peuvent ne pas être utilisées pour décrire les instances de cette ontologie. En conséquence, des techniques d'optimisation basées sur l'évaluation partielle d'une requête peuvent être définies. En effet, lorsqu'une propriété n'est pas utilisée dans l'extension d'une classe, il n'est pas nécessaire de rechercher la valeur de cette propriété pour les instances de cette classe puisque c'est la valeur NULL. Cette source d'optimisation est caractérisée par l'expression logique (2). Lorsque cette expression est évaluée à vrai, il devient possible de réduire, et parfois d'éviter d'avoir à accéder au niveau logique d'une BDBO.

Soit p une propriété, C une classe et pred un prédicat en forme normale conjonctive dont l'un des éléments implique une propriété satisfaisant l'expression logique (2) et n'utilise pas l'opérateur IS, alors :

$$\mathtt{p} \notin \mathtt{usedProperties(nomination(C))} \Rightarrow \quad (2)$$
$$\mathtt{OntoSelect}(\cdots \mathtt{OntoProject(C, ext(C), \ldots), pred)) = \emptyset \quad (3)$$

Cette règle d'optimisation peut également être appliquée pour l'opérateur OntoOJoin qui prend également un prédicat en paramètre. Cependant, cette optimisation est seulement possible pour des requêtes sur les instances directes d'une classe C portant sur une propriété p. Ainsi, elle ne peut pas être employée pour une expression algébrique utilisant la fonction $\mathtt{ext}^*$ ou des expressions de chemins. Cependant, des règles d'équivalence permettent de transformer une expression algébrique qui les utilise en une expression équivalente qui ne les utilise pas.

La fonction $\mathtt{ext}^*$ peut être retirée d'une expression algébrique en utilisant l'opérateur OntoUnion. Soit $\theta_1 \cdots \theta_n$ un ensemble d'opérateurs de *OntoAlgebra*, C une classe ayant $C_1, \ldots, C_n$ comme sous-classes directes. Retirer la fonction $\mathtt{ext}^*$ peut être réalisé en appliquant récursivement la règle d'équivalence suivante (opération de dépliage) :

$$\begin{aligned}\theta_1(\ldots\theta_n(\mathtt{OntoProject}(C, \mathtt{ext}^*(C), \ldots))) \Leftrightarrow \mathtt{OntoUnion}(&\\ \ldots \mathtt{OntoUnion}(\theta_1(\ldots\theta_n(\mathtt{OntoProject}(C, \mathtt{ext}(C), \ldots))),&\\ \theta_1(\ldots\theta_n(\mathtt{OntoProject}(C_1, \mathtt{ext}^*(C_1), \ldots)))) \ldots&\\ , \theta_1(\ldots\theta_n(\mathtt{OntoProject}(C_n, \mathtt{ext}^*(C_n), \ldots)))& \quad (4)\end{aligned}$$

Une expression de chemin peut être retirée d'une expression algébrique en utilisant l'opérateur Onto-LeftOuterOJoin. Soit p_2 une propriété de domaine C_2, θ un opérateur booléen et cste une constante. L'équivalence suivante peut être utilisée pour retirer une expression de chemin d'une expression algébrique :

$$\begin{aligned}&\mathtt{OntoSelect}(T, I_t, \mathtt{DEREF}(p_1).p_2\ \theta\ \mathtt{cste}) \Leftrightarrow\\ &\quad \mathtt{OntoSelect}(\mathtt{OntoLeftOuterOJoin}(T, I_t, RC_2^*, \lambda c_1\ \lambda c_2 \bullet c_1.p_1 = c_2.\mathtt{oid}), p_2\ \theta\ \mathtt{cste}) \quad (5)\end{aligned}$$

Cette règle peut être appliquée plusieurs fois pour retirer une expression de chemin d'une longueur quelconque. De plus, des règles d'équivalence similaires peuvent être définies pour l'ensemble des opérateurs qui peuvent prendre une expression de chemin en paramètre (OntoProject, les opérateurs de jointure et OntoSort). Ces règles suppriment l'expression de chemin de l'expression algébrique mais introduisent

la fonction $\texttt{ext}^*$ ($\texttt{RC}_2^*$). Cependant, cette fonction peut être retirée en utilisant la règle d'équivalence (4). L'exemple suivant illustre l'utilisation combinée de ces deux règles pour optimiser une requête OntoQL en utilisant les règles d'équivalence présentées dans cette section.

Exemple. Dans cet exemple, nous supposons que la classe `User` possède une sous-classe `Administrator` pour laquelle la propriété `email` n'est pas utilisée pour en décrire les instances. La requête a optimiser est la suivante :

```
SELECT p.title FROM Post AS p WHERE p.has_creator.email LIKE '%@ensma.fr'
```

Les différentes transformations qui peuvent permettent d'optimiser l'exécution de cette requête sont présentées sur la figure 5.4.

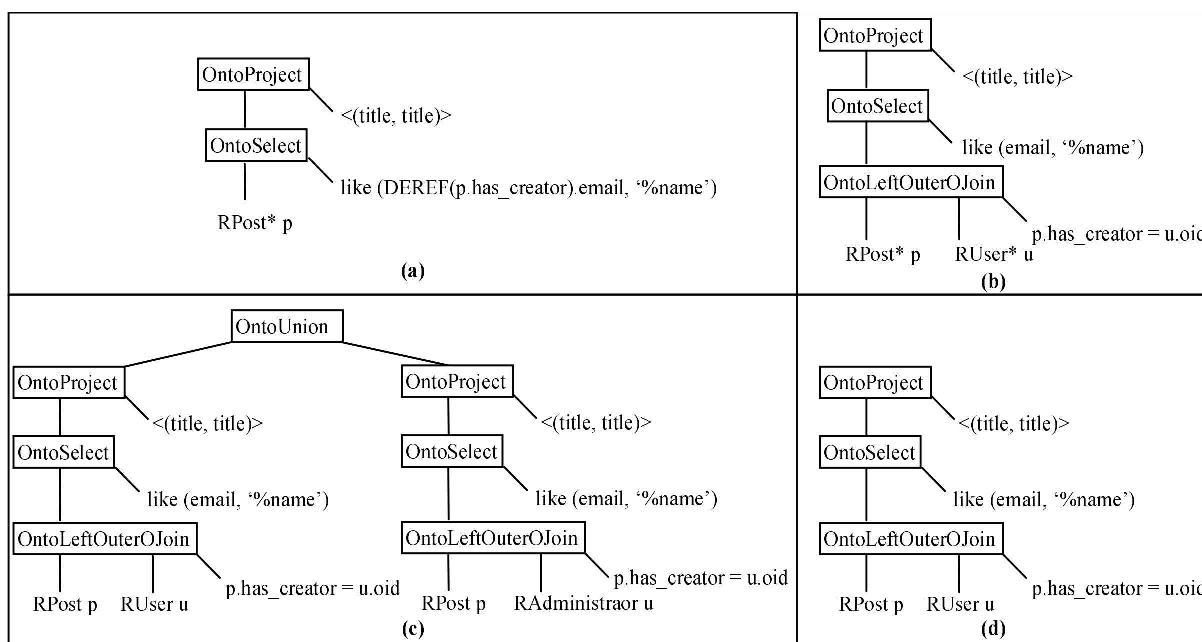


FIG. 5.4 – Exemple d'optimisation d'une requête OntoQL par évaluation partielle

Explication. L'expression algébrique initiale de cette requête est présentée sur la figure 5.4 (a). L'expression de chemin impliquant les propriétés `has_creator` et `email` est retirée de cette expression algébrique en appliquant la règle (5). L'expression obtenue est présentée sur la figure 5.4 (b). Cette expression utilise la fonction $\texttt{ext}^*$ pour construire les relations $\texttt{RPost}^*$ et $\texttt{RUser}^*$. Pour retirer ces fonctions dans cette expression, il est donc nécessaire d'appliquer deux fois la règle (4) qui permet de déplier une requête en utilisant l'opérateur `OntoUnion`. La classe `Post` n'ayant pas de sous-classe, l'application de cette règle consiste simplement à remplacer `RPost*` par `RPost`. Par contre, la classe `User` a pour sous-classe `Administrator`. L'expression algébrique doit donc être dupliquée pour `RPost` et `RAdministrator` ce qui donne l'expression présentée sur la figure 5.4 (c). La propriété `email` n'étant pas utilisée pour décrire les instances de la classe `Administrator`, la règle (3) permet de déduire que l'application de l'opérateur `OntoSelect` sur la relation, construite à partir de cette classe, retourne l'ensemble vide. L'application de l'opérateur `OntoProject` à cet ensemble vide retourne également l'ensemble vide. Puisque l'ensemble vide est un élément neutre pour l'opérateur `OntoUnion`, l'expression algébrique concernant la classe `Administrator` peut être supprimée du

plan d'exécution de la requête. En conséquence, l'expression algébrique finale de cette requête est celle présentée sur la figure 5.4 (d).

Cet exemple montre que l'optimisation logique de requête OntoQL peut être réalisée en utilisant les techniques d'évaluation partielle proposées dans cette section. L'optimisation de requêtes OntoQL reste cependant un problème ouvert.

5 Conclusion

Dans ce chapitre, nous avons d'abord défini l'algèbre *OntoAlgebra* qui permet d'interroger les données, les ontologies et à la fois les ontologies et les données d'une BDBO. Pour définir cette algèbre nous avons choisi de partir du modèle de données et l'algèbre d'opérateurs *Encore*. Nous avons observé que ce modèle de données et cette algèbre d'opérateurs ne pouvaient pas être utilisés pour toute BDBO sans un effort d'adaptation. Nous les avons donc spécialisés et étendus pour prendre en compte les différences entre une BDBO et une BDOO. Le résultat est une algèbre, *OntoAlgebra*, utilisée pour représenter toute requête OntoQL, de manière systématique, en appliquant des règles de transformation que nous avons définies. L'algèbre *OntoAlgebra* constitue donc la sémantique formelle du langage d'interrogation OntoQL.

Ensuite, nous avons utilisé l'algèbre *OntoAlgebra* pour, dans un premier temps, étudier les propriétés sémantiques du langage OntoQL. La proximité entre l'algèbre *OntoAlgebra* et l'algèbre relationnelle nous permet d'établir la fermeture et la complétude relationnelle du langage OntoQL. Dans un second temps, nous avons utilisé l'algèbre *OntoAlgebra* pour étudier l'optimisation logique de requêtes OntoQL. Nous avons montré que l'optimisation de telles requêtes peut être réalisée en combinant les techniques traditionnelles des bases de données (par exemple, pousser les sélections vers le bas de l'arbre algébrique) et les techniques d'évaluation partielle qui exploitent le fait qu'une propriété peut ne pas être utilisée pour caractériser les instances des classes sur lesquelles elle est applicable. Ce travail sur l'optimisation de requête OntoQL est préliminaire. Pour être complet, il nous reste à mener les activités suivantes :

- formaliser les preuves de complétude et de fermeture du langage OntoQL ;
- mener des évaluations de performance pour montrer l'intérêt des optimisations proposées ;
- étudier l'optimisation des requêtes portant à la fois sur l'ontologie et les données ;
- déterminer des structures redondantes (par exemple, des vues matérialisées) et/ou non redondantes (par exemple, des index) qui permettent d'optimiser le traitement de requête ;
- étudier comment se combinent les optimisations qui peuvent être réalisées au niveau de *OntoAlgebra* et celles réalisées par le SGBD lorsque l'expression algébrique est traduite en SQL.

La définition formelle du langage OntoQL présentée dans ce chapitre est fondamentale pour l'implantation de ce langage sur une BDBO. Nous montrons dans le chapitre suivant comment nous l'avons utilisée pour implanter le langage OntoQL sur la BDBO OntoDB.

Chapitre 6

Implantation du langage OntoQL sur le prototype OntoDB

Sommaire

1 Introduction . . . 191

2 Le prototype de BDBO OntoDB . . . 191

2.1 L'architecture OntoDB . . . 191

2.2 Représentation des données . . . 192

2.3 Représentation des ontologies . . . 194

3 Implantation du modèle de données de OntoQL sur OntoDB . . . 196

3.1 Représentation des données . . . 196

3.2 Représentation des ontologies . . . 198

3.3 Représentation du modèle d'ontologies . . . 200

4 Implantation d'un interpréteur de requêtes OntoQL sur OntoDB . . . 202

4.1 Traitement d'une requête OntoQL . . . 202

4.2 Traduction des requêtes OntoQL sur les données . . . 204

4.3 Traduction des requêtes OntoQL sur les ontologies . . . 209

4.4 Traduction des requêtes OntoQL sur les ontologies et sur les données . . . 211

5 Interfaces usuelles d'un langage d'interrogation de bases de données . . . 214

5.1 Interface interactive OntoQLPlus . . . 214

5.2 Interface graphique OntoQBE . . . 215

5.3 Interface JAVA JOBDBC . . . 216

5.4 Interface JAVA OntoAPI . . . 218

6 Interfaces spécifiquement conçues pour les BDBO . . . 221

6.1 Interface par mots clés . . . 221

6.2 Interface avec le langage SPARQL . . . 223

7 Conclusion . . . 230

Résumé. Dans le chapitre précédent, nous avons présenté la sémantique formelle du langage OntoQL. Dans ce chapitre, à partir de la sémantique définie, nous montrons dans un premier temps la faisabilité de son implantation en présentant sa mise en oeuvre sur la

BDBO OntoDB [Jean et al., 2007c]. Cette implantation a été conçue, d'une part, pour dépendre le moins possible de la BDBO sous-jacente (en l'occurence, OntoDB) et, d'autre part, pour favoriser l'optimisation des requêtes exécutées. Dans un second temps, nous présentons les développements que nous avons menés autour de ce langage pour compléter les outils permettant de manipuler les ontologies et les données à base ontologique stockées dans la BDBO OntoDB. Cette dernière est actuellement équipée d'outils présents dans les SGBD traditionnels tels que des éditeurs de requêtes ou des API d'accès. Elle dispose également d'une interface de recherche par mots clés des concepts des ontologies et d'une sur-couche d'accès à l'aide du langage SPARQL.

1 Introduction

Dans le chapitre précédent, nous avons présenté la sémantique formelle du langage OntoQL. Ce langage permet de définir, manipuler et interroger des BDBO indépendamment du modèle d'ontologies qu'elles supportent. Pour valider ce résultat, nous avons implanté ce langage sur la BDBO OntoDB [Jean et al., 2007c]. Cette implantation est présentée dans ce chapitre.

Nous avons choisi la BDBO OntoDB pour deux raisons principales. La première raison est que nous avions un accès total au code et à la documentation de cette BDBO, celle-ci ayant été développée au sein de notre laboratoire. La seconde raison est le besoin d'équiper cette BDBO d'un langage d'exploitation afin d'enrichir les capacités d'accès aux ontologies et aux données qu'elle permet de stocker. En effet, autour de cette BDBO, une suite logicielle qui permet de visualiser, éditer, importer/exporter et intégrer les ontologies et les données qu'elle contient a été définie. Cette suite logicielle est utilisée dans différents projets dans lesquels le besoin d'un langage d'exploitation s'est fait ressentir.

L'implantation du langage OntoQL sur la BDBO OntoDB répond ainsi à deux objectifs. D'abord, montrer la faisabilité de l'implantation de ce langage sur une BDBO et ensuite, compléter la suite logicielle construite autour de la BDBO OntoDB en mettant à disposition des utilisateurs et développeurs un langage d'exploitation et des outils facilitant son utilisation.

Ce chapitre est organisé comme suit. Dans la section suivante, nous présentons le prototype de BDBO OntoDB. Nous montrons l'architecture sur laquelle s'appuie ce prototype et la représentation choisie pour stocker des ontologies et des données à base ontologique. Nous montrons ensuite, dans la section 3, comment nous avons utilisé et entendu la représentation proposée par OntoDB pour implanter le modèle de données sur lequel s'appuie le langage OntoQL. L'implantation d'un interpréteur de requêtes de ce langage est présentée dans la section 4. Nous présentons les différentes étapes de traitement d'une requête puis montrons comment les requêtes sur les données, sur les ontologies et à la fois sur les ontologies et sur les données sont traduites en SQL pour pouvoir être exécutées sur OntoDB. Les deux sections suivantes sont consacrées aux outils que nous avons développés pour compléter les outils disponibles sur OntoDB. La section 5 présente les outils conçus en nous inspirant de ceux proposés pour les SGBD traditionnels tandis que la section 6 présente deux outils conçus spécifiquement pour répondre aux besoins d'exploitation de la BDBO OntoDB.

2 Le prototype de BDBO OntoDB

La BDBO OntoDB a été validée par un prototype implanté sur le SGBD PostgreSQL. Dans cette section, nous présentons les principaux éléments de ce prototype utilisés pour implanter le langage OntoQL. Nous commençons par donner une vue d'ensemble de la BDBO OntoDB en présentant son architecture.

2.1 L'architecture OntoDB

La figure 6.1 présente les quatre parties qui composent l'architecture de OntoDB :

1. **La partie *métabase* (1).** La *métabase*, souvent appelée *system catalog*, est une partie traditionnelle

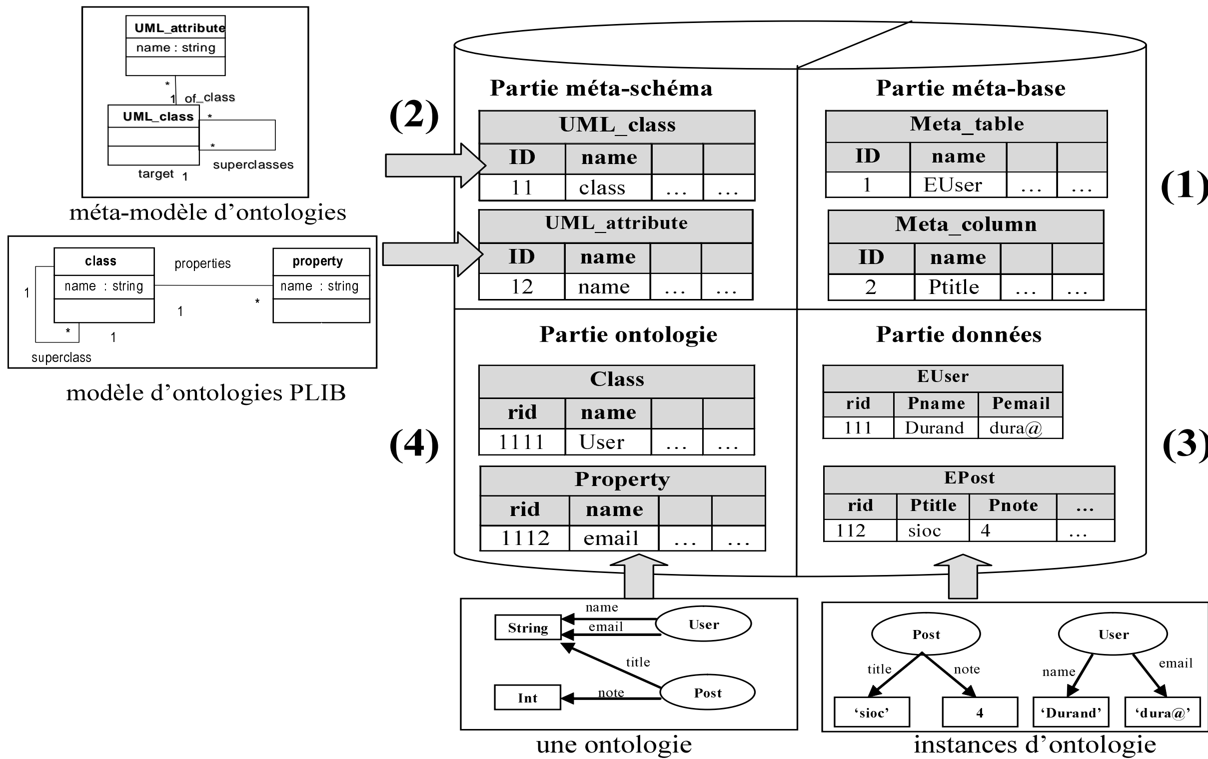


FIG. 6.1 – Architecture OntoDB

des bases de données classiques. Elle est constituée de l'ensemble des tables système. Ces tables sont celles dont le SGBD se sert pour gérer et assurer le fonctionnement de l'ensemble des données contenues dans la base de données. Dans OntoDB, elle contient en particulier la description de toutes les tables et colonnes définies dans les trois autres parties de cette architecture.

2. **La partie *données* (3).** Elle représente les objets du domaine décrits en termes d'une classe d'appartenance et d'un ensemble de valeurs de propriétés applicables à cette classe. Ces objets sont représentés en associant une table à chaque classe (représentation horizontale).
3. **La partie *ontologie* (4).** Elle contient les ontologies définissant la sémantique des différents domaines couverts par la base de données. Ce prototype supporte le modèle d'ontologies PLIB.
4. **La partie *méta-schéma* (2).** Elle représente, au sein d'un modèle réflexif, à la fois le modèle d'ontologies utilisé et le *méta-schéma* lui-même. Le *méta-schéma* est pour la partie *ontologie*, ce qu'est la *métabase* pour la partie données.

Pour implanter le langage OntoQL, nous avons utilisé les parties *données* (3) et *ontologie* (4). Dans les sections suivantes, nous présentons la représentation proposée par OntoDB pour ces deux parties.

2.2 Représentation des données

OntoDB utilise la représentation horizontale pour stocker les données. Dans cette représentation, une table est associée à chaque classe qui n'est pas abstraite. Cette table contient une colonne permettant d'identifier les instances de cette classe. Cette colonne, nommée `rid`, est définie sur la table `root_ta-`

ble_extension, super-table de toutes celles associées à des classes. En plus de cette colonne, chacune des tables contient une colonne pour chaque propriété utilisée pour décrire les instances de la classe correspondante. Afin de conserver le lien entre les ontologies et les données qu'elles décrivent, OntoDB utilise les identifiants (oid) des classes et des propriétés pour nommer cette table et ses colonnes. Le nom de cette table (respectivement d'une de ses colonnes) est la concaténation de la chaîne "E" (respectivement "P") avec l'oid de la classe (respectivement de la propriété).

Chaque propriété étant représentée comme une colonne d'une table, des règles de correspondance ont été définies entre les types de données PLIB et ceux de PostgreSQL. Les principales règles de correspondances définies pour ces types données sont les suivantes :

– les types simples ont un équivalent direct avec un type de PostgreSQL. Ces équivalences sont données dans la table suivante :

Type du modèle PLIB	**Type PostgreSQL correspondant**
Entier (int_type)	INT8
Réel (real_type)	FLOAT8
Booléen (boolean_type)	BOOLEAN
Chaîne de caractères (string_type)	VARCHAR

– le type référence (class_instance_type) n'a pas d'équivalent en PostgreSQL. Une propriété de ce type dans OntoDB est représentée par deux colonnes. La première, dont le nom est suffixé par _rid, indique l'identifiant de l'instance référencée. La seconde dont le nom est suffixé par _tablename indique le nom de la table dans laquelle cette instance est stockée ;

– le type collection (aggregate_type) a pour équivalent le type ARRAY de PostgreSQL. OntoDB utilise ce type pour représenter les collections de type simple. Les propriétés de type collection de références sont représentées par deux colonnes de type ARRAY. La première dont le nom est suffixé par _rids indique les identifiants des instances référencées. La seconde dont le nom est suffixé par _tablenames indique les noms des tables dans lesquelles ces instances sont stockées.

Exemple. La figure 6.2 présente un exemple de représentation d'instances de l'ontologie SIOC dans la partie données de OntoDB.

Table EItem	
rid	**Pname**
1	Logo de l'ontologie SIOC
...	...

Table EPost			
rid	**Ptitle**	**Phas_creator_rid**	**Phas_creator_tablename**
2	L'ontologie SIOC	10	EUser
...	...	...	...

Table EUser				
rid	**Plast_name**	**Pemail**	**Psubscriber_of_rids**	**Psubscriber_of_tablenames**
10	Dupont	dupont@gabaro.com	[100,101]	[EForum, EContainer]
...	...	...	...	...

FIG. 6.2 – Exemple de représentation des données dans OntoDB

Explication. Dans OntoDB, les tables et les colonnes sont nommées en utilisant les identifiants internes

des classes et des propriétés (des entiers comme, par exemple, E1162). Ces noms étant peu lisibles, nous utilisons dans cet exemple et dans les exemples suivants les identifiants externes des classes et des propriétés, proches des noms en anglais. La table EItem conserve ainsi les instances de la classe Item. Elle contient la colonne rid qui identifie ces instances et la colonne Pname qui correspond à la propriété name. Cette propriété étant de type String, la colonne Pname est de type VARCHAR. Dans cet exemple, nous avons donc supposé que les instances de la classe Item ne sont décrites que par la propriété name et non par les autres propriétés applicables sur cette classe (has_reply et has_container). La table EPost correspond à la classe Post. Ces instances sont décrites par la propriété titre de type String représentée par la colonne Ptitle de type VARCHAR. Elles sont également décrites par la propriété has_creator de type référence vers les instances de la classe User. Cette propriété est donc représentée par deux colonnes, l'une indiquant l'identifiant de l'instance référencée et l'autre indiquant la table dans laquelle cette instance est rangée. L'instance référencée dans cet exemple a pour identifiant 10. Elle est stockée dans la table EUser présentée en dessous. Cette table présente la particularité de contenir deux colonnes pour représenter la propriété subscriber_of de type collection de références vers des instances de la classe Forum. Comme le montre cet exemple les instances référencées dans ces collections peuvent appartenir à différentes tables.

OntoDB permet ainsi de représenter les données en utilisant une représentation horizontale. Nous présentons maintenant la représentation proposée pour les ontologies.

2.3 Représentation des ontologies

La partie *ontologie* de l'architecture OntoDB permet de stocker des ontologies PLIB. Compte tenu de la complexité du modèle d'ontologies PLIB, un programme a été réalisé pour générer le modèle logique de cette partie. Ce programme prend en entrée une représentation orientée-objet du modèle d'ontologies PLIB dans le langage de spécification EXPRESS [Schenk and Wilson, 1994]. Il applique à ce modèle des règles de transformation entre les mécanismes orientés-objets de EXPRESS et ceux du modèle relationnel-objet implanté par PostgreSQL afin de générer le script de création des tables de cette partie.

Les deux principales règles de transformation concernent l'héritage et les associations entre entités EXPRESS. L'héritage entre entités est représenté en utilisant l'héritage de tables disponible dans PostgreSQL. Ainsi chaque entité EXPRESS du modèle PLIB correspond à une table dans le modèle logique dont le nom est celui de l'entité suffixé par _e. Si une entité hérite d'une autre entité, sa table hérite également de la table correspondant à cette autre entité. Quelle que soit leur cardinalité, les associations sont représentées par l'intermédiaire de tables dites *d'aiguillage*. Pour une entité A, liée à l'entité B par une association nommée a2b, une table d'aiguillage nommée A_2_a2b est créée. Cette table contient les 5 colonnes suivantes :

- rid qui identifie les associations ;
- rid_s et tablename_s qui indiquent respectivement l'identifiant d'une instance de l'entité A impliquée dans une association et la table dans laquelle cette instance est stockée ;
- rid_d et tablename_d jouent le même rôle que rid_s et tablename_s pour une instance de l'entité B.

De plus, afin d'accélérer les accès à l'association, la table `A_e` contient également une colonne `a2b` dont le contenu est une clé étrangère (cas d'une association `1:m`) ou une collection de clés étrangères (cas d'une association `n-m`) vers la table d'aiguillage.

Exemple. La figure 6.3 présente un exemple simplifié de la représentation des ontologies dans OntoDB.

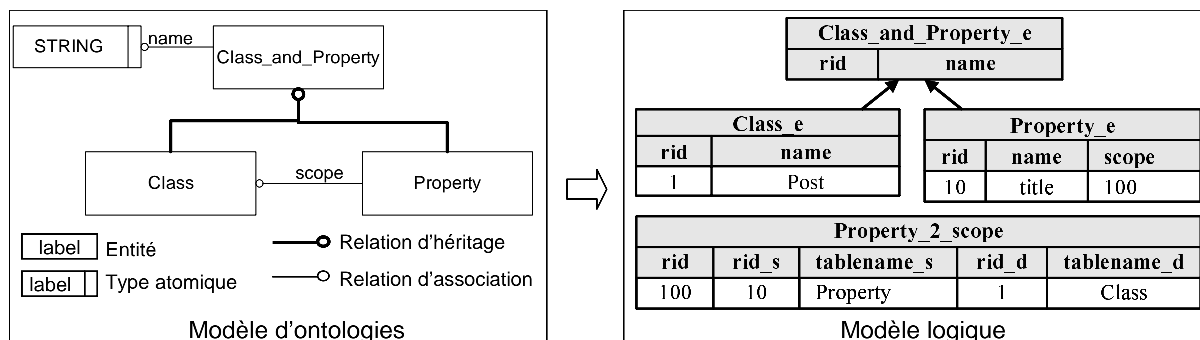


FIG. 6.3 – Exemple de représentation des ontologies dans OntoDB

Explication. Sur la partie gauche de cette figure se trouve un modèle d'ontologies simple représenté en EXPRESS. Ce modèle d'ontologies est constitué de trois entités. L'entité `Class_and_Property` a un seul attribut nommé `name`. Cet attribut indique le nom d'une classe ou d'une propriété. `Class_and_Property` est la super-entité des entités `Class` et `Property` qui représentent les classes et propriétés des ontologies. Ces deux entités sont liées par l'association `scope` qui permet de représenter le domaine d'une propriété.

Le modèle logique utilisé par OntoDB pour conserver des ontologies conformes à ce modèle d'ontologies est présenté sur la partie droite de cette même figure. Une table est créée pour chaque entité. Elles possèdent toutes l'attribut `rid` qui fournit une identification interne à la base de données des classes et des propriétés. Conformément à la hiérarchie d'entités dans le modèle d'ontologies, les tables `Class_e` et `Property_e` héritent de la table `Class_and_Property_e`. Enfin, pour représenter l'association `scope`, la table d'aiguillage `Property_2_scope` a été créée. Elle fait le lien entre une propriété et une classe. L'identifiant de cette propriété (respectivement de cette classe) ainsi que la table dans laquelle elle est stockée sont indiqués dans les colonnes `rid_s` et `tablename_s` (respectivement `rid_d` et `tablename_d`). La colonne `scope` de la table `Property_e` permet d'accéder à cette association depuis cette table. En effet, cette colonne est une clé étrangère vers la table d'aiguillage.

Nous avons présenté comment OntoDB permet de représenter des données et des ontologies conformes au modèle de données PLIB. La principale difficulté posée par l'implantation du langage OntoQL sur OntoDB est que ce langage est basé sur un modèle de données construit autour d'un modèle d'ontologies noyau qui peut être étendu. Ceci nécessite de résoudre les problèmes suivants :

– identification de la représentation du modèle noyau de OntoQL sur OntoDB et, si certains constructeurs ne sont pas représentés, extension de cette représentation ;
– possibilité d'étendre dynamiquement la représentation proposée par OntoDB lorsque de nouveaux constructeurs sont ajoutés au modèle noyau de OntoQL en prenant en compte le cas où ces exten-

sions sont déjà représentées sur OntoDB (cas des constructeurs du modèle PLIB qui ne sont pas présents dans le modèle noyau de OntoQL).

Les solutions que nous avons mises en place pour résoudre ces problèmes sont présentées dans la section suivante.

3 Implantation du modèle de données de OntoQL sur OntoDB

Le modèle de données OntoQL a été défini pour manipuler des données à base ontologique, les ontologies qui les décrivent et le modèle d'ontologies utilisé pour définir ces ontologies. Nous montrons comment nous avons implanté ces différents éléments sur OntoDB dans les sections suivantes.

3.1 Représentation des données

Les données manipulées par OntoQL sont conformes au modèle de données suivant. Chaque classe est associée à une extension qui en stocke les instances. L'extension d'une classe est constituée de l'ensemble des propriétés utilisées pour en décrire les instances. Ces propriétés ont un codomaine qui est un des types définis dans le modèle noyau de OntoQL ou ajouté à ce modèle noyau. L'implantation de ce modèle de données sur OntoDB consiste donc à définir la représentation d'une extension de classe dans cette architecture.

3.1.1 Représentation des données conformes au modèle noyau de OntoQL

Dans OntoDB, une table est créée pour chaque classe d'une ontologie. Pour chaque propriété utilisée pour décrire les instances de cette classe, une ou plusieurs colonnes sont créées dans cette table. Ainsi, une extension de classe est implantée par une table dans OntoDB. Et, chaque propriété associée à une extension est implantée par une ou plusieurs colonnes de la table qui implante cette extension.

Par exemple, la table `EPost` présentée sur la figure 6.2 est l'implantation de l'extension de la classe `Post`. Les colonnes `Phas_creator_rid` et `Phas_creator_tablename` de cette table implantent la propriété `has_creator` associée à cette extension.

OntoDB permet de représenter des propriétés dont le codomaine est un type PLIB. Comme le montre le tableau 6.1, la plupart des types de données définis dans le modèle d'ontologies noyau de OntoQL ont un équivalent dans le modèle PLIB. OntoDB permet donc de représenter les propriétés dont le codomaine est un de ces types de données.

Ce n'est par contre pas le cas pour les propriétés dont le codomaine est `#DateType` ou `#MultilingualString`. Pour permettre de représenter des propriétés de ces types de données, nous avons étendu les règles de correspondance définies sur OntoDB. Le type `#DateType` est un type simple dont l'équivalent en PostgreSQL est le type `Date`. Pour la représentation du type `#MultilingualString`, nous avons suivi l'approche proposée par OntoDB pour représenter le type `level_type` de PLIB. Une valeur de ce type contient quatre valeurs caractérisées par des qualificatifs (`min`, `nom`, `typ` et `max`). Chacune de ces valeurs est stockée dans une colonne suffixée par son qualificatif. Le type `#MultilingualString` est similaire. Une valeur de ce type peut contenir une valeur pour chaque langue naturelle supportée.

Type de données dans le modèle OntoQL	Type de données dans le modèle PLIB
#IntType	int_type
#RealType	real_type
#BooleanType	boolean_type
#StringType	string_type
#DateType	
#MultilingualStringType	
#RefType	class_instance_type
#CollectionType	aggregate_type

Tab. 6.1 – Correspondances entre les types du modèle noyau de OntoQL et ceux de PLIB

OntoQL permettant d'utiliser plus d'une centaine de langues naturelles, créer une colonne pour chacune d'elles résulterait en une table importante contenant de nombreuses valeurs nulles. Ceci provoquerait le problème d'inefficacité connu sous le nom de rareté (ou *sparsity*) [Agrawal et al., 2001]. Cependant, chaque ontologie PLIB est associée aux langues naturelles utilisées pour en décrire ses concepts. Pour les ontologies que nous avons manipulées, ce nombre n'excède pas 3. Dans ces conditions, utiliser la représentation proposée pour le type `level_type` n'engendre pas le problème de rareté. Une valeur de type `#MultilingualString` est ainsi stockée dans plusieurs colonnes, une par langue naturelle utilisée. Le nom de cette colonne est suffixé par le code de cette langue naturelle. Pour une autre BDBO dans laquelle les langues naturelles utilisées ne seraient pas précisées, une autre représentation serait à envisager.

Exemple. La figure 6.4 présente la représentation des instances de la classe `Post` de l'ontologie SIOC en considérant que le codomaine de la propriété `title` est `#MultilingualString`.

Table EPost				
rid	Ptitle_FR	Ptitle_EN	Phas_creator_rid	Phas_creator_tablename
2	L'ontologie SIOC	The SIOC ontology	10	EUser
...	...	...	...	...

Fig. 6.4 – Exemple de représentation des données conformes au modèle noyau de OntoQL dans OntoDB

Explication. Dans cet exemple, nous supposons que le français (`FR`) et l'anglais (`EN`) sont les seules langues naturelles employées pour décrire les concepts et les instances de l'ontologie SIOC. En conséquence, la propriété `title` est représentée par deux colonnes de la table `EPost` qui contient les instances de la classe `Post`. La première colonne nommée `Ptitle_FR` contient la valeur en français des instances de la classe `Post` pour la propriété `title`. La seconde, nommée `Ptitle_EN`, contient les valeurs en anglais de cette propriété pour ces mêmes instances.

3.1.2 Représentation des données utilisant des extensions du modèle noyau de OntoQL

Les propriétés associées à une extension peuvent également avoir pour codomaine un type de données ajouté au modèle d'ontologies noyau de OntoQL. Ce type de données est défini comme une sous-entité

d'un des types de données du modèle noyau. La représentation de ce type de données est faite de la même façon que pour le type dont il hérite. Par exemple, si le type PLIB `int_measure_type` est ajouté comme sous-entité du type `#IntType`, une propriété d'une extension ayant ce type pour codomaine sera représentée par une colonne de type `INT8`.

Exemple. La figure 6.5 présente la représentation des instances de la classe `User` de l'ontologie SIOC. Dans cet exemple, nous supposons que les instances de cette classe sont décrites par la propriété `salary` du type PLIB `int_currency_type` qui permet d'associer une monnaie à un ensemble d'entiers. Ceci nécessite que ce type ait été ajouté au modèle noyau de OntoQL comme sous-entité du type `#IntType`. Nous supposons également que ces instances sont décrites par une propriété `first_names` qui est une liste de chaîne de caractères, le type liste ayant été ajouté comme sous-entité de `#CollectionType`.

Table EUser				
rid	**Pfirst_names**	**Plast_name**	**Psalary**	**PEmail**
10	[Patrick, Henry]	Dupont	1614	dupont@gabaro.com
...	...	...	...	...

Fig. 6.5 – Exemple de représentation de données utilisant des extensions du modèle noyau de OntoQL dans OntoDB

Explication. Le type `int_currency_type` ayant été ajouté comme sous-entité du type `#IntType`, la propriété `salary` dont le codomaine est ce type est représentée par une colonne de type `INT8` (représentation du type `#IntType`). De même, la propriété `first_names` est représentée par une colonne de type tableau de chaîne de caractères (`VARCHAR ARRAY`) puisque son type est une sous-entité du type `#CollectionType`.

3.2 Représentation des ontologies

Les ontologies manipulées par OntoQL sont conformes au modèle noyau de OntoQL et à ses éventuelles extensions.

3.2.1 Représentation des ontologies conformes au modèle noyau de OntoQL

Pour que OntoQL permette d'exploiter les ontologies contenues dans OntoDB nous avons cherché à identifier la représentation du modèle noyau de OntoQL sur cette BDBO. Dans OntoDB, le modèle logique utilisé pour stocker les ontologies a été créé à partir du modèle d'ontologies PLIB. Pour identifier la représentation du modèle noyau de OntoQL sur cette BDBO, il est donc nécessaire d'établir les correspondances entre ce modèle noyau et le modèle d'ontologies PLIB (*mapping*). La plupart des éléments du modèle d'ontologies noyau de OntoQL ont un équivalent dans le modèle PLIB. Le tableau 6.2 montre un extrait des correspondances entre les entités et attributs du modèle noyau de OntoQL et ceux du modèle PLIB[30]. Chaque ligne de ce tableau présente le mapping d'une entité OntoQL et de ses attributs avec une entité PLIB et des attributs PLIB dont le domaine est précisé entre parenthèses lorsque

[30]L'ensemble des correspondances établies est présenté dans l'annexe D.

ce n'est pas l'entité PLIB de cette case. Ainsi, l'entité #Concept de OntoQL correspond à l'entité Class_and_Property_elements de PLIB. Seul l'attribut #definition de #Concept correspond à un attribut de Class_and_Property_elements. Les autres attributs sont définis sur d'autres entités PLIB liées par un lien d'association à Class_and_Property_elements. Par exemple, l'attribut #code est mis en correspondance avec l'attribut du même nom défini sur l'entité Basic_semantic_unit liée à l'entité Class_and_Property_elements par le lien d'association identified_by. L'entité #Class de OntoQL correspond à l'entité du même nom en PLIB. Son attribut #directSuperClasses correspond à l'union des attributs PLIB its_superclass et is_case_of. Enfin, l'entité #Property de OntoQL correspond à l'entité Property_det de PLIB. Son attribut #scope correspond à l'attribut name_scope défini sur l'entité Property_bsu. Son attribut #range correspond à l'attribut domain de l'entité Property_det.

Pour toutes les entités et tous les attributs du modèle noyau de OntoQL qui ont un équivalent en PLIB, nous pouvons donc utiliser la représentation proposée par OntoDB pour en stocker les instances. Par exemple, les propriétés OntoQL, qui sont des instances de l'entité #Property, seront stockées dans la table Property_det_e créée pour les propriétés PLIB. De même, le domaine d'une propriété OntoQL donné par l'attribut #scope sera stocké dans la table d'aiguillage Property_bsu_2_name_scope créé pour le domaine des propriétés PLIB.

Entité et attribut OntoQL	**Entité et attribut PLIB**
#Concept _#code _#name _#definition _#definedBy	Class_and_Property_elements _code (Basic_semantic_unit) _preferred_name (Item_names) _definition _defined_by (Class_bsu)
#Class _#directSuperClasses	Class _its_superclass ∪ is_case_of (Item_class_case_of)
#Property _#scope _#range	Property_det _name_scope (Property_bsu) _domain

Tab. 6.2 – Extrait des correspondances entre les entités et attributs du modèle noyau de OntoQL et ceux de PLIB

Par rapport au modèle noyau de OntoQL, seuls les types #Date et #MultilingualString n'ont pas d'équivalent dans ce modèle. Nous avons donc ajouté à OntoDB la représentation de ces deux entités conformément aux règles de transformation définies. Avec cette extension, OntoDB permet de stocker des ontologies conformes au modèle noyau de OntoQL.

3.2.2 Représentation des ontologies utilisant des extensions du modèle noyau de OntoQL

De nouvelles entités et de nouveaux attributs peuvent être ajoutés au modèle noyau de OntoQL. Pour permettre de stocker des ontologies utilisant ces extensions du modèle noyau, ces nouvelles entités et ces nouveaux attributs sont représentés par des tables et/ou colonnes créées conformément à la représentation

proposée par OntoDB. Par exemple, si le constructeur OWL de restriction `#Restriction` est ajouté au modèle noyau comme une entité héritant de l'entité `#Class`, la table `Restriction_e`, sous-table de `Class_e`, sera créée pour stocker les restrictions OWL qui pourront être définies. Cette entité a un attribut `#onProperty` qui permet d'indiquer la propriété sur laquelle porte la restriction. Cet attribut sera représenté par une table d'aiguillage `Restriction_2_onProperty`.

Ceci ne règle pas le problème des entités et attributs PLIB qui sont déjà représentés dans OntoDB et que l'on souhaite ajouter au modèle noyau de OntoQL. Nous montrons dans la section suivante comment nous avons traité ce cas.

3.3 Représentation du modèle d'ontologies

Le langage OntoQL présente la particularité de permettre de manipuler le modèle d'ontologies utilisé. Ce modèle d'ontologies doit donc être rendu persistant. La BDBO OntoDB propose pour cela la partie méta-schéma. Cependant, à notre connaissance, OntoDB est la seule BDBO à proposer cette partie. Pour ne pas lier l'implantation de OntoQL à l'architecture OntoDB, nous avons choisi de stocker le modèle d'ontologies utilisé en dehors de la BDBO dans un fichier XML nommé `ontology_model.xml`. Ce fichier contient un noeud `entity` pour chaque entité du modèle d'ontologies utilisé par OntoQL. Chaque noeud `entity` a un sous-noeud pour chaque attribut défini sur cette entité. Il existe trois sortes de noeuds pour les attributs selon que leur codomaine soit un type primitif (`attributePrimitive`), référence (`attributeRef`) ou collection (`attributeCollection`).

Exemple. L'exemple suivant présente la représentation de l'entité `#Property` dans le fichier `ontology_model.xml`.

```
<entity name="Property" nameFR="Propriété" superEntity="Concept"/>
  <attributeRef name="scope" nameFR="domaine" entity="Class"/>
  <attributeRef name="range" nameFR="codomaine" entity="Datatype"/>
</entity>
```

Explication. L'entité `#Property` est représentée par le noeud `entity` englobant. Cette entité est définie par un nom par défaut (`name`), des noms dans différentes langues naturelles (ici, seul le nom français, `nameFR`, est indiqué) et son éventuelle super-entité (`superEntity`). Les attributs `#scope` et `#range` sont définis sur l'entité `#Property`. Ces attributs sont représentés par des noeuds imbriqués dans le noeud `entity`. Puisque ces attributs sont de type référence, ils sont définis par des noeuds de type `attributeRef`. Ils sont décrits par un nom par défaut, des noms dans différentes langues naturelles et l'entité qu'ils référencent (`entity`).

Pour pouvoir implanter le langage OntoQL sur OntoDB, il est également nécessaire de stocker les correspondances entre les entités du modèle noyau et celles du modèle PLIB présentées dans le tableau 6.2. En effet pour rechercher, par exemple, les propriétés d'une ontologie, il est nécessaire de savoir qu'elles se trouvent dans la table `Property_det_e` résultant de l'entité PLIB `Property_det` qui correspond à l'entité `#Property` du modèle noyau de OntoQL. Nous avons choisi de représenter ces correspondances en complétant le fichier `ontology_model.xml`. Ainsi, le mapping de chaque entité et attribut déclarés dans ce fichier est indiqué explicitement.

Exemple. L'exemple suivant complète le précédent en présentant la représentation du mapping de l'entité #Property dans le fichier ontology_model.xml.

```
<entity name="Property" nameFR="Propriété" superEntity="Concept"
        entityPLIB="Property_det"/>
  <attributeRef name="range" nameFR="codomaine" entity="Datatype"
                attributePLIB="domain"/>
  <attributeRef name="scope" nameFR="domaine" entity="Class"
                attributePLIB="name_scope" entityPLIB="Property_bsu"/>
  <link withEntityPLIB="Property_bsu"
        attributePLIB="identified_by"/>
</entity>
```

Explication. L'entité #Property correspond à l'entité PLIB Property_det, indiqué par l'attribut entityPLIB du noeud entity ; l'attribut #range correspond à l'attribut PLIB domain de l'entité Property_det, indiqué par l'attribut attributePLIB ; et, l'attribut #scope correspond à l'attribut PLIB name_scope de l'entité Property_bsu, indiqué par les attributs attributePLIB et entityPLIB. Pour pouvoir retourner la valeur de ce dernier attribut, il est nécessaire de connaître le lien d'association qui unit dans le modèle PLIB les entités Property_det et Property_bsu. Ce lien d'association est l'attribut PLIB identified_by. Il est représenté par un noeud link imbriqué dans le noeud entity.

Notons que cette représentation du modèle d'ontologies permet de prendre en compte le cas des constructeurs du modèle PLIB qui ne sont pas présents dans le modèle noyau de OntoQL. En effet, ces constructeurs dont la représentation est déjà créée dans OntoDB peuvent être ajoutés dans le fichier ontology_model.xml en indiquant leur mapping avec les entités PLIB correspondantes. De cette manière, le langage OntoQL considérera ces constructeurs comme si ils avaient été ajoutés explicitement avec les instructions du langage.

Exemple. Ajouter le constructeur des propriétés dépendantes du contexte (Dependent_p_det) de PLIB.

```
<entity name="DependentProperty" nameFR="PropriétéDépendante"
        superEntity="Property" entityPLIB="Dependent_p_det">
  <attributeCollection name="dependsOn" nameFR="dépendDe"
                       type="ref" entity="ContextProperty"
                       attributePLIB="depends_on"/>
</entity>
```

Explication. Le constructeur des propriétés dépendantes du contexte est ajouté comme une entité dont le nom par défaut est DependentProperty, dont celui en français est PropriétéDépendante et qui hérite de l'entité #Property. Pour savoir que ce constructeur est implanté sur OntoDB, l'attribut entityPLIB indique que cette entité correspond à l'entité PLIB Dependent_p_det. Dans PLIB, cette entité a un attribut nommé depends_on qui permet d'indiquer les paramètres de contexte dans lesquels les valeurs de cette propriété sont obtenues. Cet attribut étant de type collection, il est représenté par un noeud attributeCollection. Ce noeud indique que le nom par défaut de cet attribut en OntoQL est dependsOn, que son nom en français est dépendDe, que cet attribut est une collection de référence vers des instances de l'entité #ContextProperty (ici, on suppose que cette entité

a également été ajoutée au modèle noyau de OntoQL) et que cet attribut correspond à l'attribut PLIB `depends_on`.

Nous avons présenté comment nous avons utilisé et adapté la représentation proposée par OntoDB pour implanter le modèle de données du langage OntoQL. La démarche que nous avons suivie peut être appliquée à toute BDBO. Elle consiste à établir les correspondances entre le modèle noyau de OntoQL et le modèle d'ontologies supporté par la BDBO et à les représenter sous la forme d'un fichier XML. Nous montrons maintenant comment nous avons réalisé le traitement d'une requête exprimée dans ce langage.

4 Implantation d'un interpréteur de requêtes OntoQL sur OntoDB

Nous avons réalisé l'implantation du langage OntoQL sur OntoDB dans le langage JAVA en utilisant le générateur de parseur ANTLR[31] pour réaliser l'analyse lexicale et syntaxique de ce langage. Afin de rester concis, nous ne présentons que l'implantation du langage d'interrogation de OntoQL qui constitue l'aspect central de ce langage. Dans la section suivante, nous présentons le processus de traitement d'une requête OntoQL. Nous présentons ensuite plus en détail le traitement d'une requête OntoQL sur les données (cf. section 4.2), sur l'ontologie (cf. section 4.3) et à la fois sur les ontologies et sur les données (cf. section 4.4).

4.1 Traitement d'une requête OntoQL

Pour implanter OntoQL sur OntoDB, nous avons dû choisir une méthode pour réaliser le traitement d'une requête parmi les deux approches suivies pour implanter un langage sur une BDBO. La première approche consiste à traduire une requête du langage source en une requête SQL spécifique à la BDBO. Cette approche a été suivie dans les systèmes RDF-Suite [Alexaki et al., 2001] et RStar [L.Ma et al., 2004]. La seconde approche consiste à traduire une requête du langage source en une suite d'appels d'une API dont chaque méthode retourne un ensemble de données de la BDBO. Cette approche a été suivie dans le système Sesame [Broekstra et al., 2002]. Nous avons donc dû évaluer ces deux méthodes afin d'en choisir une pour implanter le traitement d'une requête OntoQL.

La première solution présente l'avantage de profiter de l'important travail sur l'optimisation des moteurs SQL des bases de données. Elle présente cependant l'inconvénient de dépendre de la BDBO sur laquelle le langage est implanté. La seconde permet la portabilité sur différentes BDBO puisqu'il est possible de fournir une implémentation de l'API pour chaque BDBO. Cependant, dans ce cas, l'optimisation des requêtes est à la charge du moteur du langage implanté. Ces deux méthodes présentent donc chacune leurs avantages et leurs inconvénients.

Souhaitant bénéficier à la fois de l'optimisation des moteurs SQL et de la portabilité de l'implantation sur d'autres BDBO, nous avons choisi d'allier ces deux méthodes. Ainsi, le langage OntoQL est traduit en une requête SQL qui dépend de OntoDB. Mais, cette traduction est réalisée en encapsulant les spécificités de cette BDBO. Ces spécificités sont les suivantes :

[31] `http://www.antlr.org/`

– la représentation choisie pour représenter les données et les ontologies. En effet, les BDBO proposent différentes représentations pour les données et supportent différents modèles d'ontologies ce qui résultent en différents modèles logique pour représenter les ontologies ;
– le SGBD choisi. En effet, les BDBO sont implantées sur différents SGBD qui implantent plus ou moins la norme SQL.

Pour encapsuler la représentation choisie, la traduction d'une requête OntoQL en SQL est réalisée en appelant des méthodes d'interfaces JAVA. Ces interfaces, implantées pour OntoDB, peuvent l'être pour une autre BDBO. Pour encapsuler le SGBD choisi, une requête OntoQL est transformée en une expression de l'algèbre relationnelle avant d'être traduite en SQL. Ainsi, un changement de SGBD ne nécessite que de remplacer la traduction d'une expression de l'algèbre relationnelle dans le langage SQL supporté par le SGBD.

En suivant cette approche, le traitement d'une requête OntoQL suit sept étapes conformément à la figure 6.6. Ces étapes sont les suivantes :

❶ génération de l'expression algébrique *OntoAlgebra* correspondant à la requête OntoQL. Cette génération implante ainsi les règles de traduction présentées au chapitre 5 dans le tableau 5.1 ;

❷ optimisation de l'expression algébrique *OntoAlgebra* en appliquant les techniques d'optimisation présentées au chapitre 5, section 4.2 ;

❸ transformation de l'arbre algébrique *OntoAlgebra* en un arbre algébrique utilisant des opérateurs de l'algèbre relationnelle étendue avec les opérateurs disponibles dans les SGBD relationnels-objets. Cette étape est réalisée en utilisant des règles permettant de traduire les opérateurs de *OntoAlgebra* en une expression de l'algèbre relationnelle en fonction de la représentation des données et des ontologies de OntoDB. Ces règles sont présentées dans les sections suivantes ;

❹ optimisation de l'expression de l'algèbre relationnelle en appliquant les règles d'équivalence définies pour cette algèbre. Cette étape permet de faire des optimisations qui ne sont pas réalisées par le SGBD comme par exemple simplifier une requête imbriquée ;

❺ transformation de l'expression de l'algèbre relationnelle en une requête SQL conforme au SGBD PostgreSQL sur lequel OntoDB est implanté ;

❻ exécution de la requête SQL sur OntoDB en utilisant le JBDC. Le résultat retourné est ainsi un `ResultSet` ;

❼ transformation du `ResultSet` pour retourner le résultat de la requête OntoQL. Ce résultat est une instance de la classe `OntoQLResultSet` qui fait partie de l'API JOBDBC (présentée à la section 5.3).

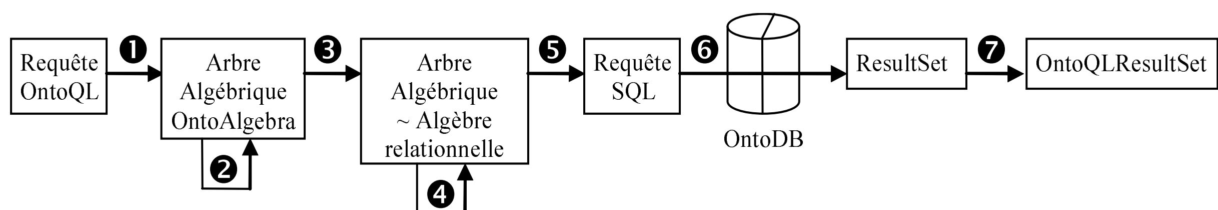


FIG. 6.6 – Les différentes étapes du traitement d'une requête OntoQL

Exemple. La figure 6.7 illustre le traitement de la requête OntoQL `SELECT DEREF(oid) FROM Post` qui permet de retrouver l'ensemble des instances de la classe `Post`.

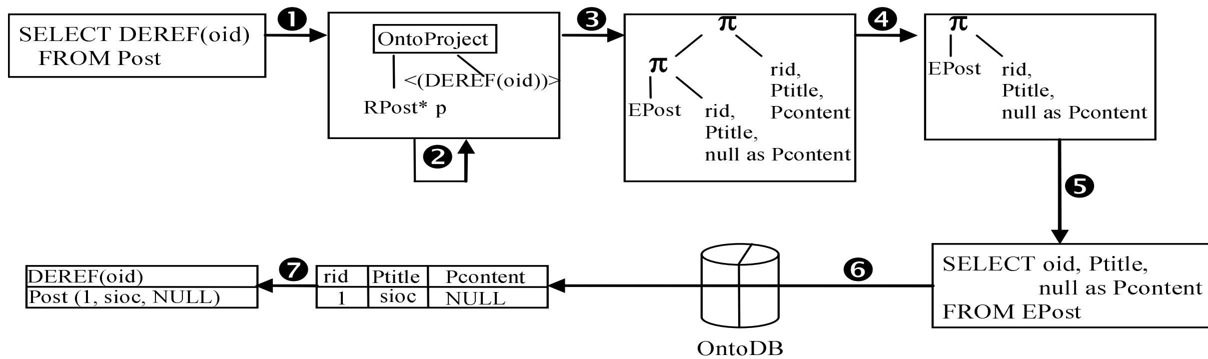


Fig. 6.7 – Exemple de traitement d'une requête OntoQL

Explication. A l'étape ❶, cette requête est passée en entrée de l'analyseur lexical et syntaxique de OntoQL. Le résultat est un arbre algébrique utilisant l'opérateur `OntoProject` de l'algèbre *OntoAlgebra*. Aucune règle d'équivalence ne permet d'optimiser cette expression. Aucun traitement n'est donc réalisé à l'étape ❷. A l'étape ❸, cet arbre est transformé en une expression de l'algèbre relationnelle. Comme nous le verrons ultérieurement, l'expression `RPost`* est traduite par une projection de la table représentant la classe `Post` (`EPost`) sur l'ensemble des colonnes correspondant aux propriétés applicables sur cette classe[32]. Pour celles qui ne sont pas représentées dans cette table (dans cet exemple, la propriété `content`), la valeur `NULL` est retournée. L'opérateur `OntoProject` applique la fonction `DEREF` à l'expression `RPost`*. Il est traduit par une projection de la classe `Post` sur l'ensemble des colonnes correspondantes à ses propriétés applicables (`rid`, `Ptitle` et `Pcontent`). L'expression de l'algèbre relationnelle, qui résulte de cette traduction, comporte deux projections en cascade. Elles peuvent être combinées en une seule projection en appliquant une des règles d'équivalence établies sur les projections [Ullman, 1980]. Ce traitement est réalisé à l'étape ❹. Cette expression algébrique est ensuite traduite en SQL (étape ❺) puis exécutée sur OntoDB (étape ❻). Le résultat est un `ResultSet` comportant trois colonnes. A l'étape ❼, ce `ResultSet` est transformé en un `OntoQLResultSet` ne comportant plus qu'une colonne correspondant à la projection impliquant l'expression `DEREF(oid)`. Cette colonne contient des instances de la classe `Post`. En JAVA, elles sont représentées comme des instances d'une classe de l'API OntoAPI présentée à la section 5.4.

La principale étape du traitement d'une requête OntoQL est ainsi de la traduire en une requête SQL. Cette étape est décrite dans les sections suivantes.

4.2 Traduction des requêtes OntoQL sur les données

La traduction d'une requête OntoQL consiste d'une part à traduire les différents opérateurs de *OntoAlgebra* (cf. section 4.2.1) et d'autre part à traduire les différentes fonctions qui peuvent être appelées par ces opérateurs (cf. section 4.2.2).

[32]pour simplifier, nous nous limitons aux propriétés `rid`, `title` et `content`.

4.2.1 Traduction des opérateurs OntoAlgebra sur les données

Le tableau 6.3 présente les principales règles de traduction d'une expression *OntoAlgebra* en une expression de l'algèbre relationnelle (étendue avec certains opérateurs relationnels-objets) selon la représentation des données proposées par OntoDB[33]. Dans ces expressions algébriques, conformément aux notations utilisées dans [Ullman et al., 2001], les symboles π, σ, $\times$, $\bowtie$, γ, τ, δ et $\cup$ désignent respectivement les opérateurs de projection, sélection, produit cartésien, jointure, agrégat, tri, suppression des doublons et union de l'algèbre relationnelle. C désigne une classe, $p_1, \ldots, p_n$ désignent des propriétés applicables sur la classe C. Parmi ces propriétés, seules $p_1, \ldots, p_u$ sont utilisées pour en décrire les instances. p_1 est une propriété de type collection de références, p_2 est une propriété de type référence et les autres propriétés sont de type primitif.

	Expression OntoAlgebra	Expression de l'algèbre relationnelle étendue
1	$\texttt{OntoProject}\,(C, ext(C), \{(p_1, p_1), \ldots, (p_n, p_n)\}$	$\pi_{Pp_1_rids, Pp_2_rid, Pp_3, \ldots, Pp_u, NULL \rightarrow Pp_{u+1}, \ldots, NULL \rightarrow Pp_n}(EC)$
2	$\texttt{OntoProject}\,(C, ext^*(C), \{(p_1, p_1), \ldots, (p_n, p_n)\}$	$\underline{\texttt{OntoProject}\,(C, ext(C), \{(p_1, p_1), \ldots, (p_n, p_n)\})} \cup$ $\underline{\texttt{OntoProject}\,(C_1, ext^*(C_1), \{(p_1, p_1), \ldots, (p_n, p_n)\})} \cup$ $\cdots \cup$ $\underline{\texttt{OntoProject}\,(C_n, ext^*(C_n), \{(p_1, p_1), \ldots, (p_n, p_n)\})}$
3	$\texttt{OntoOJoin}\,(\underline{exp_1}, \underline{exp_2}, true)$	$\underline{exp_1} \times \underline{exp_2}$
4	$\texttt{OntoOJoin}\,(\underline{exp_1}, \underline{exp_2}, pred)$	$\underline{exp_1} \bowtie_{pred} \underline{exp_2}$
5	$\texttt{OntoUnNest}\,(\underline{exp}, p_1, p)$	$\underline{exp} \bowtie_{EC_1.p = ANY(\underline{exp}.Pp_1_rids)} \underline{\texttt{OntoProject}\,(C_1, ext^*(C_1), \{(p, oid)\})}$
6	$\texttt{OntoProject}\,(\underline{exp}, \{(p_1, p_1), \ldots, (p_n, p_n)\})$	$\pi_{Pp_1_rids, Pp_2_rid, Pp_3, \ldots, Pp_n}(\underline{exp})$
7	$\texttt{OntoDupEliminate}\,(\underline{exp})$	$\delta(\underline{exp})$
8	$\texttt{OntoSelect}\,(\underline{exp}, pred)$	$\sigma_{pred}(\underline{exp})$
9	$\texttt{OntoNest}\,(\underline{exp}, \{p_{k+1}, \ldots, p_n\})$	$\gamma_{Pp_1_rids, Pp_2_rid, Pp_3, \ldots, p_k}(\underline{exp})$
10	$\texttt{OntoSort}\,(\underline{exp}, \{p_1, \ldots, p_n\})$	$\tau_{Pp_1_rids, Pp_2_rid, Pp_3, \ldots, Pp_n}(\underline{exp})$

TAB. 6.3 – Extrait des règles permettant de traduire une requête OntoQL sur les données en une requête SQL

La description de ces règles est la suivante :

– la règle 1 permet de calculer l'extension associée à une classe C qui ne comprend que ses instances directes. Dans *OntoAlgebra*, cette extension est calculée en appliquant les propriétés applicables aux instances de la classe C. L'opération correspondante sur OntoDB est la projection de la table (préfixée par E) qui correspond à cette classe sur les colonnes (préfixées par P) qui correspondent à ces propriétés . Cependant, les propriétés qui ne sont pas utilisées pour décrire les instances de C ne sont pas représentées dans cette table. La valeur NULL est donc retournée pour chacune de ces propriétés. La colonne qui résulte de cette projection est nommée (symbole →) conformément à la

[33] L'ensemble des règles sont données dans l'annexe D.

représentation des propriétés dans OntoDB afin que les autres opérations puissent l'utiliser comme si elle était représentée. Ainsi, conformément à la sémantique de *OntoAlgebra*, une projection impliquant une propriété qui n'est pas utilisée pour décrire une instance retourne la valeur `NULL` ;

– la règle 2 permet de calculer l'extension en profondeur de la classe `C`. Cette extension est calculée en appliquant les propriétés applicables de `C`, non seulement sur les instances de cette classe, mais aussi, sur celles de ses sous-classes directes et indirectes. Cette opération est réalisée récursivement dans OntoDB en réalisant l'union de l'extension de la classe `C` avec les extensions en profondeur des ces sous-classes directes par rapport aux propriétés applicables sur `C` ;

– les règles 3 et 4 permettent de traduire l'opérateur `OntoOJoin` en un produit cartésien ou une jointure relationnelle suivant le prédicat pris en paramètre ;

– La règle 5 permet de traduire l'opérateur `OntoUnNest` lorsqu'il est appliqué sur une propriété p_1 dont le codomaine est une collection de références portant sur la classe C_1. Cette traduction consiste à réaliser la jointure entre l'extension en profondeur de la classe C_1 et l'expression en paramètre de l'opérateur `OntoUnNest`. La condition de jointure est que l'identifiant d'une instance de la classe C_1 doit appartenir à la liste des références contenues dans la colonne Pp_1_rids d'une instance de `exp`. Cette transformation ne peut pas être appliquée pour les propriétés de type collection de type simple. Actuellement, nous n'avons pas trouvé de solution pour réaliser cette traduction. La proposition à venir de l'opérateur `UnNest` dans le SGBD PostgreSQL devrait permettre de résoudre ce problème ;

– les règles 6 à 10 permettent de traduire les autres opérateurs de *OntoAlgebra* dans leur équivalent de l'algèbre relationnelle. Par exemple, la règle 9 indique que l'opérateur `OntoNest` appliqué aux propriétés $p_{k+1}, \ldots, p_n$ est traduit par l'application de l'opérateur d'agrégat (`GROUP BY`) aux attributs correspondant aux propriétés $p_1, \ldots, p_k$. Ce changement de paramètres est nécessaire car l'opérateur `OntoNest` prend en paramètre les propriétés groupées et non pas les propriétés de partitionnement comme c'est le cas pour l'opérateur d'agrégat.

Exemple. La traduction de l'expression `RItem`* est présentée sur la figure 6.8.

Explication. Selon la règle 2, l'expression `RItem`* est traduite en l'union de l'extension de la classe `Item` avec la projection de l'extension en profondeur de sa sous-classe `Post` sur les propriétés `name`, `has_reply` et `has_container`. Cette expression est représentée sur la figure 6.8 (a). La règle 1 est ensuite appliquée pour traduire l'expression `RItem`. Cet exemple suppose que seule la propriété `name` est utilisée pour décrire les instances de la classe `Item`. Ainsi, l'expression `RItem` est traduite par la projection de la table `EItem` associée à la classe `Item` sur la colonne `Pname` qui correspond à cette propriété. Pour les autres propriétés la valeur `NULL` est retournée pour chaque tuple de cette table. Les colonnes correspondantes sont nommées conformément à la représentation proposée par OntoDB. Ainsi, les valeurs `NULL` retournées pour la propriété `has_reply` sont contenues dans une colonne nommée `Phas_reply_rid` puisque cette propriété est de type référence. Le résultat de l'application de cette règle est présentée sur la figure 6.8 (b). La règle 2 est ensuite utilisée pour traduire l'application de l'opérateur `OntoProject` sur la classe `Post`. Cette classe n'ayant pas de sous classe, cette règle consiste seulement à remplacer l'appel de la fonction `ext`* par l'appel de `ext` comme le montre la figure 6.8 (c). Enfin, la règle 1 est appliquée pour traduire l'opérateur `Onto-Project` en une projection sachant que parmi les trois propriétés applicables sur `Item`, seules les

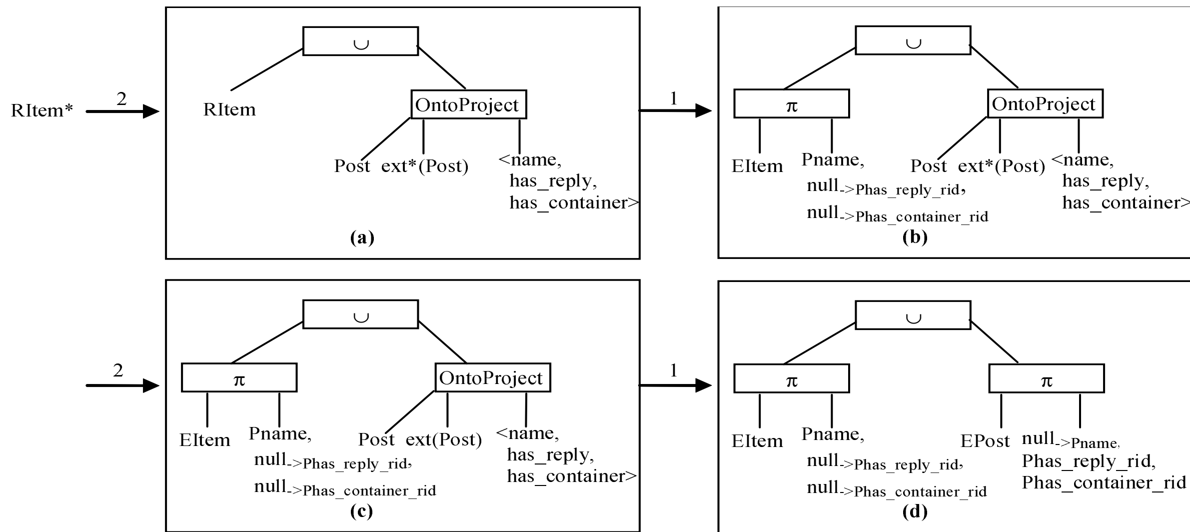


FIG. 6.8 – Exemple de traduction de l'opérateur `OntoProject`

propriétés `has_reply` et `has_container` sont utilisées pour décrire les instances de la classe `Post`. L'expression finale est présentée sur la figure 6.8 (d).

Cet exemple montre comment les règles 1 et 2 permettent de traduire les expressions de projection de *OntoAlgebra*. L'exemple suivant montre l'application de la règle 5 pour traduire l'opérateur `OntoUnNest`.

Exemple. La traduction de l'expression `OntoUnNest(Ruser, subscriber_of, f)` est présentée sur la figure 6.9.

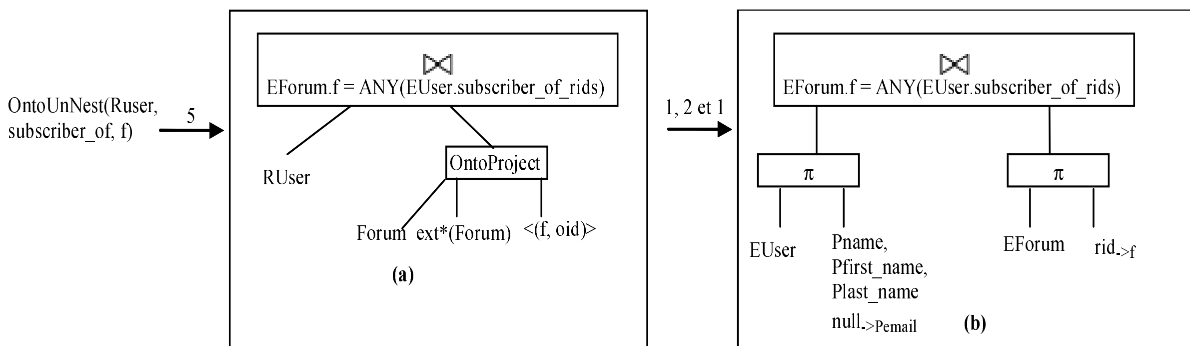


FIG. 6.9 – Exemple de traduction de l'opérateur `OntoUnnest`

Explication. L'application de la règle 5 à l'expression `OntoUnNest(Ruser, subscriber_of, f)` consiste à réaliser la jointure entre l'expression `RUser` et la projection du codomaine de la propriété `subscriber_of`, c'est-à-dire la classe `Forum`, sur la propriété `oid`, renommée `f`. Cette expression est représentée sur la figure 6.9 (a). La traduction des opérateurs `OntoProject` dans cette expression est réalisée comme dans l'exemple précédent, en appliquant les règles 1 et 2. L'expression finale est présentée sur la figure 6.8 (b).

Les règles définies dans cette section permettent de traduire les opérateurs de *OntoAlgebra*. Ces opérateurs peuvent prendre en paramètre des fonctions dont la traduction est donnée dans la section suivante.

4.2.2 Traduction des fonctions

La plupart des fonctions qui peuvent être appelées par les opérateurs de *OntoAlgebra* sont implantées sur PostgreSQL. C'est notamment le cas de la fonction `case()`, des fonctions sur les tableaux (`in()`, `array()`, etc.) et sur les autres types de données (`upper()`, `coalesce()`, etc.). La traduction de ces fonctions est donc directe. Lorsqu'elles prennent en paramètre le nom d'une classe (et/ou d'une propriétés), ce nom est traduit par le nom de la table (et/ou de la colonne) correspondante. Par exemple, l'appel de la fonction `upper(p)`, où `p` est une propriété utilisée de type primitif, sera traduit par l'appel `upper(Pp)`. Ce n'est par contre pas le cas des fonctions `is_of()`, `treat()` et `deref()` qui ne sont pas implantées sur PostgreSQL. Nous avons donc défini une traduction alternative pour ces fonctions.

L'appel de la fonction `is_of(v, T)` retourne vrai si et seulement si `v` est une instance du type `T`. Deux cas peuvent se présenter. `v` peut être un identifiant d'une instance, `T` est alors un type référence portant sur un type T'. Dans ce cas, l'expression `is_of(v, T)` peut être remplacée par l'appel de la fonction `in(v, SELECT oid FROM T')`. En effet, cette fonction teste si l'identifiant `v` appartient à l'ensemble des identifiants des instances de type `T'` en utilisant une requête imbriquée. `v` peut également être une instance, `T` est alors une classe. Dans ce cas, l'expression `is_of(v, T)` peut être remplacée par l'appel de la fonction `in(v.oid, SELECT oid FROM T)`.

Exemple. L'exemple suivant montre comment une requête OntoQL utilisant l'opérateur `IS OF` peut être remplacé par une requête n'utilisant pas cet opérateur.

```
SELECT i.name                        SELECT i.name
  FROM Item AS i               <=>     FROM Item AS i
 WHERE i.oid IS OF (REF(Post))        WHERE i.oid IN (SELECT oid FROM Post)
      (a)                                             (b)
```

Explication. Dans la requête `(a)`, l'opérateur `IS OF` est utilisé pour tester si l'identifiant de l'instance `i` (`i.oid`) appartient au type référence construit à partir de la classe `Post` (`REF(Post)`). Cette requête peut être remplacée par la requête `(b)` qui utilise l'opérateur `IN` à la place de l'opérateur `IS OF` pour retrouver, grâce à une requête imbriquée, l'ensemble des valeurs du type référence `REF(Post)`.

La fonction `is_of` est surchargée pour permettre de tester si une instance appartient à un des types passés en paramètre à cette fonction. Quel que soit le nombre de types pris en paramètre de cette fonction, elle peut également être remplacée par l'appel de la fonction `in` avec une requête imbriquée utilisant l'opérateur `UNION`. Par exemple, si `v` est une instance et T_1 et T_2 sont des classes, alors, l'appel de la fonction `is_of`(v, T_1, T_2) peut être remplacé par l'appel à la fonction `in(v, SELECT oid FROM` T_1 `UNION SELECT oid FROM` T_2`)`.

L'appel de la fonction `treat`(v, T_r) retourne une instance v_r de type T_r à partir de l'instance `v` dont le type déclaré dans la requête est `T`. L'appel de cette fonction peut être remplacé en réalisant une jointure entre les types `T` et T_r sur les identifiants de leurs instances et en retournant les instances de T_r introduites ou leur identifiant si `v` est un identifiant.

Exemple. L'exemple suivant complète le précédent en montrant comment l'opérateur `TREAT AS` peut être remplacé dans une requête OntoQL.

```
SELECT TREAT(DEREF(i.oid)           SELECT p.title
       AS Post).title                 FROM Item AS i INNER JOIN Post AS p
  FROM Item AS i               <=>      ON i.oid = p.oid
 WHERE i.oid IS OF (REF(Post))       WHERE i.oid IN (SELECT oid FROM Post)
       (a)                                  (b)
```

Explication. Dans la requête `(a)`, l'opérateur `TREAT` est utilisé pour retourner une instance de la classe `Post` à partir de l'instance `i` dont le type déclaré dans la requête est `Item`. Cette requête peut être remplacée par la requête `(b)` qui réalise une jointure interne entre les classes `Item` et `Post`. L'appel de la fonction `TREAT` est remplacé par l'itérateur `p` qui parcourt les instances de la classe `Post`.

L'appel de la fonction `deref(v)` retourne une instance v_r de type `T` lorsque `v` appartient au type `REF[T]`. Pour traduire cet appel de fonction, nous avons distingué deux cas. Lorsque l'opérateur `DEREF` n'est pas utilisé dans une expression de chemin, comme nous l'avons montré sur l'exemple présenté en figure 6.7, il est remplacé par la projection de `T` sur l'ensemble des propriétés applicables. Lorsqu'il est utilisé dans une expression de chemin construite à partir d'un type T_p, il est remplacé en réalisant une jointure entre les types `T` et T_p.

Exemple. L'exemple suivant montre comment l'opérateur `DEREF` peut être remplacé lorsqu'il est utilisé dans une expression de chemin.

```
SELECT DEREF(p.has_creator).email    SELECT User.email
  FROM Post AS p                 <=>   FROM Post AS p LEFT OUTER JOIN
                                            User AS u ON p.has_creator=u.oid
     (a)                                           (b)
```

Explication. Dans la requête `(a)`, l'opérateur `DEREF` est utilisé explicitement dans l'expression de chemin `p.has_creator.email` construite à partir de la classe `Post`. Le type de `p.has_creator` est `REF[User]`. Cette requête peut donc être remplacée par la requête `(b)` qui réalise une jointure externe gauche entre les classes `Post` et `User`. Une jointure externe gauche est utilisée car une valeur doit être retournée pour chaque message qu'il ait ou non un auteur. L'appel de la fonction `DEREF` est remplacé par l'itérateur `u` qui parcourt les instances de la classe `User`.

4.3 Traduction des requêtes OntoQL sur les ontologies

La représentation des ontologies dans OntoDB est différente de celle des données. Et donc, les règles de traduction des opérateurs de *OntoAlgebra* utilisés dans des requêtes OntoQL portant sur les ontologies sont différentes de celles présentées pour les requêtes portant sur les données (cf. section 4.2.1). Par contre, les fonctions qui peuvent être appelées par ces opérateurs peuvent être traduites comme lorsqu'elles sont appelées dans une requête sur les données (cf. section 4.2.2). En conséquence, nous ne présentons dans cette section que les règles de traduction des opérateurs de *OntoAlgebra*.

Le tableau 6.4 présente les règles qui permettent de traduire les opérateurs `OntoProject` et `OntoUnNest` de *OntoAlgebra* en une expression de l'algèbre relationnelle selon la représentation des ontologies

proposée par OntoDB. Les règles de traduction pour les autres opérateurs peuvent se déduire de celles-ci (cf. Annexe D, section 1). Dans ces expressions, #E est une entité du modèle noyau de OntoQL qui correspond à l'entité PLIB E. $\#a_1, \ldots, \#a_n$ sont des attributs définis sur #E qui correspondent aux attributs PLIB $a_1, \ldots, a_n$. Ces attributs PLIB sont tous définis sur l'entité E à l'exception de l'attribut a_4 qui est défini sur l'entité E_{def} liée à l'entité E par l'attribut (association) a_{lien}. L'attribut a_2 est un attribut de type référence, a_3 est de type collection de références et les autres attributs sont de type primitif.

	Expression OntoAlgebra	**Expression de l'algèbre relationnelle étendue**
1	OntoProject $(\#E, ext(\#E), \{(\#a_1, \#a_1), \ldots, (\#a_n, \#a_n)\})$	ONLY(E_e)
2	OntoProject $(\#E, ext^*(\#E), \{(\#a_1, \#a_1), \ldots, (\#a_n, \#a_n)\})$	E_e
3	OntoProject $(\underline{exp}, \{(\#a_1, \#a_1)\})$	$\pi_{a_1}(\ \underline{exp}\ )$
4	OntoProject $(\underline{exp}, \{(\#a_2, \#a_2)\})$	$\pi_{rid_s}(\ \underline{exp} \underset{a_2=rid}{\bowtie} E_2_a_2\ )$
5	OntoProject $(\underline{exp}, \{(\#a_3, \#a_3)\})$	$\pi_{Array(\pi_{rid_s}(\sigma_{rid=ANY(\underline{exp}.a_3)}(E_2_a_3)))}(\underline{exp})$
6	OntoProject $(\underline{exp}, \{(\#a_4, \#a_4)\})$	$\pi_{a_4}\ (\underline{exp} \underset{a_{lien}=rid}{\bowtie} E_2_a_{lien} \underset{rid_s=rid}{\bowtie} E_{def}_e)$
7	OntoUnNest $(\underline{exp}, \#a_3, a)$	$\pi_{rid_s \rightarrow a}\ (\ E_2_a_3 \underset{rid=ANY(a_3)}{\bowtie} \underline{exp}\ )$

TAB. 6.4 – Extrait des règles permettant de traduire une requête OntoQL sur les ontologies en une requête SQL

La description de ces règles est la suivante :

- les règles 1 et 2 définissent comment l'extension directe ou en profondeur d'une entité est traduite. L'extension en profondeur d'une entité est directement traduite en SQL par le nom de la table dans laquelle se trouvent ses instances (suffixe _e) puisque l'héritage de tables de PostgreSQL est utilisé dans la représentation proposée par OntoDB. Pour n'obtenir que son extension directe, il est nécessaire d'utiliser le mot clé ONLY ;
- la règle 3 présente le cas le plus simple de la traduction de l'opérateur OntoProject c'est-à-dire lorsqu'il est appliqué sur un attribut simple défini sur l'expression pris en paramètre. Dans ce cas, il est traduit par une projection impliquant l'attribut PLIB correspondant ;
- la règle 4 présente le cas d'une projection impliquant un attribut de type référence. La traduction de cette projection nécessite de réaliser une jointure entre exp et la table d'aiguillage correspondant à cet attribut et de projeter cette table sur l'attribut rid_s (identifiant des instances référencées) ;
- la règle 5 présente le cas d'une projection impliquant un attribut de type collection. Sa traduction nécessite d'utiliser le constructeur de collections à partir d'une requête (Array()). La requête imbriquée retourne les valeurs de l'attribut a_3 contenues dans la table d'aiguillage correspondante. Le constructeur Array() retourne ces valeurs comme une collection ;
- la règle 6 présente le cas d'une projection impliquant l'attribut $\#a_4$ correspondant à l'attribut PLIB a_4 qui n'est pas défini sur E mais sur E_{def}. Cette attribut étant défini sur l'entité E_{def}, il est nécessaire que la traduction de cette projection introduise la table correspondante E_{def}_e. Pour cela, une jointure est effectuée entre l'expression initiale exp et cette table en passant par la table d'aiguillage construite grâce au lien d'association a_{lien} ($E_2_a_{lien}$) ;

– la règle 7 présente la traduction de l'opérateur OntoUnNest. Cet opérateur est traduit en réalisant une jointure avec la table d'aiguillage correspondant à l'attribut collection sur lequel il s'applique et en retournant la colonne rid_s de cette table.

Exemple. La traduction de la requête SELECT #code FROM #Property est présentée sur la figure 6.10.

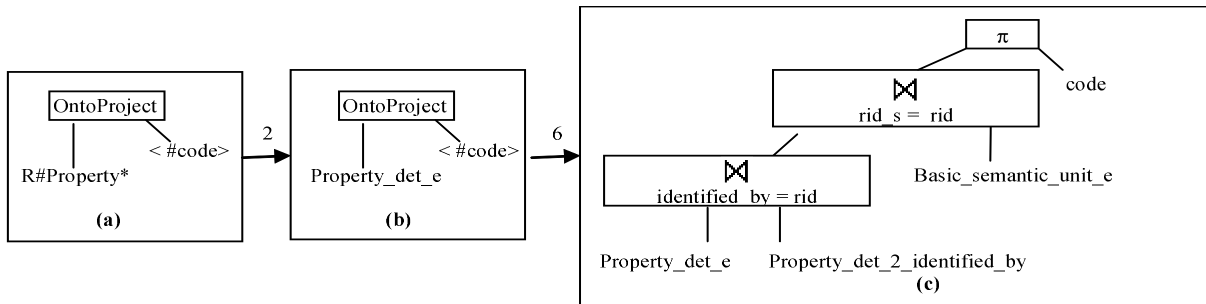


FIG. 6.10 – Exemple de traduction d'une requête sur les ontologies

Explication. La figure 6.10 (a) présente l'expression algébrique de cette requête. Elle consiste à appliquer l'attribut #code aux instances directes et indirectes de l'entité #Property. La règle 2 est appliquée sur cette expression pour traduire l'extension en profondeur de #Property par le nom de la table correspondante dans OntoDB. L'entité OntoQL #Property étant mise en correspondance avec l'entité PLIB Property_det, cette table se nomme Property_det_e. Le résultat de l'application de cette règle est présenté sur la figure 6.10 (b). La règle 6 est ensuite appliquée pour traduire l'opérateur OntoProject. Cette règle doit être appliquée car l'attribut #code correspond à l'attribut PLIB code défini sur l'entité Basic_semantic_unit. Cette entité est liée à l'entité Property_det par le lien d'association identified_by. En suivant la règle 6, la traduction de l'opérateur OntoProject nécessite de réaliser la jointure entre les tables Property_det_e et Basic_semantic_unit_e en passant par la table d'aiguillage construite à partir de l'attribut identified_by (Property_det_2_identified_by). Le résultat de l'application de cette règle, qui est également le résultat final de la traduction, est présenté sur la figure 6.10 (c).

Dans cette section, nous avons présenté les règles permettant de traduire les opérateurs de *OntoAlgebra* lorsqu'ils sont utilisés dans une requête OntoQL sur les ontologies. Notons que ces règles sont valides lorsque les attributs pris en paramètre de ces opérateurs sont représentés dans OntoDB où lorsqu'ils sont dérivés et implantés par une fonction utilisateur dans OntoDB. Dans la section suivante, nous montrons comment nous avons implanté la traduction des requêtes portant à la fois sur les données et sur les ontologies.

4.4 Traduction des requêtes OntoQL sur les ontologies et sur les données

Les requêtes portant à la fois sur les ontologies et sur les données permettent de naviguer dans le sens ontologie vers données ou dans le sens inverse.

4.4.1 Des ontologies vers les données

Pour permettre d'interroger une BDBO dans le sens ontologie vers données, nous avons introduit les deux mécanismes suivants :

- les itérateurs dynamiques introduits par une expression de la forme `Class AS c, c AS i` ;
- l'utilisation de propriétés déterminées à l'exécution d'une requête dans une projection, par une expression de la forme `i.p`.

Dans *OntoAlgebra*, l'introduction des itérateurs dynamiques est réalisée en effectuant une jointure entre les classes (`R#Class* c`) et les instances de la classe racine (`R#ObjectC* i`). La condition de jointure est que l'instance doit appartenir à l'extension directe ou en profondeur de la classe ($i \in$ `ext(c)` ou `ext*(c)`). Dans OntoDB, l'extension de la classe racine est représentée par la table `root_table_extension` dont hérite chaque table associée à chaque classe. Cette table contient ainsi les instances de toutes les classes. Elle comporte la colonne `rid` qui donne l'identifiant interne de ces instances. L'introduction d'itérateurs dynamiques peut donc être réalisée en effectuant la jointure entre la table des classes (`Class_e`) et la table `root_table_extension`. Cependant, la table `root_table_extension` n'indique pas les classes d'appartenance de chaque instance qu'elle contient ce qui est nécessaire pour implanter la condition de jointure. Pour ajouter cette information, une solution possible consiste à modifier la table `root_table_extension`. Cependant, ceci nécessite de modifier les programmes qui permettent de manipuler les données de OntoDB pour qu'ils prennent en compte cette modification. Vu la complexité de cette tâche, nous avons choisi de définir une vue à partir de la table `root_table_extension` qui calcule, pour chaque instance, ses classes d'appartenance. L'inconvénient de ce choix est que le calcul de la vue doit être ajouté au coût de traitement de la requête. Plus précisément, nous avons défini deux vues. La première nommée `Instances` donne pour chaque instance son identifiant interne (colonne `rid`) et sa classe de base (colonne `classid`). Elle est utilisée pour tester si une instance appartient à l'extension directe d'une classe (`c.rid = i.classid`). La seconde nommée `Instances_polymorph` donne pour chaque instance son identifiant interne (colonne `rid`) et ses classes d'appartenance (colonne `classid` de type collection d'entier). Elle est utilisée pour tester si une instance appartient à l'extension en profondeur d'une classe (`c.rid = ANY (i.classid)`). Comme le montre la figure 6.11, les itérateurs dynamiques sont implantés en faisant une jointure entre la table des classes et l'une de ces deux vues.

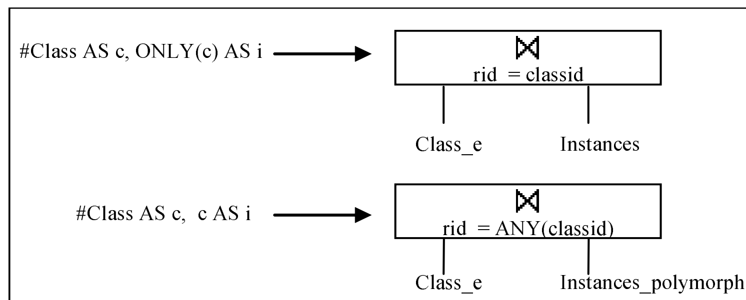


FIG. 6.11 – Implantation des itérateurs dynamiques

Les projections utilisant des propriétés déterminées à l'exécution d'une requête sont réalisées en deux temps. Ce traitement est illustré sur la figure 6.12. Dans un premier temps l'expression `i.p` est

remplacée par les identifiants de la propriété `p`, de l'instance `i` et de la classe de base de l'instance `i`. Dans un second temps, une fois la requête exécutée et donc que la propriété à appliquer est connue, une nouvelle requête OntoQL est exécutée pour rechercher la valeur de l'instance `i` pour cette propriété et le `ResultSet` est modifié en conséquence. Notons que cette traduction nécessite l'exécution d'une requête pour chaque instance retournée ce qui nuit aux performances de la requête globale. Une traduction plus optimale consisterait à n'exécuter qu'une requête par classe retournée.

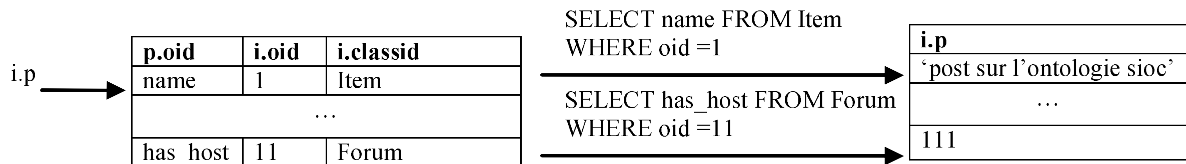


FIG. 6.12 – Implantation des projections impliquant des propriétés déterminées à l'exécution

4.4.2 Des données vers les ontologies

Pour permettre d'interroger les instances des classes et de rechercher, en même temps, la description de ces classes, nous avons introduit la fonction `TYPEOF` qui permet de retourner la classe de base d'une instance. Cette classe étant une instance de l'entité `#Class`, nous avons implanté cet opérateur en ajoutant à la requête une jointure avec cette entité, la condition de jointure étant que l'instance `i` appartienne à l'extension directe de la classe `C` parcourue (`i` ∈ `ext(C)`). Cette implantation est illustrée dans l'exemple suivant.

Exemple. La figure 6.13 montre la traduction de la requête :

```
SELECT TYPEOF(i.oid).#code FROM Item AS i
```

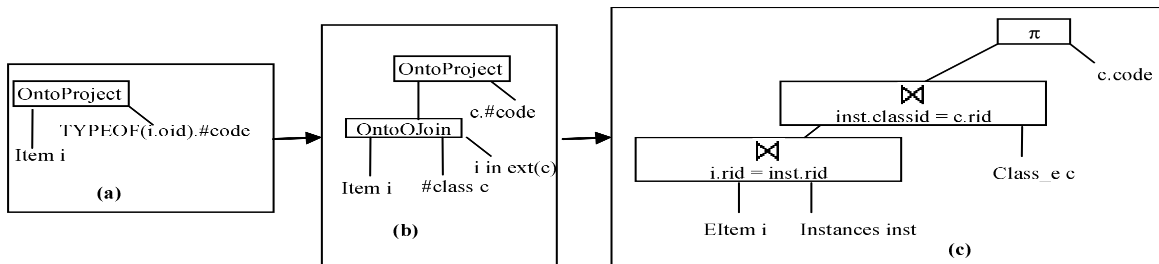


FIG. 6.13 – Exemple montrant l'implantation de l'opérateur `TYPEOF`

Explication. L'expression algébrique de la requête est présentée sur la figure 6.13 (a). Cette expression utilise l'opérateur `OntoProject` qui appelle la fonction `TYPEOF`. L'appel de cette fonction est remplacée par une jointure avec l'entité `#Class` sur la figure 6.13 (b). Pour implanter le prédicat de cette jointure, nous avons noté dans la section précédente qu'il est nécessaire d'utiliser la vue `Instances` qui indique, pour chaque instance, sa classe de base. La traduction de cette expression de *OntoAlgebra* en une expression de l'algèbre relationnelle (cf. figure 6.13 (c)) consiste donc en deux jointures, la première sur la vue `Instances` et la seconde, sur la table `Class_e` contenant les classes.

Dans cette première partie de chapitre, nous avons présenté l'implantation du langage OntoQL sur OntoDB montrant ainsi la faisabilité de l'implantation de ce langage sur une BDBO. Le second objectif de l'implantation du langage OntoQL est de compléter la suite d'outils développés autour de OntoDB pour permettre d'en manipuler les données et les ontologies. Nous présentons les développements que nous avons menés à cette fin dans les sections suivantes.

5 Interfaces usuelles d'un langage d'interrogation de bases de données

Pour faciliter l'utilisation du langage SQL, de très nombreux outils ont été proposés dont certains ont été très largement adoptés. Les utilisateurs du langage SQL sont ainsi habitués à disposer de ces outils. A l'image de SQL et pour faciliter l'adoption du langage OntoQL, nous avons choisi de concevoir des outils similaires pour ce langage. Dans cette section, nous présentons ces outils en montrant comment nous les avons conçus à partir de ceux proposés pour SQL en les adaptant aux particularités du langage OntoQL.

5.1 Interface interactive OntoQLPlus

Tous les SGBD traditionnels proposent un éditeur en ligne de commandes permettant d'exécuter des requêtes SQL . Par exemple, le logiciel *SQL*Plus* permet d'exécuter des requêtes SQL sur une base de données Oracle ; pour SQLServer, le logiciel *isql* peut être utilisé. Pour permettre d'exécuter des requêtes OntoQL sur une BDBO via un éditeur en ligne de commandes, nous avons conçu le logiciel *OntoQLPlus*.

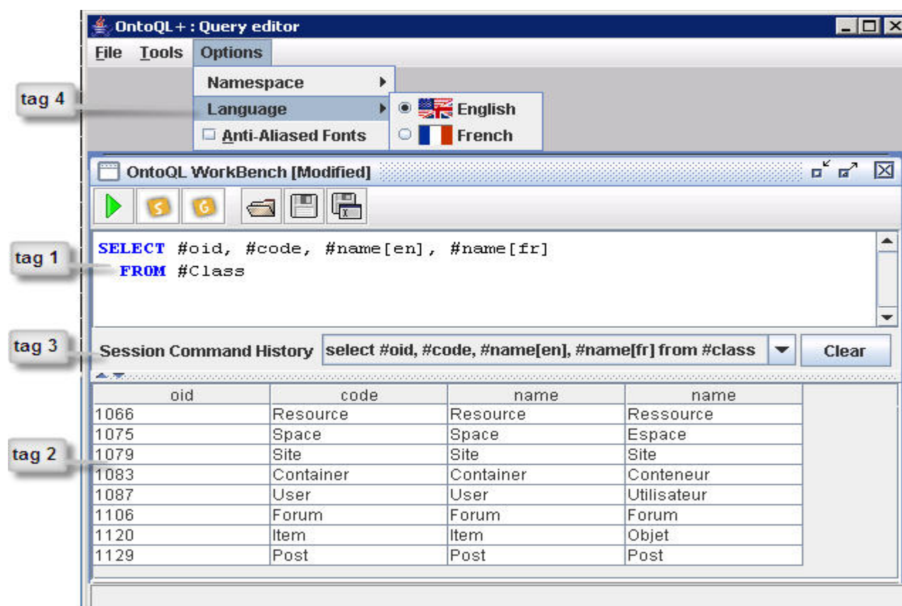


FIG. 6.14 – OntoQL*Plus, un éditeur d'instructions OntoQL en ligne de commandes

Une capture d'écran du logiciel OntoQLPlus est présentée sur la figure 6.14. La zone identifiée par le tag 1 permet de saisir une instruction OntoQL. Une coloration syntaxique de l'instruction saisie est effectuée afin d'en faciliter l'édition. Lorsque cette instruction est exécutée, le résultat s'affiche dans

la zone identifiée par le tag 2 sous la forme d'un tableau. L'historique des instructions exécutées est conservé et présenté dans une liste de sélection (tag 3). Ceci permet de ne pas avoir à saisir à nouveau une requête déjà exécutée. Le langage OntoQL étant paramétré par la langue naturelle dans laquelle les noms des éléments composant une instruction sont donnés et par l'espace de noms dans lequel ces éléments doivent être recherchés, le menu *Options* permet de définir un espace de noms et une langue naturelle par défaut (tag 4).

L'utilisation du logiciel OntoQLPlus nécessite la connaissance de la syntaxe du langage OntoQL. Ce pré-requis est une contrainte forte pour les utilisateurs non avertis souhaitant effectuer des requêtes basiques sur une BDBO. En conséquence, nous avons conçu un éditeur graphique d'instructions OntoQL. Cet éditeur, nommé *OntoQBE* est présenté dans la section suivante.

5.2 Interface graphique OntoQBE

Le langage *Query-By-Example* (QBE) [Zloof, 1977] est un langage graphique pour les bases de données relationnelles. Il a eu une influence importante sur la mise en oeuvre visuelle du langage SQL. Par exemple, le SGBD Access[34] propose une interface graphique pour construire une requête SQL qui reprend les principaux concepts du langage QBE. De plus, QBE est aujourd'hui considéré comme une référence pour la mise en oeuvre visuelle des nouveaux langages d'interrogation comme par exemple ceux pour le modèle XML [Braga et al., 2005]. En conséquence, nous avons choisi de nous baser sur cette proposition pour concevoir l'éditeur graphique de requêtes OntoQL nommé *OntoQBE*.

L'éditeur OntoQBE est une extension du logiciel PLIBEditor[35] qui permet de visualiser et d'éditer des ontologies PLIB. La figure 6.15 présente une capture d'écran de l'interface proposée. Elle illustre les éléments essentiels de QBE. Ainsi, lorsqu'un utilisateur sélectionne une classe dans la hiérarchie (tag 1), une table est construite dans l'interface (tag 2). Cette table contient une colonne pour chaque propriété utilisée pour décrire au moins une instance de cette classe. La ou les tables ainsi définies sont une représentation visuelle de la requête OntoQL construite. Ces tables permettent d'indiquer les projections effectuées dans la requête, les conditions de sélection et la manière selon laquelle le résultat de la requête doit être trié. Lorsqu'elle est exécutée, la requête OntoQL et sa traduction en SQL sont affichées dans deux zones de texte (tag 3 et tag 4). La zone de texte contenant la requête OntoQL est éditable. Ceci permet de modifier la requête exécutée afin, par exemple, d'utiliser des opérateurs qui ne peuvent pas être représentés dans l'interface proposée. Les concepts principaux de QBE ont ainsi été mis en oeuvre pour concevoir l'interface OntoQBE. Nous avons également ajouté des éléments à cette interface pour prendre en compte les particularités du langage OntoQL.

Le langage OntoQL présente la particularité de disposer d'opérateurs orientés-objets. Notamment, il permet d'exprimer des expressions de chemins et des requêtes polymorphes. Pour permettre d'utiliser ces opérateurs dans OntoQBE, nous avons ajouté un menu contextuel (tag 5) qui permet d'indiquer si la requête est polymorphe. De plus, la table construite à partir d'une classe peut avoir une colonne qui correspond à une expression de chemin construite à partir des propriétés de cette classe (tag 6).

Une seconde particularité du langage OntoQL est que les éléments d'une requête exprimée dans ce langage sont associés à une description ontologique qui peut être donnée dans plusieurs langues natu-

[34]`http://office.microsoft.com/en-us/access/`
[35]`http://www.plib.ensma.fr/` (Tools)

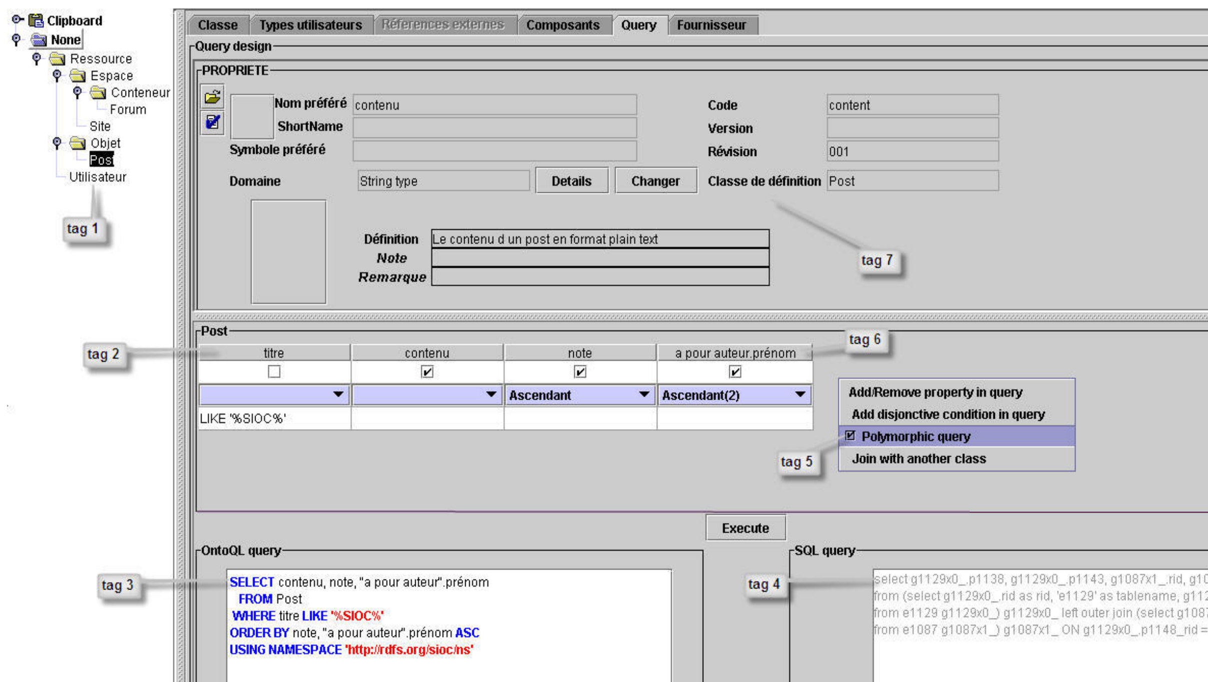


FIG. 6.15 – OntoQBE, un éditeur graphique de requêtes OntoQL

relles. Pour prendre en compte cet aspect, une zone identifiée par le tag 7 permet d'afficher la description des propriétés utilisées dans une requête. Cette description est composée d'un nom, d'une définition, éventuellement d'une illustration, etc. Elle est donnée dans la langue naturelle choisie par l'utilisateur. Elle est traduite dès que cette langue est changée.

Les logiciels OntoQL*Plus et OntoQBE permettent une utilisation interactive du langage OntoQL. Pour permettre de construire un programme informatique manipulant les données d'une BDBO, nous avons conçu des interfaces d'accès au langage OntoQL. L'interface *JOBDBC* permettant d'exécuter des instructions OntoQL depuis le langage JAVA est présentée dans la section suivante.

5.3 Interface JAVA JOBDBC

Pour permettre d'exécuter des instructions SQL dans un programme JAVA, l'API JDBC a été définie. Cette API a été très largement adoptée, la plupart des SGBD proposant une implémentation de cette interface. En conséquence, nous avons choisi de nous baser sur cette API pour définir l'API *JOBDBC* permettant d'exécuter des instructions OntoQL.

Les principales interfaces de l'API JOBDBC sont `OntoQLSession`, `OntoQLStatement` et `OntoQL-ResultSet`. Elles sont présentées sur le diagramme UML de la figure 6.16. L'interface `OntoQLSession` encapsule une connexion JDBC. Les méthodes `setReferenceLanguage` et `setDefaultNamespace` de cette interface permettent respectivement de définir la langue naturelle et l'espace de noms par défaut par rapport auxquels les instructions OntoQL doivent être évaluées. La méthode `createOntoQLStatement` permet de créer une instance de l'interface `OntoQLStatement`. Cette interface permet d'exécuter une instruction OntoQL en proposant des méthodes similaires à celles de l'interface `Statement` du JDBC

(`executeQuery`, `executeUpdate`, etc.). Le résultat d'une requête OntoQL est retourné sous la forme d'un objet de l'interface `OntoQLResultSet`. Cet objet peut être parcouru avec les méthodes `next`, `getInt`, `getString`, etc. comme un objet `ResultSet` de l'API du JDBC.

<<interface>> **OntoQLSession**	<<interface>> **OntoQLStatement**	<<interface>> **OntoQLResultSet**
+setDefaultNamespace(namespace: String) +setReferenceLanguage(lgCode: String) +createOntoQLStatement(): OntoQLStatement	+executeQuery(query: String): OntoQLResultSet +executeUpdate(command: String): int	+next(): boolean +getInt(index: int): int +getString(index: int): String

Fig. 6.16 – Extrait de l'interface JOBDBC

Exemple. Afficher le titre des messages.

```
OntoQLSession session = new OntoQLSessionImpl(connection);
session.setDefaultNamespace("http://rdfs.org/sioc/ns");
session.setReferenceLanguage("FR");
String queryOntoQL = "SELECT titre FROM Message";
OntoQLStatement statement = s.createOntoQLStatement();
OntoQLResultSet resultSet = statement.executeQuery(queryOntoQL);
while (resultSet.next()) {
    System.out.println("Titre du message : " + resultSet.getString(1));
}
```

Explication. La première ligne de ce code source JAVA permet de créer un objet de type `OntoQLSession` à partir de l'objet `connection`, une instance de la classe `Connection` du JDBC. `OntoQLSession` étant une interface, cet objet est créé en utilisant le constructeur de la classe `OntoQLSessionImpl` qui implémente cette interface. L'espace de noms et la langue par défaut sont ensuite définis en utilisant les méthodes `setDefaultNamespace` et `setReferenceLanguage`. Ceci permet d'exécuter une requête OntoQL en français, sans indiquer l'espace de noms utilisé, pour rechercher le titre des messages. Pour exécuter cette requête un objet `OntoQLStatement` est créé. La méthode `executeQuery` de cet objet permet d'exécuter la requête et retourne le résultat sous la forme d'un objet `OntoQLResultSet`. Les lignes obtenues sont parcourues en réalisant une boucle appelant la méthode `next`. Pour chacune de ces lignes le titre du message courant est affiché en utilisant la méthode `getString`.

Le JOBDBC fournit les interfaces de base permettant d'exécuter une instruction OntoQL dans un programme JAVA. Cependant l'utilisation de cette API pour concevoir un programme JAVA présente deux difficultés. D'abord, elle requiert de connaître le langage OntoQL. Par ailleurs, elle requiert de réaliser une traduction entre la représentation relationnelle des données retournées dans un `OntoQLResultSet` et la représentation orientée-objets des données décrite en JAVA. Nous avons donc décidé de concevoir une API nommée *OntoAPI* pour fournir une couche d'accès JAVA aux données d'une BDBO. Nous présentons cette API dans la section suivante et montrons comment elle est intégrée à l'API JOBDBC.

5.4 Interface JAVA OntoAPI

Dans le contexte des bases de données relationnelles, plusieurs solutions ont été proposées pour simplifier la conception de la couche d'accès aux données. Actuellement, une des solutions les plus suivies est d'utiliser un outil de transposition de l'objet vers le relationnel tel que Hibernate[36]. Cet outil permet notamment d'obtenir une API JAVA qui est une représentation objet des données et dont les méthodes chargent automatiquement les données de la base de données. Nous avons conçu l'API *OntoAPI* pour permettre un accès similaire aux ontologies et aux données d'une BDBO.

5.4.1 OntoAPI pour l'accès aux ontologies

Pour créer une API JAVA permettant d'accéder aux ontologies d'une BDBO, les deux solutions suivantes sont possibles :

1. concevoir une API *à liaison différée* (*late binding*), c'est-à-dire indépendante du modèle d'ontologies utilisé. Dans ce cas, l'API contiendrait des méthodes génériques permettant d'obtenir les instances d'une entité du modèle d'ontologies ainsi que leurs valeurs d'attributs en passant en paramètre l'entité et le ou les attributs concernés ;
2. concevoir une API *à liaison préalable* (*early binding*), c'est-à-dire générée à partir du modèle d'ontologies utilisé. Dans ce cas, l'API présenterait une interface JAVA pour chaque entité du modèle d'ontologies et une méthode pour chaque attribut.

La première solution présente l'avantage de ne pas nécessiter de modification de l'API lorsque le modèle d'ontologies évolue tandis que la seconde permet d'obtenir une API beaucoup plus simple à utiliser. Considérant que le modèle d'ontologies utilisé dans une BDBO est relativement stable, nous avons choisi d'implanter la partie de OntoAPI permettant d'accéder aux ontologies en suivant la seconde solution.

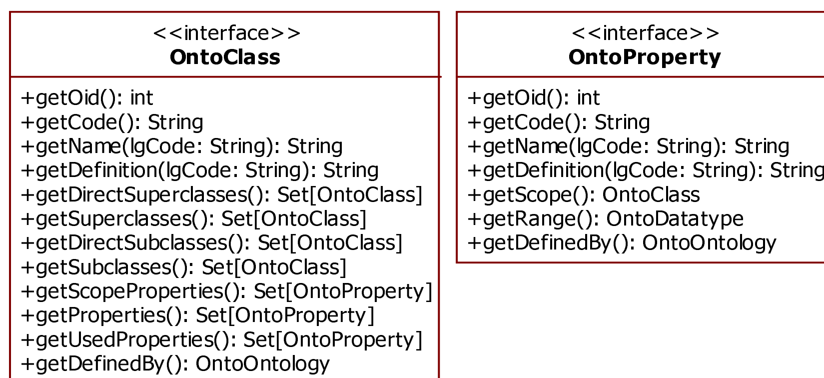


FIG. 6.17 – Extrait de l'interface OntoAPI pour l'accès aux ontologies

Ainsi, OntoAPI est composée d'une interface JAVA pour chaque entité du modèle d'ontologies noyau de OntoQL. Cette interface contient une méthode pour chaque attribut de cette entité. Un extrait de cette API est présenté sur la figure 6.17. OntoAPI contient les interfaces `OntoClass` et `OntoProperty`. Ces interfaces possèdent entre autres les méthodes `getOid`, `getCode`, `getName` et `getDefini-`

[36]`http://www.hibernate.org/`

tion pour obtenir la valeur des attributs correspondants. Lorsque de nouveaux éléments sont ajoutés au modèle noyau, cette API peut être re-générée pour les prendre en compte. Cette API implante le concept de *chargement à la demande* ou *chargement paresseux* proposé dans le framework Hibernate. Le chargement à la demande consiste à ne charger un objet à partir de la base de données que lorsqu'un utilisateur y accède via un accesseur. L'exemple suivant illustre ce concept.

Exemple. Afficher le code et la définition de la classe Post ainsi que le nom des propriétés définies sur cette classe.

```
OntoQLSession session = new OntoQLSession(connection);
OntoClass post = session.newOntoClass("Post");
System.out.println("Code de la classe Post " + post.getCode());
System.out.println("Définition associée à la classe Post "
                   + post.getDefinition());
Set scopeProperties= post.getScopeProperties();
for (int i=0 ; i<scopeProperties.size() ; i++) {
    System.out.println("Nom d'une propriété définie par la classe Post " +
                      ((OntoProperty)scopeProperties.get(i)).getName("FR");
}
```

Explication. La première ligne permet d'obtenir un objet OntoQLSession de l'API JOBDBC. L'appel de la méthode newOntoClass("Post") sur cet objet permet de charger la description de la classe Post dans une instance de l'interface OntoClass. Initialement, seules les valeurs des attributs de type primitif (#oid, #code, #name et #definition) ont été chargées par un accès à la BDBO. Les deux lignes suivantes affichant le code et la définition de la classe Post ne provoquent donc pas d'accès à la base de données. Par contre, lorsque la méthode getScopeProperties est appelée, les identifiants des propriétés définies par la classe Post sont automatiquement chargés dans des objets de type OntoProperty. La boucle réalisée ensuite permet d'afficher le nom de chacune de ces propriétés en appelant la méthode getName des objets OntoProperty correspondants. Le nom des propriétés n'étant pas chargé, chaque appel à cette méthode provoque un accès à la base de données qui charge non seulement le nom de la propriété courante mais aussi ses autres attributs primitifs. De cette façon, les ontologies de la BDBO peuvent être chargées dans les objets de OntoAPI par l'appel de requêtes OntoQL de manière transparente pour l'utilisateur.

Dans cet exemple, nous avons vu que des objets de l'API OntoAPI peuvent être créés en appelant les méthodes newOntoClass, newOntoProperty, etc. d'un objet OntoQLSession. L'appel de cette méthode nécessite de connaître l'identifiant des éléments à charger. Or, dans bien des cas, les identifiants des éléments à charger sont inconnus. C'est notamment le cas lorsque l'on veut afficher la hiérarchie des classes, une opération nécessaire pour concevoir un éditeur d'ontologies. Une solution possible est de commencer par exécuter une requête OntoQL pour obtenir les identifiants des classes puis d'appeler la méthode newOntoClass pour chaque classe du résultat. Afin de simplifier ce traitement, nous avons choisi de lier le langage d'interrogation et l'API OntoAPI. Ce lien est effectué grâce à l'opérateur DEREF de OntoQL comme le montre l'exemple suivant.

Exemple. Afficher le nom en français de la classe Item.

```
OntoQLSession session = new OntoQLSession(connection);
String queryOntoQL = "SELECT DEFEF(c.#oid) FROM #Class AS c
                      WHERE c.#name[EN]='Item'";
OntoQLStatement statement = s.createOntoQLStatement();
OntoQLResultSet resultSet = statement.executeQuery(queryOntoQL);
if (resultSet.next()) {
    OntoClass item = resultSet.getOntoClass(1);
    System.out.println("Le nom en français de la classe Item est"
                      + item.getName("FR"));
}
```

Explication. La requête exécutée retourne la classe dont le nom en anglais est `Item` grâce à l'opérateur `DEREF`. L'objet `OntoQLResultSet` résultat contient une seule colonne dont le type est `OntoClass`. Les objets contenus dans cette colonne peuvent être retournés en appelant la méthode `getOntoClass` de l'interface `OntoQLResultSet`. Cette interface possède ainsi une méthode `get` pour chaque entité du modèle d'ontologies.

5.4.2 OntoAPI pour l'accès aux données

Contrairement au modèle d'ontologies relativement stable, les ontologies d'une BDBO peuvent être régulièrement mises à jour. Afin de ne pas avoir à re-générer régulièrement l'API d'accès aux données, nous avons donc choisi de la concevoir comme une API à liaison différée.

Ainsi, une seule interface JAVA permet d'accéder aux données d'une BDBO. Cette interface, nommée `Instance`, est présentée sur la figure 6.18. Un objet de cette interface est créé en passant l'identifiant de l'instance à charger ainsi que sa classe de base. Les valeurs de propriétés de cette instance peuvent être récupérées en appelant une des méthodes `getIntPropertyValue`, `getStringPropertyValue`, etc. qui prennent en paramètre l'identifiant de la propriété dont on souhaite récupérer la valeur. Comme pour l'accès aux ontologies, cette interface implante le concept de chargement à la demande et est liée au langage de requêtes comme le montre l'exemple suivant.

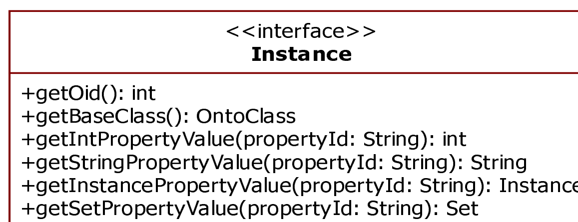


FIG. 6.18 – Extrait de l'interface OntoAPI pour l'accès aux données

Exemple. Afficher le titre des messages.

```
OntoQLSession session = new OntoQLSession(connection);
String queryOntoQL = "SELECT DEFEF(p.oid) FROM Post AS p;
OntoQLStatement statement = s.createOntoQLStatement();
OntoQLResultSet resultSet = statement.executeQuery(queryOntoQL);
```

```
while (resultSet.next()) {
    Instance post = resultSet.getInstance(1);
    System.out.println("Le titre de la classe Item est"
                       + post.getStringPropertyValue("title"));
}
```

Explication. La requête exécutée retourne les instances de la classe `Post` en utilisant l'opérateur `DEREF`. L'objet `OntoQLResultSet` résultat contient une seule colonne dont le type est `Instance`. Les objets contenus dans cette colonne peuvent être retournés en appelant la méthode `getInstance` de l'interface `OntoQLResultSet`. Pour chacun de ces objets, la méthode `getStringProperty-Value("title")` est appelée afin de retourner leur valeur pour la propriété `title`.

OntoAPI permet donc de charger *à la demande* les ontologies et les données d'une BDBO. Le chargement à la demande permet de minimiser les accès à la BDBO. Par contre, actuellement, nous n'avons pas implanté les autres mécanismes proposés par Hibernate pour minimiser ces accès (gestion d'un cache et construction `fetch`).

Les outils et les API présentés dans cette section permettent d'utiliser le langage OntoQL avec des outils proches de ceux proposés pour les bases de données traditionnelles. Dans la section suivante, nous présentons les outils que nous avons conçus pour prendre en compte d'autres particularités des BDBO.

6 Interfaces spécifiquement conçues pour les BDBO

Nous avons conçu deux interfaces spécifiquement pour les BDBO. La première présentée dans la section 6.1 permet de faciliter la recherche des concepts définis dans une ontologie. La seconde présentée dans la section 6.2 permet de faciliter l'interopérabilité avec les systèmes conçus dans le contexte du Web Sémantique.

6.1 Interface par mots clés

Une ontologie conçue selon le modèle en oignon propose une couche linguistique qui associe aux concepts qu'elle définit une description textuelle éventuellement donnée dans plusieurs langues naturelles. Cette description textuelle est fournie pour compléter le plus précisément possible le concept auquel elle est associée. Les mots qu'elle contient sont donc choisis pour caractériser ce concept. Aussi, il serait très utile de pouvoir rechercher un concept d'une ontologie selon les mots contenus dans sa description textuelle. Or, le langage OntoQL et les outils présentés jusqu'à présent ne permettent de rechercher les concepts d'une ontologie qu'en fonction des valeurs de leurs attributs. Ils ne permettent pas de les rechercher en utilisant les techniques développées dans le contexte de la recherche d'information telles que la recherche par mots clés. Nous avons mené un travail préliminaire pour combler ce manque. Le résultat de ce travail est une interface qui permet de rechercher les classes et propriétés d'une ontologie en fonction de mots clés saisis.

L'interface par mots clés est présentée sur la figure 6.19. Elle a été conçue comme une fenêtre modale (tag 1) du logiciel PLIBEditor (tag 2). Cette fenêtre permet de saisir des mots clés dans un champ de texte

puis de rechercher les classes et propriétés dont la description textuelle contient ces mots. La recherche avancée permet de préciser si la recherche doit porter sur les classes et/ou sur les propriétés. Elle permet aussi d'indiquer les attributs constituant la description textuelle des classes et propriétés qui doivent être considérés pour la recherche. Le résultat est affiché comme une liste de noms des classes et propriétés trouvées. Lorsqu'un de ces éléments est sélectionné sa description est automatiquement affichée dans PLIBEditor (tag 3).

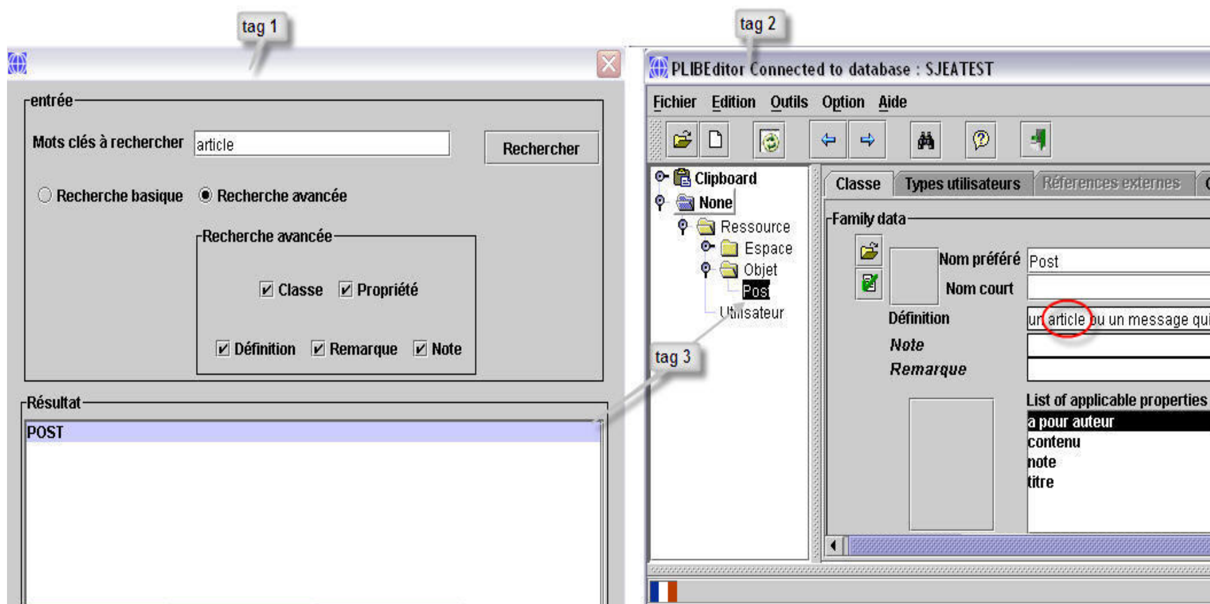


FIG. 6.19 – Interface de recherche de classes et de propriétés par mots clés

Actuellement, la recherche des classes et des propriétés dans cette interface est réalisée en exécutant des requêtes OntoQL utilisant l'opérateur LIKE. Ceci nécessite de parcourir l'ensemble des classes et des propriétés de la BDBO pour retourner les résultats. Pour résoudre ce problème, notre idée est de coupler la BDBO à un moteur de recherche plein texte. Celui-ci indexera les descriptions textuelles associées aux concepts permettant ainsi une recherche plus efficace. De plus, ce moteur de recherche offrira des opérateurs de recherche plein texte beaucoup plus puissants que l'opérateur LIKE comme par exemple les opérateurs de recherches approximatives ou par proximité.

L'interface par mots clés complète les outils conçus autour du langage OntoQL pour permettre de manipuler les données et les ontologies stockées dans la BDBO OntoDB. Cependant, lorsque cette BDBO est utilisée dans le contexte du Web Sémantique, l'utilisation du langage OntoQL pose des problèmes d'interopérabilité avec les systèmes conçus dans ce contexte, la plupart de ces systèmes proposant maintenant une implantation du langage SPARQL qui émerge comme un standard. C'est notamment le cas dans le projet e-WOK_HUB[37], auquel le LISI participe, où l'approche suivie vise à étudier le passage à l'échelle de la BDBO OntoDB conjointement avec le moteur de recherche sémantique CORESE [Corby et al., 2006] qui propose une implantation du langage SPARQL. Pour simplifier, l'interopérabilité de la BDBO OntoDB avec les systèmes développés pour le Web Sémantique, nous avons décidé de fournir une interface SPARQL à cette BDBO.

[37] http://www-sop.inria.fr/acacia/project/ewok/

6.2 Interface avec le langage SPARQL

Plutôt que de réaliser une implantation du langage SPARQL en partant du modèle de données de OntoDB, nous avons choisi de traduire ce langage en des appels aux opérateurs de l'algèbre *OntoAlgebra*. De cette manière, nous bénéficions de l'étude que nous avons menée sur cette algèbre (cf. chapitre 4) ainsi que de son implantation sur la BDBO OntoDB (cf. section 4). Actuellement, nous avons implanté un sous-ensemble de ce langage permettant d'exploiter des ontologies respectant un ensemble d'hypothèses définies ci-dessous. Nous avons défini un modèle pour les requêtes portant sur les données (cf. section 6.2.1) et pour celles portant sur les ontologies (cf. section 6.2.2) qui peuvent être exécutées sur OntoDB.

6.2.1 Requêtes SPARQL sur les données

Une requête sur les données retourne des identifiants d'instances et/ou des valeurs de propriétés de ces instances à partir des classes et des propriétés. La clause `WHERE` d'une requête SPARQL sur les données est ainsi composée de deux types de triplets :

– (?`oid`, `type`, `C`)[38]où `C` est une classe. Ce triplet permet de retrouver les identifiants (`oid`) des instances de la classe `C`.
– (?`oid`, `p`, ?`val`) où `p` est une propriété. Ce triplet permet de retrouver les identifiants des instances (`oid`) qui ont une valeur (`val`) pour la propriété `p`.

Ces triplets peuvent être séparés par les différents opérateurs offerts par SPARQL tels que `OPTIONAL` ou `UNION`. Des conditions peuvent également être définies avec l'opérateur `FILTER`.

6.2.1.1 Définition des requêtes SPARQL sur les données considérées

Jusqu'à présent, nous avons considéré les requêtes SPARQL dont la clause `WHERE` est de la forme suivante :

$(?\mathtt{oid}_1, \mathtt{type}, \mathtt{C}_1)$ [OPTIONAL] $(?\mathtt{oid}_1, \mathtt{p}_{11}, ?\mathtt{val}_{11}) \cdots$ [OPTIONAL] $(?\mathtt{oid}_1, \mathtt{p}_{n1}, ?\mathtt{val}_{n1})$ [FILTER()] θ
$(?\mathtt{oid}_2, \mathtt{type}, \mathtt{C}_2)$ [OPTIONAL] $(?\mathtt{oid}_2, \mathtt{p}_{12}, ?\mathtt{val}_{12}) \cdots$ [OPTIONAL] $(?\mathtt{oid}_2, \mathtt{p}_{n2}, ?\mathtt{val}_{n2})$ [FILTER()] θ
...
$(?\mathtt{oid}_n, \mathtt{type}, \mathtt{C}_n)$ [OPTIONAL] $(?\mathtt{oid}_n, \mathtt{p}_{1n}, ?\mathtt{val}_{1n}) \cdots$ [OPTIONAL] $(?\mathtt{oid}_n, \mathtt{p}_{nn}, ?\mathtt{val}_{nn})$ [FILTER()]

Dans cette écriture, l'opérateur `OPTIONAL` est entre `[]` pour indiquer qu'il n'est pas obligatoire. L'opérateur θ représente l'un des opérateurs SPARQL ('`.`', `OPTIONAL` ou `UNION`).

La traduction des requêtes de cette forme n'est réalisée que lorsque les hypothèses suivantes sont respectées.

Hypothèse 1. Pour chaque triplet `(?oid, p, ?val)`, la propriété `p` doit être définie sur la classe `C` utilisée dans le triplet `(?oid, type, C)`.

Hypothèse 2. Chaque requête doit être typée, c'est-à-dire que pour chaque triplet `(?oid, p, ?val)`, il

[38]Pour simplifier l'écriture des triplets, nous utilisons une écriture entre parenthèses et remplaçons `rdf:type` par `type`.

doit exister un triplet `(?oid, type, C)` qui définit la classe dans laquelle cette instance doit être recherchée.

Hypothèse 3. Le résultat d'une requête SPARQL dépend des triplets présents dans la source de données. La traduction que nous fournissons rend le résultat de requêtes SPARQL lorsque tous les triplets qui peuvent se déduire d'autres triplets sont effectivement représentés.

Pour clarifier cette dernière règle, prenons l'exemple de la requête `SELECT ?oid WHERE {?oid rdf:type User}`. Si dans la source de données, les triplets `(?oid, type, C)` lient chaque instance à sa ou ses classes de base, elle ne retourne que les instances de la classe `User`. Par contre, si les triplets `(?oid, type, C)` lient chaque instance non seulement à sa ou ses classes de base mais également à ses autres classes d'appartenance, alors cette requête retourne les instances de la classe `User` ainsi que ses sous-classes. Notre traduction s'appuie sur cette deuxième hypothèse.

6.2.1.2 Traduction des requêtes qui appliquent l'opérateur OPTIONAL aux propriétés

Chaque bloc $(\texttt{?oid}, \texttt{type}, \texttt{C})$ `OPTIONAL` $(\texttt{?oid}, \texttt{p}_1, \texttt{?val}_1) \cdots$ `OPTIONAL` $(\texttt{?oid}, \texttt{p}_n, \texttt{?val}_n)$ se traduit par la suite d'appels suivants aux opérateurs de *OntoAlgebra* :

$\texttt{Resultat}_1 ::= \texttt{OntoProject}(\texttt{C}, \texttt{ext}^*(\texttt{C}), \{(\texttt{?oid}, \texttt{oid}), (\texttt{?p}_1, \texttt{p}_1), \ldots, (\texttt{?p}_n, \texttt{p}_n)\})$
$\texttt{Resultat}_2 ::= \texttt{OntoUnnest}(\texttt{Resultat}_1, \texttt{p}_1, \texttt{p}_1)$
...
$\texttt{Resultat}_{n+1} ::= \texttt{OntoUnnest}(\texttt{Resultat}_n, \texttt{p}_n, \texttt{p}_n)$

Ces appels correspondent à la requête OntoQL suivante :

```
SELECT oid, p1, ..., pn
  FROM C, UNNEST(C.p1) AS p1, ..., UNNEST(C.pn) AS pn
```

Cette traduction nécessite l'utilisation de l'opérateur `UNNEST` parce que d'une part, chaque propriété RDF est par défaut une collection, et, d'autre part, SPARQL retourne une ligne par valeur d'une propriété alors que OntoQL retourne une seule ligne pour l'ensemble des valeurs d'une propriété de type collection.

Exemple. Retourner les identifiants et adresses email des utilisateurs.

```
SELECT *                                  <=>  SELECT user.oid, user.email
 WHERE { ?user type User .                      FROM User AS user,
         OPTIONAL { ?user email ?email } }           UNNEST(user.email) AS email
```

Explication. La requête SPARQL est présentée à gauche[39]. La requête OntoQL qui correspond à la traduction de la requête SPARQL en appels d'opérateurs de *OntoAlgebra* est présentée à droite. Elle utilise l'opérateur `UNNEST` pour retourner la valeur de la propriété `email` qui peut prendre en RDF plusieurs valeurs.

6.2.1.3 Traduction des requêtes n'appliquant pas l'opérateur OPTIONAL aux propriétés

Si le triplet $(\texttt{?oid}, \texttt{p}_1, \texttt{?val}_1)$ n'est pas précédé de l'opérateur `OPTIONAL`, une instance ne doit être retournée que si elle fournit une valeur pour la propriété $\texttt{p}_1$. En OntoQL, une instance est retournée avec

[39]Pour simplifier, les espaces de noms utilisés ne sont pas précisés.

la valeur NULL si elle ne fournit pas de valeur pour la propriété. En conséquence, la suite d'appels aux opérateurs de *OntoAlgebra* présentée précédemment doit être modifiée de la manière suivante :

$\texttt{Resultat}_2$ ::= $\texttt{OntoSelect(OntoUnnest(Resultat}_1, \texttt{p}_1, \texttt{p}_1), \texttt{p}_1$ IS NOT NULL)

Ce qui correspond à la requête OntoQL suivante :

```
SELECT oid, p1, ..., pn
  FROM C, UNNEST(C.p1) AS p1, ..., UNNEST(C.pn) AS pn
 WHERE p1 IS NOT NULL
```

Exemple. Retourner les identifiants et adresses email des utilisateurs dont l'adresse email est connue.

```
SELECT *                              <=>    SELECT user.oid, user.email
 WHERE { ?user type User .                    FROM User AS user,
         ?user email ?email }                      UNNEST(user.email) AS email
                                              WHERE user.email IS NOT NULL
```

Explication. Cet exemple est similaire au précédent à part que dans ce cas le prédicat email IS NOT NULL est ajouté pour ne pas retourner les utilisateurs dont l'adresse email est inconnue.

6.2.1.4 Traduction des requêtes utilisant l'opérateur FILTER

SPARQL permet de définir des expressions évaluables sur les variables grâce à l'opérateur FILTER. Les variables sont typées par les types XMLSchema string, integer, boolean, double, float, decimal et datetime. Le prédicat peut être composé d'opérateurs logiques (NOT, AND, OR), de comparaison (<, ≤, = != ≥, >) ou arithmétiques (+, -, *, /). Ces types de données et ces opérateurs sont fournis par OntoQL. Ainsi, la traduction de l'opérateur FILTER est un appel à l'opérateur OntoSelect de *OntoAlgebra* en conservant les opérateurs définis dans le filtre.

Exemple. Retourner les identifiants des messages qui ont une note supérieure à 3.

```
SELECT ?post                              <=>  SELECT post.oid
 WHERE { ?post type Post .                      FROM Post AS post,
         OPTIONAL { ?post note ?note }               UNNEST(post.note) AS note
         FILTER (?note > 3) }                  WHERE note > 3
```

Explication. L'opérateur > étant fourni par OntoQL, l'opérateur FILTER est traduit par le prédicat de sélection note > 3.

SPARQL définit également des fonctions spécifiques. Le tableau 6.5 décrit ces fonctions et propose éventuellement une traduction en OntoQL. Par exemple, la fonction bound qui permet de savoir si une valeur est la valeur UNBOUND peut être traduite par la condition IS NOT NULL en OntoQL. Actuellement, notre implantation ne permet pas d'utiliser les fonctions qui n'ont pas d'équivalent en OntoQL. Nous prévoyons de les intégrer en définissant des fonctions utilisateurs grâce au langage de programmation associé à la BDBO.

Opérateurs	Description	Traduction en OntoQL
`bound`	Retourne vrai si une variable est liée.	`IS NOT NULL`
`isIRI`	Retourne vrai si la valeur d'une variable est un URI.	
`isBLANK`	Retourne vrai si la valeur d'une variable représente un noeud blanc.	
`isLiteral`	Retourne vrai si la valeur d'une variable est un littéral.	
`str`	Retourne une chaîne de caractères à partir d'une valeur littérale ou URI.	`CAST (AS String)`
`datatype`	Retourne le type d'un littéral.	`typeOf`
`lang`	Retourne la langue naturelle dans laquelle une valeur est donnée.	
`langMatch`	Retourne vrai si deux valeurs sont données dans la même langue naturelle.	
`regex`	Retourne vrai si une valeur est conforme à une expression régulière.	`SIMILAR TO`

Tab. 6.5 – Traduction des fonctions de SPARQL en OntoQL

6.2.1.5 Traduction des requêtes comportant plusieurs classes séparées par l'opérateur UNION

L'opérateur `UNION` de SPARQL est traduit par l'opérateur `OntoUnion` de *OntoAlgebra*. Les noms des variables sont utilisés pour faire correspondre les projections des requêtes impliquées dans l'union.

Exemple. Retourner les identifiants des utilisateurs et des messages.

```
SELECT ?x                    <=>      SELECT oid FROM User
WHERE { { ?x type User }              UNION
         UNION                        SELECT oid FROM Post
        { ?x type Post } }
```

Explication. La variable `?x` correspond à l'application de la propriété `oid`. Les valeurs retournées par cette propriété étant du même type qu'elle soit appliquée sur les instances de la classe `User` ou sur celles de `Post`, l'opérateur `UNION` de SPARQL peut être traduit par l'opérateur `OntoUnion`.

SPARQL permet de faire une union entre des triplets dont certaines variables sont différentes. Pour ces variables, la valeur `NULL` peut être utilisée pour la traduction. Ceci est illustré dans l'exemple suivant.

Exemple. Retourner les identifiants et adresses email des utilisateurs ainsi que les identifiants et titres des messages.

```
SELECT ?x ?email ?title                  <=>     SELECT oid, email, NULL
 WHERE { { ?x type User                            FROM User
           OPTIONAL { ?x email ?email } }         UNION
         UNION                                   SELECT oid, NULL, title
         { ?x type Post                            FROM Post
           OPTIONAL { ?x title ?title } } }
```

Explication. L'opérateur `OntoUnion` de *OntoAlgebra* nécessite que le même nombre de projections soit réalisé dans chaque requête et que les projections correspondantes soient de même type. La valeur `NULL` appartenant à tous les types de données, sa projection permet de remplacer les projections qui ne peuvent pas être réalisées.

6.2.1.6 Traduction des requêtes comportant plusieurs classes séparées par l'opérateur '.'

Deux classes impliquées dans une requête SPARQL sont jointes en utilisant les variables communes aux triplets. Ce cas est donc traduit par l'opérateur `OntoOJoin` de *OntoAlgebra*.

Exemple. Retourner les messages et les utilisateurs qui ont le même identifiant.

```
SELECT ?x                    <=>       SELECT Post.oid
 WHERE { ?x type Post .                  FROM Post, User
         ?x type User }                 WHERE Post.oid = User.oid
```

Explication. Les classes `Post` et `User` sont séparées par l'opérateur '.'. En OntoQL, ceci est traduit par une jointure entre ces deux classes. La condition de jointure est construite à partir des variables identiques dans les triplets correspondants à ces classes. Dans cet exemple, la variable `?x` est commune. Elle correspond à la propriété `oid`. La condition de jointure est donc réalisée sur cette propriété.

Dans [Cyganiak, 2005], l'auteur a montré qu'une jointure en SPARQL a une sémantique différente de celle de SQL (et donc de OntoQL) lorsque la condition de jointure implique des propriétés optionnelles. Il a également proposé une traduction alternative dans ce cas. Dans l'exemple précédent, le problème ne se pose pas car la propriété `oid` est obligatoire. C'est par contre le cas dans l'exemple suivant.

Exemple. Retourner les identifiants et le nom des utilisateurs qui ont le même nom.

```
SELECT ?u1, ?u2, ?last_name         <=>  SELECT u1.oid, u2.oid,
 WHERE { ?u1 type User                          COALESCE(u1.last_name,
         OPTIONAL {                                      u2.last_name)
            ?u1 last_name ?last_name }      FROM User AS u1, User AS u2
         ?u2 type User                     WHERE u1.last_name = u2.last_name
         OPTIONAL {                           OR u1.last_name IS NULL
            ?u2 last_name ?last_name } }      OR u2.last_name IS NULL
```

Explication. Contrairement à la sémantique SQL, la jointure de cet exemple exclut seulement les couples pour lesquels les noms des deux utilisateurs sont connus et différents. Ainsi, si le nom d'un utilisateur n'est pas connu, il sera retourné avec tous les utilisateurs de la base de données. Le nom retourné pour ce couple est celui qui est connu ou `UNBOUND` si les deux sont inconnus.

Pour reproduire cette sémantique, la traduction en OntoQL teste si l'une des deux propriétés impliquées dans la jointure vaut la valeur `NULL`, et la projection impliquant la propriété `last_name` utilise la fonction `COALESCE` pour déterminer le nom qui est connu. En effet, la fonction `COALESCE` prend deux valeurs en paramètre et retourne la valeur non nulle parmi ces deux valeurs ou la valeur `NULL` si les deux valeurs sont nulles.

6.2.1.7 Traduction des requêtes comportant plusieurs classes séparées par l'opérateur OPTIONAL

Lorsque deux classes impliquées dans une requête SPARQL sont jointes en utilisant l'opérateur `OPTIONAL`, toutes les instances de la première classe sont retournées même si elles sont exclues dans la jointure. Les variables des triplets de la classe de gauche sont retournées avec la valeur `UNBOUND`. Cette opération correspond à une jointure externe gauche en SQL et est donc traduite par l'opérateur `OntoLeftOuterOJoin` de *OntoAlgebra*.

Exemple. Retourner les messages avec leur auteur même si il est inconnu.

```
SELECT ?p, ?u                              <=>  SELECT p.oid, u.oid,
 WHERE { ?p type Post                             FROM Post AS p
         OPTIONAL { ?p has_creator ?u }                LEFT JOIN User AS u
         OPTIONAL { ?u type User } }                   ON p.has_creator = u.oid
```

Explication. Dans cette requête une jointure est réalisée entre les classes `Post` et `User`. L'opérateur `OPTIONAL` étant appliquée à la classe `User`, une jointure externe gauche est réalisée. La condition de jointure est déterminée par la variable `?u` qui correspond à la valeur de la propriété `has_creator` pour les messages et à la valeur de la propriété `oid` pour les utilisateurs.

6.2.1.8 Traduction des requêtes non typées

Le modèle des requêtes que nous avons défini impose que les requêtes soient typées, c'est-à-dire que le type des instances recherchées soit précisé. Ceci permet de traduire directement ces requêtes en OntoQL. Cependant, lorsque les requêtes ne sont pas typées, la classe dans laquelle les instances doivent être recherchées peut être déterminée à partir des propriétés qui sont appliquées à ces instances. Cette classe est la plus basse classe dans la hiérarchie qui contient les domaines de l'ensemble de ces propriétés.

Exemple. Rechercher les messages et leur auteur.

```
SELECT *                         <=> 1. C = SELECT #scope FROM #property
 WHERE { ?p has_creator ?u }                  WHERE #code='has_creator'
                                     2. SELECT has_creator
                                          FROM C AS c,
                                            UNNEST(c.has_creator) AS has_creator
                                         WHERE has_creator IS NOT NULL
```

Explication. Dans cet exemple, la traduction consiste en deux requêtes. La première retourne le domaine de la propriété `has_creator`, noté `C`, dans une requête imbriquée. La seconde recherche les valeurs de la propriétés `has_creator` pour les instances de `C`.

Les requêtes non typées peuvent donc être exécutées sur OntoDB. Cependant, pour obtenir de meilleures performances il est souhaitable qu'elles soient typées.

6.2.2 Requêtes SPARQL sur les ontologies

Une requête sur les ontologies retourne des identifiants de concepts définis dans les ontologies et/ou leurs valeurs d'attributs à partir des éléments du modèle d'ontologies RDF-Schema. La clause `WHERE`

d'une requête SPARQL sur l'ontologie est donc composée de deux types de triplets :

- `(?oid, type, E)` où `E` est une entité du modèle d'ontologies. Ce triplet permet de retrouver les identifiants (`oid`) des éléments des ontologies, instances de `E` ;
- `(?oid, a, ?val)` où `a` est un attribut défini sur le modèle d'ontologies. Ce triplet permet de retrouver la valeur (`val`) de l'attribut `a` de l'élément d'une ontologie d'identifiant `oid`.

6.2.2.1 Définition des requêtes SPARQL sur les ontologies considérées

Les requêtes sur l'ontologie considérées ont une forme similaire à celle définie sur les données :

$(?\mathtt{oid}_1, \mathtt{type}, \mathtt{E}_1)$ [OPTIONAL] $(?\mathtt{oid}_1, \mathtt{a}_{11}, ?\mathtt{val}_{11}) \cdots$ [OPTIONAL] $(?\mathtt{oid}_1, \mathtt{a}_{n1}, ?\mathtt{val}_{n1})$ [FILTER()] θ
$(?\mathtt{oid}_2, \mathtt{type}, \mathtt{E}_2)$ [OPTIONAL] $(?\mathtt{oid}_2, \mathtt{a}_{12}, ?\mathtt{val}_{12}) \cdots$ [OPTIONAL] $(?\mathtt{oid}_1, \mathtt{a}_{n2}, ?\mathtt{val}_{n2})$ [FILTER()] θ
...
$(?\mathtt{oid}_n, \mathtt{type}, \mathtt{E}_n)$ [OPTIONAL] $(?\mathtt{oid}_n, \mathtt{a}_{1n}, ?\mathtt{val}_{1n}) \cdots$ [OPTIONAL] $(?\mathtt{oid}_n, \mathtt{a}_{nn}, ?\mathtt{val}_{nn})$ [FILTER()]

Dans ces requêtes les entités et attributs que l'on peut trouver sont ceux définis dans le modèle RDF-Schema [Brickley and Guha, 2004].

6.2.2.2 Traduction des requêtes sur les ontologies considérées en OntoQL

La conversion d'une requête sur l'ontologie est similaire à la traduction des requêtes sur les données. Ces requêtes portant sur des entités et attributs de RDF-Schema, il est nécessaire de connaître le mapping entre ces entités et attributs et ceux du modèle noyau de OntoQL. Ce mapping est présenté dans le tableau 6.6 pour les entités et attributs principaux de RDF-Schema[40]. Ce tableau montre que ces entités et attributs ont, pour la plupart, un équivalent dans le modèle noyau de OntoQL. Pour ceux dont ce n'est pas le cas, les capacités d'extension du modèle noyau permettent de les prendre en compte. Par exemple, l'attribut `rdfs:subPropertyOf` de domaine et codomaine `rdf:Property` peut être ajouté à ce modèle, ce qui permet de traduire les requêtes portant sur cet attribut. L'exemple suivant illustre l'utilisation de ce mapping pour traduire une requête sur les ontologies.

Exemple. Rechercher le domaine et codomaine des propriétés.

```
SELECT ?d ?r                                  <=>     SELECT p.#scope, p.#range
 WHERE { ?p type rdf:Property                           FROM #Property AS p
         OPTIONAL { ?p rdfs:domain ?d }
         OPTIONAL { ?p rdfs:range ?r } }
```

Explication. Cette requête SPARQL ne comporte qu'une seule entité. Les valeurs de deux attributs de cette entité sont retournées. Cette requête est donc traduite par une projection. L'entité RDF-Schema `rdf:Property` correspond à l'entité `#Property` et les attributs `domain` et `range` correspondent aux attributs OntoQL `#scope` et `#range`. La traduction est donc la projection de l'entité `#Property` sur les attributs `#scope` et `#range`. Cette traduction ne nécessite pas l'utilisation de l'opérateur `UNNEST` puisque ces deux attributs sont monovalués dans le modèle noyau de OntoQL.

[40]Le mapping complet est présenté dans l'annexe D, section 3.

Entités et attributs RDF-Schema	Entités et attributs du modèle noyau de OntoQL
rdfs:Resource _rdfs:label _rdfs:comment	#Concept _#name _#definition
rdfs:Class _rdfs:subClassOf	#Class _#superClasses
rdf:Property _rdfs:domain _rdfs:range _rdfs:subPropertyOf	#Property _#scope _#range _
rdfs:Datatype	#Datatype
rdfs:Literal	#SimpleType

Tab. 6.6 – Extrait du mapping entre les entités et attributs du modèle RDF-Schema et ceux du modèle noyau de OntoQL

Ainsi, des requêtes respectant les modèles de requêtes SPARQL présentés dans cette section peuvent être exécutées sur OntoDB. Ceci complète la suite d'outils disponibles pour exploiter la BDBO OntoDB. Cette interface avec le langage SPARQL est le dernier développement que nous avons mené autour du langage OntoQL. D'autres sont en cours. Nous les indiquons dans la conclusion suivante.

7 Conclusion

Dans la première partie de ce chapitre, nous avons présenté l'implantation du langage OntoQL sur la BDBO OntoDB. Cette implantation a d'abord consisté à représenter le modèle de données du langage OntoQL sur OntoDB. Les deux principales difficultés rencontrées sont (1) que le modèle de données de OntoQL est construit autour d'un modèle noyau qui peut être étendu et (2) que le modèle d'ontologies supporté par OntoDB est le modèle PLIB alors que le langage OntoQL est indépendant d'un modèle d'ontologies particulier. Pour résoudre la première difficulté, le modèle d'ontologies utilisé par OntoQL est rendu persistant dans un fichier XML à l'extérieur de la BDBO OntoDB. Pour résoudre la seconde difficulté, souhaitant réutiliser la représentation proposée par OntoDB, utilisée pour stocker de nombreuses ontologies, nous avons défini une correspondance entre le modèle PLIB et le modèle noyau de OntoQL. Ce mapping est également rendu persistant dans le fichier XML définissant le modèle d'ontologies utilisé.

La seconde étape de l'implantation du langage OntoQL consistait à concevoir l'interpréteur de requêtes. Souhaitant bénéficier de l'optimisation de l'interpréteur de requêtes du SGBD sous-jacent à la BDBO, le traitement d'une requête OntoQL consiste à la traduire en une requête SQL dépendant de la représentation proposée par la BDBO OntoDB. Pour minimiser la dépendance de cette implantation avec la BDBO, cette traduction est réalisée, d'une part, en appelant des méthodes d'interface, et, d'autre part, la requête OntoQL est d'abord traduite en une expression de l'algèbre relationnelle avant d'être traduite dans le langage SQL supporté par le SGBD. La principale étape du traitement d'une requête OntoQL est

la traduction de l'expression *OntoAlgebra* correspondante en une expression de l'algèbre relationnelle. La plupart des opérateurs et des fonctions du langage OntoQL sont traduits dans leur équivalent relationnel. Pour ceux qui n'ont pas d'équivalent, nous avons proposé une traduction alternative. Outre les optimisations réalisées par le SGBD, le processus de traitement d'une requête OntoQL comporte deux phases d'optimisation. La première phase consiste à optimiser l'expression de *OntoAlgebra* générée à partir de la requête. Elle utilise les règles d'optimisation que nous avons établies sur cette algèbre. La seconde phase consiste à optimiser l'expression de l'algèbre relationnelle qui résulte de la traduction de l'expression de *OntoAlgebra*. Elle utilise les règles d'optimisation établies sur l'algèbre relationnelle et nécessite d'avoir une bonne connaissance des optimisations réalisées par le SGBD sur lequel la BDBO est implantée.

Dans la seconde partie de ce chapitre, nous avons présenté les développements que nous avons réalisés pour compléter la suite d'outils permettant de manipuler les ontologies et les données à base ontologique stockées dans la BDBO OntoDB. En s'inspirant des outils couramment utilisés avec les SGBD traditionnels, nous avons conçu les outils suivants :

– OntoQLPlus, un éditeur de requêtes OntoQL en ligne de commandes ;
– OntoQBE, un éditeur graphique de requêtes OntoQL similaire à QBE qui permet d'utiliser des opérateurs relationnels-objets et d'obtenir la description ontologique des propriétés utilisées dans les requêtes construites ;
– JOBDBC, une API similaire au JDBC pour exécuter des requêtes OntoQL depuis le langage JAVA ;
– OntoAPI, une API qui permet de charger *à la demande* les ontologies et les données d'une BDBO sans utiliser explicitement de requêtes OntoQL.

Pour prendre en compte les particularités des BDBO par rapport aux bases de données traditionnelles, nous avons également conçu les outils suivants :

– une interface de recherche par mots clés des concepts représentés dans les ontologies. Cette interface vise à exploiter la couche linguistique des ontologies stockées dans la BDBO ;
– une interface avec le langage SPARQL. L'objectif de cette interface est de faciliter l'interopérabilité des outils OntoDB avec les systèmes du Web Sémantique qui implantent ce standard émergeant. Actuellement, l'implantation proposée permet d'exécuter des requêtes sur les ontologies et sur les données selon un modèle défini qui intègre les principaux opérateurs de SPARQL.

L'implantation réalisée ouvre de nouvelles perspectives. Pour améliorer l'implantation du langage OntoQL et les outils associés, nous avons identifié les évolutions suivantes :

– optimiser l'implantation des requêtes qui portent à la fois sur les ontologies et sur les données. La traduction de ces requêtes consiste en de nombreuses requêtes dont le nombre pourrait être minimisé ;
– implanter les techniques mises en place dans les framework de mapping objet-relationnel tel que Hibernate pour optimiser l'accès aux données et aux ontologies avec OntoAPI (gestion d'un cache, opérateur `fetch`, etc.) ;
– coupler la BDBO OntoDB avec un moteur de recherche plein texte pour améliorer la recherche dans les ontologies par mots clés tant en performance (indexation) qu'en qualité de réponse (re-

cherche approximative, etc.) ;
- compléter l'interface SPARQL pour permettre l'exécution de requêtes portant à la fois sur les ontologies et sur les données.

Conclusion et perspectives

Conclusion

Depuis les années 2000, des approches permettant d'assurer la gestion simultanée des données et des ontologies dans des bases de données appelées BDBO ont vu le jour. Dans le contexte du Web Sémantique, des BDBO ont été conçues pour permettre de rendre persistantes et de manipuler des données issues du Web. Les architectures de BDBO proposées dans ce cadre, ainsi que les langages associés sont spécifiques d'un modèle d'ontologies particulier. Ces architectures supposent une structuration logique bien particulière des données, que les langages ne permettent ni d'exploiter, ni de changer. Enfin, ces langages n'ont aucune compatibilité avec les langages traditionnels d'exploitation des bases de données de la série de SQL et ils n'exploitent pas les définitions linguistiques qui peuvent êtres associées aux concepts d'une ontologie.

Récemment, plusieurs architectures de BDBO ont été proposées [Dehainsala et al., 2007a, Pierra et al., 2005, Park et al., 2007]. En introduisant des hypothèses de typage, elles proposent une implantation des données à base ontologique se rapprochant de la structure des bases de données traditionnelles. Elles permettent ainsi non seulement de gérer des données à base ontologique, telles que celles appelées a être générées dans le cadre du Web Sémantique, mais également d'expliciter la sémantique dans des bases de données existantes, et donc exploitées par des langages de type SQL. Dans nos travaux, nous avons généralisé cette architecture pour prendre en compte les différents modèles d'ontologies et catégories d'ontologies. Puis, nous avons proposé le langage OntoQL permettant d'exploiter cette architecture.

Ainsi, les différentes contributions de ce travail sont les suivantes.

Analyse du concept d'ontologie et proposition d'un modèle de structuration en couches

Les ontologies étant utilisées par différentes communautés, telles que la communauté du traitement des données ou de la linguistique informatique, pour différents problèmes, tels que l'intégration d'information ou le traitement du langage naturel, plusieurs catégories d'ontologies sont apparues. Cette multitude d'approches a entraîné des difficultés pour (1) comprendre ce qu'est une ontologie et ce qui la différencie des autres modèles informatiques et (2) déterminer la catégorie d'ontologies à utiliser pour un problème donné.

Pour préciser le sens du concept d'ontologie, nous avons proposé trois critères pour caractériser une ontologie. Une ontologie de domaine doit être *formelle*, c'est-à-dire permettre des capacités de raisonnement et de vérification de sa consistance, *consensuelle* dans une communauté et doit *pouvoir être référencée* par des identifiants à partir de n'importe quel environnement. Ces trois critères caractérisent une ontologie de domaine comme un nouveau modèle informatique. Ils nous ont conduits à proposer une nouvelle définition d'une ontologie de domaine comme un *dictionnaire formel et consensuel des catégories et propriétés des entités d'un domaine d'étude et des relations qui les lient*.

Pour préciser les catégories d'ontologies existantes, nous avons proposé une taxonomie des ontologies de domaine composée de trois catégories : OCC, OCNC et OL. Afin de permettre la coopération entre ces différentes catégories, nous avons proposé un modèle en couches, nommé le *modèle en oignon*, qui permet de concevoir et d'utiliser les capacités de chaque catégorie d'ontologies dans un environnement intégré.

Mise en évidence de la complémentarité des différents modèles d'ontologies

Les modèles d'ontologies utilisés pour représenter des ontologies sont nombreux. En plus des modèles d'ontologies dits « traditionnels » tels que PLIB, F-Logic, CARIN, Classic, Ontolingua, OCML ou OKBC qui sont notamment utilisés dans le contexte de l'intégration de données, des modèles d'ontologies dits « Web » tels que SHOE, DAML, OIL, RDF-Schema ou OWL sont apparus dans le contexte du Web Sémantique. Cette diversité engendre des difficultés pour, d'une part, déterminer le modèle d'ontologies à utiliser pour un problème donné, et, d'autre part, utiliser conjointement des ontologies représentées par différents modèles d'ontologies.

En nous basant sur le modèle en oignon, nous avons analysé les constructeurs proposés par les modèles d'ontologies PLIB, F-Logic, RDF-Schema et OWL. Nous avons montré, d'une part, que ces modèles présentaient un noyau de constructeurs communs permettant de définir une OCC, et, d'autre part, qu'ils présentaient chacun des spécificités complémentaires pour la conception d'une ontologie selon le modèle en oignon. Nous avons exploité ces résultats pour généraliser les architectures récentes de BDBO et pour leur associer un langage d'exploitation basé sur ce noyau commun qui peut être étendu pour prendre en compte les spécificités de chaque modèle.

Généralisation des architectures récentes de BDBO

Dans les architectures de BDBO proposées dans le contexte du Web Sémantique, le niveau logique de la base de données est utilisé pour implanter les données à base ontologique selon des modèles éclatés de type binaire ou ternaire ne représentant pas la structure relationnelle que peuvent avoir les données. Or, la représentation de cette structure et sa manipulation sont nécessaires dans les bases de données traditionnelles.

Récemment, des architectures de bases de données dont l'implantation des données à base ontologique se rapproche de la structure relationnelle ont été proposées. En nous basant sur la complémentarité des modèles d'ontologies et sur le modèle en oignon d'une ontologie de domaine, nous avons proposé une généralisation de ces architectures. Cette architecture étend l'architecture ANSI/SPARC avec (1) le niveau ontologique constitué d'ontologies couvrant le domaine pour lequel la base de données est conçue

et (2) le niveau conceptuel constitué de l'ensemble des concepts d'une ontologie utilisés pour décrire les données d'une application particulière. Les ontologies stockées au niveau ontologique sont représentées selon le noyau commun des modèles d'ontologies qui peut être étendu. Elles sont construites selon le modèle en oignon. Ainsi, cette architecture est flexible et réutilisable puisqu'elle n'est pas spécifique d'un modèle d'ontologie ou d'une catégorie d'ontologie particulière.

Pour compléter cette proposition, en suivant l'exemple de la définition de l'architecture ANSI/SPARC, nous avons établi une liste d'exigences pour un langage d'exploitation d'une telle architecture. Cette architecture nous a ainsi servi de référence pour concevoir le langage OntoQL.

Définition d'un langage d'exploitation de BDBO

La plupart des langages de BDBO proposés dans la littérature ne proposent qu'un langage de requête, sont conçus pour un modèle d'ontologies donné, utilisent peu ou pas la description linguistique d'une ontologie et ne permettent pas de manipuler les données au niveau logique. En conséquence, ils ne satisfont pas les exigences établies pour l'architecture de référence de BDBO considérée.

Pour répondre à ces exigences, nous avons proposé le langage OntoQL. Ce langage permet de définir, manipuler et interroger les données et les ontologies d'une BDBO selon les couches suivantes :

- la couche d'accès aux données du niveau logique. Elle permet de manipuler les données comme dans une base de données relationnelle en étant compatible avec le langage SQL implanté par les SGBD couramment utilisés tels que PostgreSQL ou Oracle ;
- la couche d'accès aux données du niveau ontologique, partie OCC. Elle permet d'accéder aux données indépendamment du modèle logique des données en proposant une syntaxe proche de SQL et en intégrant des opérateurs relationnels-objets introduits dans la norme SQL99 ;
- la couche d'accès aux données du niveau ontologique, partie OCNC. Elle permet de définir des concepts non canoniques en utilisant le mécanisme de vues des bases de données ;
- la couche d'accès aux données du niveau ontologique, partie OL. Elle permet d'exprimer des requêtes multilingues en exploitant les termes associés aux concepts d'une ontologie ;
- la couche d'accès aux ontologies. Elle permet de manipuler les ontologies selon le modèle d'ontologies noyau contenant les constructeurs communs aux différents modèles d'ontologies. Ce noyau peut être étendu via des instructions du langage.

De plus, le langage OntoQL permet d'interroger à la fois les ontologies et les données d'une BDBO. Pour fournir cette capacité, nous avons d'abord identifié des cas d'utilisation où cette capacité présente un intérêt. Puis, nous avons augmenté le langage OntoQL par des mécanismes inspirés des langages proposés pour les bases de données fédérées tels que SchemaSQL et MSQL.

Le langage OntoQL permet ainsi un accès homogène à une BDBO puisque sa syntaxe reste proche de SQL dans les différentes couches d'accès. Il propose de plus les opérateurs traditionnels des bases de données ce qui n'est pas le cas des autres langages d'accès aux BDBO.

Définition d'une sémantique formelle du langage OntoQL

Proposer un langage de requêtes de bases de données sans en donner la définition formelle rend difficile son implantation, l'étude de ses propriétés sémantiques et de l'optimisation de requêtes écrites

dans ce langage. Nous avons donc défini une algèbre d'opérateurs pour le langage OntoQL nommée *OntoAlgebra*.

L'algèbre *OntoAlgebra* a été conçue à partir de l'algèbre *Encore* proposée pour les BDOO. Cette algèbre restant proche de l'algèbre relationnelle, ceci nous a permis de garantir la fermeture et la complétude relationnelle du langage OntoQL et de réutiliser les techniques traditionnelles des bases de données pour proposer quelques optimisations des requêtes OntoQL. Pour aller plus loin sur l'étude de l'optimisation de telles requêtes, nous avons proposé des techniques d'évaluation partielle qui exploitent les particularités des BDBO par rapport aux BDOO.

Implantation et outillage du langage OntoQL

La BDBO OntoDB, conçue au sein du laboratoire LISI pour stocker des données et des ontologies PLIB, a été équipée d'un éditeur d'ontologies et de différentes API d'accès aux données et aux ontologies qu'elle contient. Elle est utilisée dans différents projets de recherche dans lesquels le besoin d'un langage d'interrogation de données s'est fait sentir. Nous avons donc implanté le langage OntoQL afin de montrer, d'une part, la faisabilité de l'implantation de ce langage et, d'autre part, pour compléter les outils disponibles pour OntoDB.

Le résultat des développements menés autour du langage OntoQL est d'abord que ce langage est maintenant disponible sur cette BDBO ; et, ensuite, que cette BDBO est équipée d'outils supplémentaires facilitant l'utilisation du langage OntoQL. Ces développements comportent des outils similaires aux outils classiques des bases de données (QBE, JDBC, etc.) mais adaptés aux particularités des BDBO et des outils spécifiques permettant la recherche de concepts des ontologies par mots clés et d'utiliser le langage SPARQL sur cette BDBO facilitant ainsi son interopérabilité avec les outils du Web Sémantique.

Perspectives

Les travaux présentés dans ce mémoire laissent envisager de nombreuses perspectives tant à caractère théorique que pratique. Dans cette section, nous présentons succinctement celles qui nous paraissent être les plus intéressantes.

Extension sémantique du modèle noyau de OntoQL

En étendant le modèle noyau de OntoQL par spécialisation des entités de ce modèle, les nouvelles entités héritent automatiquement de la sémantique pré-définie des entités dont elles héritent. Une perspective très séduisante serait de proposer des mécanismes permettant d'associer de façon simple des caractéristiques sémantiques spécifiques aux constructeurs ajoutés. Nous envisageons deux approches principales.

Nous avons d'abord vu, dans le chapitre 4, section 4.1, qu'il était possible d'associer une nouvelle sémantique aux constructeurs tels que `hasValue` en utilisant les différents langages de OntoQL. En effet, pour chaque restriction OWL `hasValue` recherchée par une requête, une instruction OntoQL peut être exécutée pour définir son extension comme une vue dont la requête dépend des valeurs d'attributs

de cette restriction. Nous prévoyons d'intégrer cette possibilité dans le langage. L'approche que nous envisageons permettrait d'associer une requête OntoQL paramétrée à une entité héritant de `#Class`. A la création d'une instance de cette entité (une classe), son extension sera automatiquement construite en instanciant les paramètres de la requête paramétrée en fonction des valeurs d'attributs de cette classe.

Notre seconde idée est de fournir une API, dont l'implémentation serait à la charge des développeurs, et qui permettrait de modifier le comportement des opérateurs de OntoQL. Nous comptons donc modifier l'interpréteur de requêtes OntoQL que nous avons conçu de manière à ce qu'il soit capable de fonctionner à l'aide de *plugin*, par exemple codant la sémantique des constructeurs ajoutés.

Optimisation des requêtes OntoQL

L'étude de l'optimisation de requêtes OntoQL que nous avons présentée dans ce mémoire est un travail préliminaire. Outre la nécessité de valider expérimentalement les optimisations proposées, il serait intéressant d'étudier de nouvelles techniques d'optimisation. Nous pensons que l'étude de l'optimisation de requêtes OntoQL nécessite, en premier lieu, d'adapter les techniques d'optimisation proposées pour les BDRO au contexte des BDBO. Par exemple, il nous semble intéressant d'identifier des vues matérialisées susceptibles d'optimiser les traitements réalisés sur une BDBO puis d'étudier l'utilisation de ces vues dans l'exécution d'une requête OntoQL. En second lieu, nous pensons également étudier de nouvelles techniques d'optimisation exploitant les spécificités du langage OntoQL qui permettraient une meilleure optimisation des requêtes. Par exemple, une caractéristique de ce langage est qu'il est paramétré par l'espace de noms et la langue naturelle par rapport auxquels une requête est évaluée. Nous pensons que l'exploitation de cette caractéristique, pour partitionner les données selon ces paramètres afin de diminuer l'espace de recherche considéré pour répondre à une requête, est une piste à explorer.

Utilisation du langage OntoQL pour l'intégration de données

Dans [Nguyen-Xuan, 2006], des scénarios d'intégration de BDBO dans une architecture d'entrepôt ont été proposés. Ces scénarios ont été formalisés en spécifiant les algorithmes permettant de construire le système intégré à partir d'un ensemble de sources hétérogènes. Deux approches ont été considérées pour l'implantation de ces scénarios. La première, qualifiée de spécifique, consiste à utiliser au maximum le langage SQL de manière à minimiser l'accès à la base de données et à profiter des optimisations réalisées par le SGBD. Cette implantation présente l'inconvénient d'être dépendante de la représentation de la BDBO puisque le nom des tables stockant les ontologies ou les instances est codé « en dur » dans les requêtes SQL. La seconde approche, qualifiée de générique, consiste à définir une API permettant l'accès aux éléments de la BDBO. Cette approche permet de minimiser la dépendance entre l'implantation des algorithmes de ces scénarios et la représentation de la BDBO. En effet, l'implantation de ces scénarios sur une BDBO proposant une représentation différente nécessite seulement de fournir une nouvelle implémentation de ces interfaces. Cependant, cette API encapsule l'accès à la base de données ce qui ne permet pas de les optimiser. En effet, cette API nécessite souvent l'appel de plusieurs méthodes réalisant un accès complet à la base de données là où l'exécution d'une seule requête réaliserait le même traitement plus efficacement.

La proposition du langage OntoQL ouvre une troisième approche pour l'implantation de ces scéna-

rios combinant les avantages des deux précédentes approches. Elle consiste à utiliser ce langage pour implanter ces scénarios. Ceci permet de minimiser l'accès à la BDBO et de profiter des optimisations réalisées sur le traitement d'une requête OntoQL. De plus, puisque le langage OntoQL encapsule l'accès aux données d'une BDBO, cette approche est indépendante de la représentation qu'elle propose.

Enfin, les perspectives des travaux présentées dans [Nguyen-Xuan, 2006] consistent à adapter les approches d'intégration proposées à une architecture de médiation. Une telle architecture permet d'exécuter une requête qui est réécrite sur chaque source de données. Le langage OntoQL peut-être utilisé comme langage de requête ce qui permettrait d'étudier la réécriture de requêtes écrites dans ce langage et l'optimisation de ces requêtes dans une architecture de médiation.

Utilisation du langage OntoQL pour les systèmes de méta-modélisation

Un système de méta-modélisation est un système gérant trois niveaux d'information : instance, modèle et méta-modèle. C'est le cas des BDBO où le niveau modèle est constitué des ontologies et où le niveau méta-modèle contient le modèle d'ontologies utilisé. C'est également le cas des bases de données, qu'elles soient relationnelles, objets ou relationnelles-objets. En effet, les données sont des instances d'un schéma dont la structure est représentée en instanciant une métabase. Enfin, dans le contexte du génie logiciel, l'OMG a suggéré de représenter les modèles UML en utilisant le MOF [MOF, 2002]. A nouveau, trois niveaux de modélisation doivent être représentés dans le même système.

Le problème principal qui se pose pour la gestion de systèmes de méta-modélisation est que, désormais, les méta-modèles sont nombreux et évolutifs. Nous l'avons vu pour les BDBO avec la variété de modèles d'ontologies disponibles. De même, dans le contexte des bases de données, chaque système emploie sa propre structure de métabase. C'est également le cas dans le contexte du génie logiciel où chaque atelier UML implante, de façon plus ou moins complète, une des versions de la norme UML. Ainsi, les utilisateurs des systèmes de méta-modélisation sont confrontés à la nécessité de s'adapter à la diversité et à l'évolution des méta-modèles. Or, si des langages ont été définis sur ces structures afin de permettre la manipulation des instances et des modèles, à notre connaissance, seul le langage OntoQL permet, dans le contexte des BDBO, de manipuler le niveau méta-modèle de la même manière que les niveaux modèle et instance. Nous pensons qu'il serait intéressant d'étudier son implantation pour d'autres systèmes de méta-modélisation car le modèle noyau sur lequel il est défini, à savoir le modèle classe/propriété, est proche des modèles utilisés dans ces différents systèmes.

Utilisation du langage OntoQL pour l'indexation sémantique de bases de données

Le langage OntoQL permet de créer le schéma logique des données à partir des ontologies d'une BDBO. En effet, il permet d'associer à chaque classe d'une ontologie une extension représentée au niveau logique par une table relationnelle dont le nom et les noms de colonnes sont générés automatiquement par défaut.

Si un schéma de base de données existe, la seule solution qu'offre aujourd'hui le langage OntoQL pour associer les tables de ce schéma à une ontologie est de définir des vues. En effet, des vues peuvent être créées à partir des tables existantes pour représenter les extensions de classes. Cependant, cette opération nécessite la connaissance des conventions de nommage du langage OntoQL pour attribuer des

noms, d'une part, à la vue construite et, d'autre part, à ses attributs pour que l'interpréteur OntoQL puisse les utiliser pour répondre aux requêtes niveau ontologique. De plus, ce processus est limité par le pouvoir d'expression du LID niveau logique de OntoQL. Une perspective de travail sur le langage OntoQL est donc de fournir des mécanismes permettant d'associer de manière déclarative des tables existantes au niveau logique d'une BDBO aux ontologies qu'elle conserve au niveau ontologique. On disposerait alors d'un environnement complet pour indexer sémantiquement les bases de données existantes et offrir ainsi des accès au niveau connaissance, indépendant des schémas logiques spécifiques.

Bibliographie

[MOF, 2002] (2002). *Meta Object Facility (MOF), formal/02-04-03*. Object Management Group.

[ODM, 2006] (2006). *Ontology Definition Metamodel (ODM) Final Adopted Specification ptc/06-10-11*. Object Management Group.

[Abadi et al., 2007] Abadi, D. J., Marcus, A., Madden, S., and Hollenbach, K. J. (2007). Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07)*, pages 411–422.

[Abiteboul and Bonner, 1991] Abiteboul, S. and Bonner, A. J. (1991). Objects and Views. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data (SIGMOD'91)*, pages 238–247.

[Agrawal et al., 2001] Agrawal, R., Somani, A., and Xu, Y. (2001). Storage and Querying of E-Commerce Data. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 149–158. Morgan Kaufmann Publishers Inc.

[Alexaki et al., 2001] Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., and Tolle, K. (2001). The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *Proceedings of the 2nd International Workshop on the Semantic Web*, pages 1–13.

[An et al., 2006] An, Y., Borgida, A., and Mylopoulos, J. (2006). Discovering the Semantics of Relational Tables Through Mappings. *Journal on Data Semantics VII*, pages 1–32.

[ANSI/X3/SPARC, 1975] ANSI/X3/SPARC (1975). Study Group on Data Management Systems, Interim Report. *Bulletin of ACM SIGMOD*, 7(2).

[Arens et al., 1993] Arens, Y., Chee, C. Y., Hsu, C.-N., and Knoblock, C. A. (1993). Retrieving and Integrating Data from Multiple Information Sources. *International Journal of Cooperative Information Systems (IJCIS)*, 2(2):127–158.

[Athanasis et al., 2004] Athanasis, N., Christophides, V., and Kotzinos, D. (2004). Generating On the Fly Queries for the Semantic Web: The ICS-FORTH Graphical RQL Interface (GRQL). In *Proceedings of the 3rd International Semantic Web Conference (ISWC'04)*, pages 486–501.

[Aussenac-Gilles et al., 2000] Aussenac-Gilles, N., Biebow, B., and Szulman, S. (2000). Revisiting Ontology Design: A Methodology Based on Corpus Analysis. In Dieng, R. and Corby, O., editors, *Proceedings of the 12th European Knowledge Acquisition Workshop (EKAW'00)*, pages 172–188.

[Aussenac-Gilles and Mothe, 2004] Aussenac-Gilles, N. and Mothe, J. (2004). Ontologies as Background Knowledge to Explore Document Collections. In *Actes de la Conférence sur la Recherche d'Information Assistée par Ordinateur (RIAO'04)*, pages 129–142.

[Bailey et al., 2005] Bailey, J., Bry, F., Furche, T., and Schaffert, S. (2005). Web and Semantic Web Query Languages: A Survey. In *Reasoning*

Web, First International Summer School, Lecture Notes in Computer Science, pages 35–133. Springer.

[Bancilhon et al., 1992] Bancilhon, F., Delobel, C., and Kanellakis, P. C., editors (1992). *Building an Object-Oriented Database System, The Story of O2*. Morgan Kaufmann.

[Banerjee et al., 1987] Banerjee, J., Chou, H.-T., Garza, J. F., Kim, W., Woelk, D., Ballou, N., and Kim, H.-J. (1987). Data model issues for object-oriented applications. *ACM Transactions on Information Systems (TOIS)*, 5(1):3–26.

[Barrasa et al., 2004] Barrasa, J., Corcho, Ó., and Gómez-Pérez, A. (2004). R2O, an Extensible and Semantically Based Database-to-ontology Mapping Language. In *Proceedings of the 2nd Workshop on Semantic Web and Databases (SWDB'04)*.

[Bellatreche et al., 2004] Bellatreche, L., Pierra, G., Nguyen-Xuan, D., Hondjack, D., and Ait-Ameur, Y. (2004). An a Priori Approach for Automatic Integration of Heterogeneous and Autonomous Databases. In *Proceedings of the 15th International Conference on Database and Expert Systems Applications (DEXA'04)*, pages 475–485.

[Beneventano et al., 2000] Beneventano, D., Bergamaschi, S., Castano, S., Corni, A., Guidetti, R., Malvezzi, G., Melchiori, M., and Vincini, M. (2000). Information Integration: The MOMIS Project Demonstration. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB'00)*, pages 611–614. Morgan Kaufmann.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*, 284(5):34–43.

[B.McBride, 2001] B.McBride (2001). Jena: Implementing the RDF Model and Syntax Specification. *Proceedings of the 2nd International Workshop on the Semantic Web*.

[Borst, 1997] Borst, W. N. (1997). *Construction of Engineering Ontologies*. PhD thesis, University of Twente, Enschede.

[Bozsak et al., 2002] Bozsak, E., Ehrig, M., Handschuh, S., Hotho, A., Maedche, A., Motik, B., Oberle, D., Schmitz, C., Staab, S., Stojanovic, L., Stojanovic, N., Studer, R., Stumme, G., Sure, Y., Tane, J., Volz, R., and Zacharias, V. (2002). KAON - Towards a Large Scale Semantic Web. In *Proceedings of the 3rd International Conference on E-Commerce and Web Technologies (EC-WEB'02)*, pages 304–313, London, UK. Springer-Verlag.

[Braga et al., 2005] Braga, D., Campi, A., and Ceri, S. (2005). XQBE (XQuery By Example): A Visual Interface to the Standard XML Query Language. *ACM Transactions on Database Systems (TODS)*, 30(2):398–443.

[Brickley and Guha, 2004] Brickley, D. and Guha, R. V. (2004). *RDF Vocabulary Description Language 1.0: RDF Schema*. World Wide Web Consortium. `http://www.w3.org/TR/rdf-schema`.

[Broekstra et al., 2002] Broekstra, J., Kampman, A., and van Harmelen, F. (2002). Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Horrocks, I. and Hendler, J., editors, *Proceedings of the 1st International Semantic Web Conference (ISWC'02)*, number 2342 in Lecture Notes in Computer Science, pages 54–68. Springer Verlag.

[Broeskstra and Kampman, 2003] Broeskstra, J. and Kampman, A. (2003). SeRQL: A Second Generation RDF Query Language. In *SWAD-Europe Workshop on Semantic Web Storage and Retrieval*.

[Carroll et al., 2005] Carroll, J. J., Bizer, C., Hayes, P., and Stickler, P. (2005). Named Graphs, Provenance and Trust. In *Proceedings of the 14th international conference on World Wide Web (WWW'05)*, pages 613–622, New York, NY, USA. ACM Press.

[Carroll et al., 2004] Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). Jena: Implementing the Semantic Web Recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (WWW'04)*, pages 74–83, New York, NY, USA. ACM Press.

[Cattell, 1993] Cattell, R. G. G. (1993). *The Object Database Standard: ODMG-93*. Morgan Kaufmann.

[Chawathe et al., 1994] Chawathe, S. S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J. D., and Widom, J. (1994). The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proceedings of the 10th Meeting of the Information Processing Society of Japan (IPSJ'94)*, pages 7–18.

[Chong et al., 2005] Chong, E. I., Das, S., Eadon, G., and Srinivasan, J. (2005). An Efficient SQL-based RDF Querying Scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05)*, pages 1216–1227.

[Christophides et al., 1994] Christophides, V., Abiteboul, S., Cluet, S., and Scholl, M. (1994). From Structured Documents to Novel Query Facilities. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data (SIGMOD'94)*, pages 313–324, New York, NY, USA. ACM Press.

[Christophides et al., 1996] Christophides, V., Cluet, S., and Moerkotte, G. (1996). Evaluating Queries with Generalized Path Expressions. In Jagadish, H. V. and Mumick, I. S., editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*, pages 413–422. ACM Press.

[Codd, 1972] Codd, E. F. (1972). Relational Completeness of Data Base Sublanguages. *In: R. Rustin (ed.): Database Systems, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 6:65–98.

[Connolly et al., 2001] Connolly, D., van Harmelen, F., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2001). *DAML+OIL Reference Description*. World Wide Web Consortium. `http://www.w3.org/TR/daml+oil-reference`.

[Corby et al., 2006] Corby, O., Dieng-Kuntz, R., Faron-Zucker, C., and Gandon, F. (2006). Searching the Semantic Web: Approximate Query Processing Based on Ontologies. *IEEE Intelligent Systems*, 21(1):20–27.

[Cullot et al., 2003] Cullot, N., Parent, C., Spaccapietra, S., and Vangenot, C. (2003). Ontologies: A Contribution to the DL/DB Debate. In *Proceedings of the 1st International Workshop on Semantic Web and Databases (SWDB'03)*, pages 109–129.

[Cyganiak, 2005] Cyganiak, R. (2005). A Relational Algebra for SPARQL. Technical Report 2005-170, HP-Labs.

[Das et al., 2004] Das, S., Chong, E. I., Eadon, G., and Srinivasan, J. (2004). Supporting Ontology-Based Semantic matching in RDBMS. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, pages 1054–1065.

[Date, 1982] Date, C. J. (1982). Null Values in Database Management. In *Proceedings of the 2nd British National Conference on Databases (BNCOD'82)*, pages 147–166.

[de Bruijn et al., 2005] de Bruijn, J., Lara, R., Polleres, A., and Fensel, D. (2005). OWL DL vs. OWL Flight: Conceptual Modeling and Reasoning for the Semantic Web. In Ellis, A. and Hagino, T., editors, *Proceedings of the 14th international conference on World Wide Web (WWW'05)*, pages 623–632. ACM.

[de Laborda and Conrad, 2006] de Laborda, C. P. and Conrad, S. (2006). Database to Semantic Web Mapping Using RDF Query Languages.

In *Proceedings of the 25th International Conference on Conceptual Modeling (ER'06)*, pages 241–254.

[Dean and Schreiber, 2004] Dean, M. and Schreiber, G. (2004). *OWL Web Ontology Language Reference*. World Wide Web Consortium. `http://www.w3.org/TR/owl-ref`.

[Dehainsala, 2007] Dehainsala, H. (2007). *Explicitation de la sémantique dans les bases de données : Le modèle OntoDB de bases de données à base ontologique*. PhD thesis, LISI/ENSMA et Université de Poitiers.

[Dehainsala et al., 2007a] Dehainsala, H., Pierra, G., and Bellatreche, L. (2007a). OntoDB: An Ontology-Based Database for Data Intensive Applications. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, volume 4443 of *Lecture Notes in Computer Science*, pages 497–508. Springer.

[Dehainsala et al., 2007b] Dehainsala, H., Pierra, G., Bellatreche, L., and Aït-Ameur, Y. (2007b). Conception de bases de données à partir d'ontologies de domaine : Application aux bases de données du domaine technique. In *Actes des 1ère Journées Francophones sur les Ontologies (JFO'07)*, pages 215–230.

[del Mar Roldán García et al., 2005] del Mar Roldán García, M., Delgado, I. N., and Montes, J. F. A. (2005). A Design Methodology for Semantic Web Database-Based Systems. In *Proceedings of the 3rd International Conference on Information Technology and Applications (ICITA'05)*, pages 233–237. IEEE Computer Society.

[Deutsch et al., 1999] Deutsch, A., Fernández, M. F., Florescu, D., Levy, A. Y., and Suciu, D. (1999). A Query Language for XML. *Computer Networks*, 31(11-16):1155–1169.

[dos Santos et al., 1994] dos Santos, C. S., Abiteboul, S., and Delobel, C. (1994). Virtual Schemas and Bases. In Jarke, M., Jr., J. A. B., and Jeffery, K. G., editors, *Proceedings of the 4th International Conference on Extending Database Technology (EDBT'94)*, volume 779 of *Lecture Notes in Computer Science*, pages 81–94. Springer.

[Dou et al., 2003] Dou, D., McDermott, D., and Qi, P. (2003). Ontology Translation on the Semantic Web. In *Proceeding of the 2nd International Conference on Ontologies, Databases and Applications of Semantics (ODBASE'03)*, pages 952–969.

[Eisenberg et al., 2004] Eisenberg, A., Melton, J., Kulkarni, K., Michels, J.-E., and Zemke, F. (2004). SQL:2003 Has Been Published. *SIGMOD Record*, 33(1):119–126.

[Estival et al., 2004] Estival, D., Nowak, C., and Zschorn, A. (2004). Towards Ontology-based Natural Language Processing. In *RDF/RDFS and OWL in Language Technology: 4th Workshop on NLP and XML (NLPXML-2004), ACL 2004*.

[Fankam, 2007] Fankam, C. (2007). Prise en compte des ontologies non canoniques dans les BDBO : le modèle ONTODB2. In *Actes du XXVème congrès INFORSID (INFORSID'07)*, pages 561–562.

[Farquhar et al., 1997] Farquhar, A., Fikes, R., and Rice, J. (1997). The Ontolingua Server: a Tool for Collaborative Ontology Construction. *International Journal of Human Computer Studies (IJHCS)*, 46(6):707–727.

[Fensel et al., 2001] Fensel, D., van Harmelen, F., Horrocks, I., McGuinness, D. L., and Patel-Schneider, P. F. (2001). OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2):38–45.

[Fikes et al., 2004] Fikes, R., Hayes, P. J., and Horrocks, I. (2004). OWL-QL - a language for deductive query answering on the Semantic Web. *Journal of Web Semantics*, 2(1):19–29.

[Gangemi et al., 2003] Gangemi, A., Guarino, N., Masolo, C., and Oltramari, A. (2003). Sweete-

ning WORDNET with DOLCE. *AI Magazine*, 24(3):13–24.

[Genesereth, 1991] Genesereth, M. R. (1991). Knowledge Interchange Format. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 599–600.

[Goh, 1997] Goh, C. H. (1997). *Representing and reasoning about semantic conflicts in heterogeneous information systems*. PhD thesis, MIT Sloan School of Management.

[Gruber, 1993] Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220.

[Gruber, 1995] Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies (IJHCS)*, 43(5-6):907–928.

[Guarino, 1998] Guarino, N. (1998). Formal Ontology and Information Systems. In Guarino, N., editor, *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems (FOIS'98)*, pages 3–15. IOS Press.

[Guerrini et al., 1997] Guerrini, G., Bertino, E., Catania, B., and Garcia-Molina, J. (1997). A Formal Model of Views for Object-Oriented Database Systems. *Theory and Practice of Object Systems*, 3(3):157–183.

[Haarslev and Möller, 2001] Haarslev, V. and Möller, R. (2001). Description of the RACER System and its Applications. In *Working Notes of the 2001 International Description Logics Workshop (DL'01)*.

[Haase et al., 2004] Haase, P., Broekstra, J., Eberhart, A., and Volz, R. (2004). A Comparison of RDF Query Languages. In *Proceedings of the 3nd International Semantic Web Conference (ISWC'04)*, pages 502–517.

[Haav and Lubi, 2001] Haav, H.-M. and Lubi, T.-L. (2001). A Survey of Concept-based Information Retrieval Tools on the Web. In *Proceedings of the 5th East European Conferenceon Advances in Databases and Information Systems (ADBIS'01)*, pages 29–41.

[Harris and Gibbins, 2003] Harris, S. and Gibbins, N. (2003). 3store: Efficient bulk RDF Storage. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PPP'03)*, pages 1–15.

[Horrocks et al., 2003] Horrocks, I., Patel-Schneider, P. F., and van Harmelen, F. (2003). From $\mathcal{SHIQ}$ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1(1):7–26.

[IEC61360-4, 1999] IEC61360-4 (1999). Standard data element types with associated classification scheme for electric components - Part 4 : IEC reference collection of standard data element types, component classes and terms. Technical report, International Standards Organization.

[ISO10303, 1994] ISO10303 (1994). Initial release of international standard(is) 10303. Technical report is 10303, International Standards Organization.

[ISO13584-25, 2004] ISO13584-25 (2004). Industrial automation systems and integration – Parts library – Part 25: Logical resource: Logical model of supplier library with aggregate values and explicit content. Technical report, International Standards Organization, Genève.

[ISO13584-42, 1998] ISO13584-42 (1998). Industrial automation systems and integration – Parts library – Part 42: Description methodology: Methodology for structuring parts families. Technical report, International Standards Organization, Genève.

[ISO639-1, 2002] ISO639-1 (2002). Codes for the representation of names of languages. Technical report, International Information Centre For Terminology.

[Jean et al., 2006a] Jean, S., Aït-Ameur, Y., and Pierra, G. (2006a). Querying Ontology Based Database Using OntoQL (an Ontology Query Language). In *Proceedings of On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, OTM Confederated International Conferences (ODBASE'06)*, volume 4275 of *Lecture Notes in Computer Science*, pages 704–721. Springer.

[Jean et al., 2006b] Jean, S., Aït-Ameur, Y., and Pierra, G. (2006b). Querying ontology based databases. The OntoQL proposal. In *Software Engineering and Knowledge Engineering (SEKE'06)*, pages 166–171.

[Jean et al., 2007a] Jean, S., Aït-Ameur, Y., and Pierra, G. (2007a). An Object-Oriented Based Algebra for Ontologies and their Instances. In *Proceedings of the 11th East European Conference in Advances in Databases and Information Systems (ADBIS'07)*, volume 4690 of *Lecture Notes in Computer Science*, pages 141–156. Springer.

[Jean et al., 2007b] Jean, S., Aït-Ameur, Y., and Pierra, G. (2007b). Une approche langage pour la gestion de données dans les systèmes de métamodélisation. In *Actes du XXVème Congrès INFORSID (INFORSID'07)*, pages 171–188.

[Jean et al., 2007c] Jean, S., Dehainsala, H., Nguyen Xuan, D., Pierra, G., Bellatreche, L., and Aït-Ameur, Y. (2007c). OntoDB: It is Time to Embed your Domain Ontology in your Database. In Kotagiri, R., Krishna, P. R., Mohania, M., and Nantajeewarawat, E., editors, *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07) (Demo Paper)*, volume 4443 of *Lecture Notes in Computer Science*, pages 1119–1122. Springer.

[Jean et al., 2005] Jean, S., Pierra, G., and Ait-Ameur, Y. (2005). OntoQL: an exploitation language for OBDBs. In *Proceedings of the VLDB 2005 PhD Workshop. Co-located with the 31th International Conference on Very Large Data Bases (VLDB'05)*, pages 41–45.

[Jean et al., 2007d] Jean, S., Pierra, G., and Ameur, Y. A. (2007d). Domain Ontologies: A Database-Oriented Analysis. In *Web Information Systems and Technologies, International Conferences, WEBIST 2005 and WEBIST 2006. Revised Selected Papers*, Lecture Notes in Business Information Processing, pages 238–254. Springer Berlin Heidelberg.

[Karvounarakis et al., 2002] Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., and Scholl, M. (2002). RQL: a declarative query language for RDF. In *Proceedings of the Eleventh International World Wide Web Conference (WWW'02)*, pages 592–603.

[Karvounarakis et al., 2004] Karvounarakis, G., Magkanaraki, A., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M., and Tolle, K. (2004). RQL: A Functional Query Language for RDF. In Gray, P. M. D., Kerschberg, L., King, P. J. H., and Poulovassilis, A., editors, *The Functional Approach to Data Management: Modelling, Analyzing and Integrating Heterogeneous Data*, LNCS, pages 435–465. Springer-Verlag.

[Kifer et al., 1995] Kifer, M., Lausen, G., and Wu, J. (1995). Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM (JACM)*, 42(4):741–843.

[Lakshmanan et al., 1999] Lakshmanan, L. V. S., Sadri, F., and Subramanian, S. N. (1999). On Efficiently Implementing SchemaSQL on an SQL Database System. In Atkinson, M. P., Orlowska, M. E., Valduriez, P., Zdonik, S. B., and Brodie, M. L., editors, *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99)*, pages 471–482. Morgan Kaufmann.

[Lakshmanan et al., 2001] Lakshmanan, L. V. S., Sadri, F., and Subramanian, S. N. (2001). SchemaSQL: An Extension to SQL for Multidata-

base Interoperability. *ACM Transactions on Database Systems (TODS)*, 26(4):476–519.

[Levy et al., 1996] Levy, A. Y., Rajaraman, A., and Ordille, J. J. (1996). Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB'96)*, pages 251–262.

[Levy and Rousset, 1998] Levy, A. Y. and Rousset, M.-C. (1998). Combining Horn Rules and Description Logics in CARIN. *Artificial Intelligence*, 104(1-2):165–209.

[Litwin et al., 1989] Litwin, W., Abdellatif, A., Zeroual, A., Nicolas, B., and Vigier, P. (1989). MSQL: a Multidatabase Language. *Information Sciences: an International Journal*, 49(1-3):59–101.

[L.Ma et al., 2004] L.Ma, Su, Z., Pan, Y., Zhang, L., and Liu, T. (2004). RStar: an RDF storage and query system for enterprise resource management. *Proceedings of the 13th ACM International Conference on Information and Knowledge Management (CIKM'04)*, pages 484 – 491.

[Magiridou et al., 2005] Magiridou, M., Sahtouris, S., Christophides, V., and Koubarakis, M. (2005). RUL: A Declarative Update Language for RDF. In *Proceedings of the 4th International Semantic Web Conference (ISWC'05)*, pages 506–521.

[Magkanaraki et al., 2004] Magkanaraki, A., Tannen, V., Christophides, V., and Plexousakis, D. (2004). Viewing the Semantic Web Through RVL Lenses. *Journal of Web Semantics*, 1(4):359–375.

[Manola and Miller, 2004] Manola, F. and Miller, E. (2004). *RDF Primer*. World Wide Web Consortium. `http://www.w3.org/TR/rdf-primer`.

[Miller et al., 2002] Miller, L., Seaborne, A., and Reggiori, A. (2002). Three Implementations of SquishQL, a Simple RDF Query Language. In *Proceedings of the 1st International Semantic Web Conference (ISWC'02)*, pages 423–435.

[Mizoguchi-Shimogori et al., 2002] Mizoguchi-Shimogori, Y., Murayama, H., and Minamino, N. (2002). Class Query Language and its application to ISO13584 Parts Library Standard. In *Proceedings of the 9th European Concurrent Engineering Conference (ECEC'02)*, pages 128–135.

[Motik et al., 2006] Motik, B., Horrocks, I., Rosati, R., and Sattler, U. (2006). Can OWL and Logic Programming Live Together Happily Ever After? In *Proceedings of the 2006 International Semantic Web Conference (ISWC'06)*, volume 4273 of *Lecture Notes in Computer Science*, pages 501–514. Springer.

[Nguyen-Xuan, 2006] Nguyen-Xuan, D. (2006). *Intégration de base de données hétérogènes par articulation a priori d'ontologies : application aux catalogues de composants industriels*. PhD thesis, LISI/ENSMA et Université de Poitiers.

[Niles and Pease, 2001] Niles, I. and Pease, A. (2001). Towards a standard upper ontology. In *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS'01)*, pages 2–9.

[Noy, 2004] Noy, N. F. (2004). Semantic Integration: A Survey Of Ontology-Based Approaches. *SIGMOD Record*, 33(4):65–70.

[Noy and McGuinness, 2001] Noy, N. F. and McGuinness, D. L. (2001). Ontology Development 101: A Guide to Creating Your First Ontology. Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, Stanford Knowledge Systems Laboratory.

[Pan and Heflin, 2003] Pan, Z. and Heflin, J. (2003). DLDB: Extending Relational Databases to Support Semantic Web Queries. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, pages 109–113.

[Park et al., 2007] Park, M. J., Lee, J. H., Lee, C. H., Lin, J., Serres, O., and Chung, C. W. (2007). An Efficient and Scalable Management of Ontology. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, volume 4443 of *Lecture Notes in Computer Science*, pages 975–980. Springer.

[Patel-Schneider et al., 2004] Patel-Schneider, P. F., Hayes, P., and Horrocks, I. (2004). *OWL Web Ontology Language Semantics and Abstract Syntax*. World Wide Web Consortium. http://www.w3.org/TR/owl-semantics/.

[Patel-Schneider and Horrocks, 2006] Patel-Schneider, P. F. and Horrocks, I. (2006). A Comparison of Two Modelling Paradigms in the Semantic Web. In *Proceedings of the 15th International World Wide Web Conference (WWW'06)*, pages 3–12. ACM.

[Pérez et al., 2006] Pérez, J., Arenas, M., and Gutierrez, C. (2006). Semantics and Complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference (ISWC'06)*, pages 30–43.

[Pierra, 2003] Pierra, G. (2003). Context-Explication in Conceptual Ontologies: The PLIB Approach. In Jardim-Gonçalves, R., Cha, J., and Steiger-Garçao, A., editors, *Proceedings of the 10th ISPE International Conference on Concurrent Engineering (CE'03)*, pages 243–254.

[Pierra, 2007] Pierra, G. (2007). Context Representation in Domain Ontologies and its Use for Semantic Integration of Data. *Journal Of Data Semantics (JODS)*, X:34–43.

[Pierra et al., 2005] Pierra, G., Dehainsala, H., Aït-Ameur, Y., and Bellatreche, L. (2005). Base de Données à Base Ontologique : principes et mise en œuvre. *Ingénierie des Systèmes d'Information*, 10(2):91–115.

[Prud'hommeaux and Seaborne, 2006] Prud'hommeaux, E. and Seaborne, A. (2006). SPARQL Query Language for RDF. *W3C Candidate Recommendation 14 June 2007*. http://www.w3.org/TR/rdf-sparql-query/.

[Psyché et al., 2003] Psyché, V., Mendes, O., and Bourdeau, J. (2003). Apport de l'ingénierie ontologique aux environnements de formation à distance. *Revue Sciences et Technologies de l'Information et de la Communication pour l'Éducation et la Formation (STICEF)*, 10.

[Rousset et al., 2002] Rousset, M.-C., Bidault, A., Froidevaux, C., Gagliardi, H., Goasdoué, F., Reynaud, C., and Safar, B. (2002). Construction de médiateurs pour intégrer des sources d'information multiples et hétérogènes: PICSEL. *Revue Information - Interaction - Intelligence (I3)*, 2(1):9–59.

[Roussey et al., 2002] Roussey, C., Calabretto, S., and Pinon, J.-M. (2002). Le thésaurus sémantique : contribution à l'ingénierie des connaissances documentaires. In *Actes des 6èmes Journées Ingénierie des Connaissances*, pages 209–220.

[Rundensteiner, 1992] Rundensteiner, E. A. (1992). Multiview: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In Yuan, L.-Y., editor, *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB'92)*, pages 187–198. Morgan Kaufmann.

[Schenk and Wilson, 1994] Schenk, D. and Wilson, P. (1994). *Information Modelling The EXPRESS Way*. Oxford University Press.

[Seaborne, 2004] Seaborne, A. (2004). RDQL – A Query Language for RDF. *W3C Member Submission 9 January 2004*. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[Smith et al., 2004] Smith, M. K., Welty, C., and McGuinness, D. L. (2004). *OWL Web Ontology Language Guide*. World Wide Web Consortium. http://www.w3.org/TR/owl-guide/.

[Stein et al., 2000] Stein, L. A., Connolly, D., and McGuinness, D. L. (2000). *DAML-ONT Initial Release*. http://www.daml.org/2000/10/daml-ont.html.

[Stoffel et al., 1997] Stoffel, K., Taylor, M., and Hendler, J. (1997). Efficient Management of Very Large Ontologies. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference AAAI'97/IAAI'97*, pages 442–447.

[Stonebraker and Moore, 1996] Stonebraker, M. and Moore, D. (1996). *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann.

[Sugumaran and Storey, 2006] Sugumaran, V. and Storey, V. C. (2006). The role of domain ontologies in database design: An ontology management and conceptual modeling environment. *ACM Transactions on Database Systems (TODS)*, 31(3):1064–1094.

[Tetlow et al., 2005] Tetlow, P., Pan, J., Oberle, D., Wallace, E., Uschold, M., and Kendall, E. (2005). *Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering*. World Wide Web Consortium. http://www.w3.org/2001/sw/BestPractices/SE/ODA/.

[Theoharis et al., 2005] Theoharis, Y., Christophides, V., and Karvounarakis, G. (2005). Benchmarking Database Representations of RDF/S Stores. In *Proceedings of the 4th International Semantic Web Conference (ISWC'05)*, pages 685–701.

[Tobies, 2001] Tobies, S. (2001). *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, RWTH Aachen, Germany.

[Tolle and Wleklinski, 2004] Tolle, K. and Wleklinski, F. (2004). *easy RDF Query Language (eRQL)*. http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL.

[Ullman, 1980] Ullman, J. D. (1980). *Principles of Database Systems*. Computer Science Press.

[Ullman et al., 2001] Ullman, J. D., Garcia-Molina, H., and Widom, J. (2001). *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Volz et al., 2003] Volz, R., Oberle, D., and Studer, R. (2003). Implementing Views for Light-Weight Web Ontologies. In *Proceedings of the 7th International Database Engineering and Applications Symposium (IDEAS'03)*, pages 160–169.

[Wache et al., 2001] Wache, H., Vögele, T., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H., and Hübner, S. (2001). Ontology-based Integration of Information — a Survey of Existing Approaches. In *Proceedings of the IJCAI-01 Workshop: Ontologies and Information Sharing*, pages 108–117.

[Wilkinson, 2006] Wilkinson, K. (2006). Jena Property Table Implementation. Technical Report 2006-140, HP-Labs.

[Wilkinson et al., 2003] Wilkinson, K., Sayers, C., Kuno, H., and Reynolds, D. (2003). Efficient RDF Storage and Retrieval in Jena2. *HP Laboratories Technical Report HPL-2003-266*, pages 131–150.

[Zdonik and Mitchell, 1991] Zdonik, S. B. and Mitchell, G. (1991). ENCORE: An Object-Oriented Approach to Database Modelling and Querying. *IEEE Data Engineering Bulletin*, 14(2):53–57.

[Ziegler et al., 2005] Ziegler, P., Sturm, C., and Dittrich, K. R. (2005). Unified Querying of Ontology Languages with the SIRUP Ontology Query API. In *Datenbanksysteme in Business, Technologie und Web (BTW'05)*, pages 325–344.

[Zloof, 1977] Zloof, M. M. (1977). Query-by-Example: A Data Base Language. *IBM Systems Journal*, 16(4):324–343.

Annexe **A**

Syntaxe complète du langage OntoQL

Cette annexe définit les éléments lexicaux du langage OntoQL et les règles grammaticales auxquelles ils doivent obéir.

1 notations

Nous utilisons les notations suivantes pour définir la syntaxe du langage OntoQL.
- ⟨*element*⟩ est un élément non terminal ;
- `element` est un élément terminal ;
- [*element*] est un élément optionnel ;
- {*element*} est un élément qui peut être répété de 0 à n fois ;
- | représente une alternative ;
- ⟨*element list*⟩ est un élément dont la définition est ⟨*element*⟩ { , ⟨*element*⟩ }.

2 Les tokens

Les règles suivantes spécifient les tokens utilisés pour les langages de définition, manipulation et interrogation de OntoQL.

2.1 Les mot-clés

Cette sous-section donne les règles définissant les mots-clés du langage OntoQL. Dans ces règles, les mots-clés sont indiqués en minuscule. Cependant, puisque le langage OntoQL n'est pas sensible à la casse, ils peuvent être écrits sans se soucier des majuscules et des minuscules.

⟨*ABS*⟩ ::= `abs`
⟨*ADD*⟩ ::= `add`
⟨*ALL*⟩ ::= `all`
⟨*ALTER*⟩ ::= `alter`
⟨*AND*⟩ ::= `and`
⟨*ANY*⟩ ::= `any`
⟨*ARRAY*⟩ ::= `array`

⟨*AS*⟩	::=	`as`
⟨*ASC*⟩	::=	`asc`
⟨*ATTRIBUTE*⟩	::=	`attribute`
⟨*AVG*⟩	::=	`avg`
⟨*BETWEEN*⟩	::=	`between`
⟨*BOOLEAN*⟩	::=	`boolean`
⟨*CARDINALITY*⟩	::=	`cardinality`
⟨*CASE*⟩	::=	`case`
⟨*CAST*⟩	::=	`cast`
⟨*CHECK*⟩	::	`check`
⟨*COALESCE*⟩	::=	`coalesce`
⟨*COLUMN*⟩	::=	`column`
⟨*CONSTRAINT*⟩	::=	`constraint`
⟨*COUNT*⟩	::=	`count`
⟨*CREATE*⟩	::=	`create`
⟨*CROSS*⟩	::=	`cross`
⟨*DATE*⟩	::=	`date`
⟨*DELETE*⟩	::=	`delete`
⟨*DEREF*⟩	::=	`deref`
⟨*DERIVED*⟩	::=	`derived`
⟨*DESC*⟩	::=	`desc`
⟨*DESCRIPTOR*⟩	::=	`descriptor`
⟨*DISTINCT*⟩	::=	`distinct`
⟨*DROP*⟩	::=	`drop`
⟨*ELSE*⟩	::=	`else`
⟨*END*⟩	::=	`end`
⟨*ENTITY*⟩	::=	`entity`
⟨*ESCAPE*⟩	::=	`escape`
⟨*EXCEPT*⟩	::=	`except`
⟨*EXISTS*⟩	::=	`exists`
⟨*EXP*⟩	::=	`exp`
⟨*EXTENT*⟩	::=	`extent`
⟨*FALSE*⟩	::=	`false`
⟨*FLOAT*⟩	::=	`float`
⟨*FLOOR*⟩	::=	`floor`
⟨*FOR*⟩	::=	`for`
⟨*FOREIGN*⟩	::=	`foreign`
⟨*FROM*⟩	::=	`from`
⟨*FULL*⟩	::=	`full`
⟨*GROUP BY*⟩	::=	`group by`
⟨*HAVING*⟩	::=	`having`
⟨*IN*⟩	::=	`in`
⟨*INNER*⟩	::=	`inner`
⟨*INSERT*⟩	::=	`insert`
⟨*INT*⟩	::=	`int`
⟨*INTEGER*⟩	::=	`integer`
⟨*INTERSECT*⟩	::=	`intersect`
⟨*INTO*⟩	::=	`into`
⟨*IS*⟩	::=	`is`
⟨*JOIN*⟩	::=	`join`
⟨*KEY*⟩	::=	`key`
⟨*LANGUAGE*⟩	::=	`language`
⟨*LEFT*⟩	::=	`left`
⟨*LIKE*⟩	::=	`like`
⟨*LN*⟩	::=	`ln`
⟨*LOWER*⟩	::=	`lower`
⟨*MAX*⟩	::=	`max`
⟨*MIN*⟩	::=	`min`
⟨*MOD*⟩	::=	`mod`
⟨*MULTILINGUAL*⟩	::=	`multilingual`
⟨*NAMESPACE*⟩	::=	`namespace`
⟨*NATURAL*⟩	::=	`natural`
⟨*NONE*⟩	::=	`none`
⟨*NOT*⟩	::=	`not`
⟨*NULL*⟩	::=	`null`
⟨*NULLIF*⟩	::=	`nullif`
⟨*OF*⟩	::=	`of`
⟨*ON*⟩	::=	`on`
⟨*ONLY*⟩	::=	`only`
⟨*OR*⟩	::=	`or`
⟨*ORDER BY*⟩	::=	`order by`
⟨*OUTER*⟩	::=	`outer`
⟨*POWER*⟩	::=	`power`
⟨*PRIMARY*⟩	::=	`primary`
⟨*PROPERTY*⟩	::=	`property`
⟨*REAL*⟩	::=	`real`
⟨*REF*⟩	::=	`ref`

⟨*REFERENCES*⟩ ::= `references`
⟨*RIGHT*⟩ ::= `right`
⟨*SELECT*⟩ ::= `select`
⟨*SET*⟩ ::= `set`
⟨*SIMILAR*⟩ ::= `similar`
⟨*SOME*⟩ ::= `some`
⟨*SQRT*⟩ ::= `sqrt`
⟨*STRING*⟩ ::= `string`
⟨*SUBSTRING*⟩ ::= `substring`
⟨*SUM*⟩ ::= `sum`
⟨*TABLE*⟩ ::= `table`
⟨*THEN*⟩ ::= `then`
⟨*TREAT*⟩ ::= `treat`
⟨*TRUE*⟩ ::= `true`
⟨*TYPEOF*⟩ ::= `typeof`
⟨*UNDER*⟩ ::= `under`
⟨*UNION*⟩ ::= `union`
⟨*UNIQUE*⟩ ::= `unique`
⟨*UNNEST*⟩ ::= `unnest`
⟨*UPDATE*⟩ ::= `update`
⟨*UPPER*⟩ ::= `upper`
⟨*USING*⟩ ::= `using`
⟨*VALUES*⟩ ::= `values`
⟨*VARCHAR*⟩ ::= `varchar`
⟨*VIEW*⟩ ::= `view`
⟨*WHEN*⟩ ::= `when`
⟨*WHERE*⟩ ::= `where`

Afin d'améliorer la lisibilité de la grammaire, nous écrivons ces mots clés comme des éléments terminaux en majuscule dans les sections suivantes.

2.2 Les éléments lexicaux

Les règles suivantes indiquent comment certaines combinaisons de caractères sont interprétées comme des éléments lexicaux dans le langage. Les nombres :

⟨*unsigned numeric literal*⟩ ::= ⟨*exact numeric literal*⟩
  | ⟨*approximate numeric literal*⟩
⟨*exact numeric literal*⟩ ::= ⟨*unsigned integer*⟩ [`.` [⟨*unsigned integer*⟩]]
  | `.` ⟨*unsigned integer*⟩
⟨*unsigned integer*⟩ ::= ⟨*digit*⟩ { ⟨*digit*⟩ }
⟨*digit*⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨*approximate numeric literal*⟩ ::= ⟨*exact numeric literal*⟩ `E` ⟨*signed integer*⟩
⟨*signed integer*⟩ ::= [⟨*sign*⟩] ⟨*unsigned integer*⟩
⟨*sign*⟩ ::= `+` | `-`

Les chaînes de caractères, les dates et les booléens :

⟨*general literal*⟩ ::= ⟨*character string literal*⟩
  | ⟨*date literal*⟩
  | ⟨*boolean literal*⟩
⟨*character string literal*⟩ ::= `'` [⟨*character representation list*⟩] `'`
⟨*character representation*⟩ ::= ⟨*nonquote character*⟩
  | ⟨*quote symbol*⟩

⟨nonquote character⟩ ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

⟨quote symbol⟩ ::= ''

⟨date literal⟩ ::= DATE ' *⟨date value⟩* '

⟨date value⟩ ::= *⟨unsigned integer⟩* - *⟨unsigned integer⟩* - *⟨unsigned integer⟩*

⟨boolean literal⟩ ::= TRUE | FALSE

L'élément lexical *identifier* utilisé pour référencer les différents éléments manipulés par le langage OntoQL :

⟨identifier⟩ ::= *⟨identifier start⟩* { *⟨identifier part⟩* }

⟨identifier part⟩ ::= *⟨identifier start⟩*
| *⟨identifier extend⟩*

⟨identifier start⟩ ::= _ | *⟨nonquote character⟩*

⟨identifier extend⟩ ::= $ | *⟨digit⟩*

2.3 Les identifiants interprétés

Les règles suivantes définissent les identifiants des différents éléments manipulés par le langage OntoQL :

⟨table name⟩ ::= *⟨identifier⟩*

⟨column name⟩ ::= *⟨identifier⟩*

⟨constraint name⟩ ::= *⟨identifier⟩*

⟨alias name⟩ ::= *⟨identifier⟩*

⟨function name⟩ ::= *⟨identifier⟩*

⟨class id⟩ ::= *⟨identifier⟩*

⟨property id⟩ ::= *⟨identifier⟩*

⟨entity id⟩ ::= # *⟨identifier⟩*

⟨attribute id⟩ ::= # *⟨identifier⟩*

⟨category id⟩ ::= *⟨table name⟩*
| *⟨class id⟩*
| *⟨entity id⟩*

⟨category id polymorph⟩ ::= *⟨table name⟩*
| *⟨class id⟩* | ONLY (*⟨class id⟩*)
| *⟨entity id⟩* | ONLY (*⟨entity id⟩*)

⟨description id⟩ ::= *⟨column name⟩*
| *⟨property id⟩*
| *⟨attribute id⟩*

⟨namespace id⟩ ::= *⟨identifier⟩*

⟨namespace alias⟩ ::= *⟨identifier⟩*

⟨*language id*⟩ ::= AA | AB | AF | AM | AR | AS | AY | AZ | BA | BE | BG | BH | BI | BN | BO | BR | CA | CO | CS | CY | DA | DE | DZ | EL | EN | EO | ES | ET | EU | FA | FI | FJ | FO | FR | FY | GA | GD | GL | GN | GU | HA | HI | HR | HU | HY | IA | IE | IK | IN | IS | IT | IW | JA | JI | JW | KA | KK | KL | KM | KN | KO | KS | KU | KY | LA | LN | LO | LT | LV | MG | MI | MK | ML | MN | MO | MR | MS | MT | MY | NA | NE | NL | NO | OC | OM | OR | PA | PL | PS | PT | QU | RM | RN | RO | RU | RW | SA | SD | SG | SH | SI | SK | SL | SM | SN | SO | SQ | SR | SS | ST | SU | SV | SW | TA | TE | TG | TH | TI | TK | TL | TN | TO | TR | TS | TT | TW | UK | UR | UZ | VI | VO | WO | XH | YO | ZH | ZU

3 Les ressources

Dans cette section, nous définissons les éléments de la grammaire utilisés par les différents langages proposés par le langage OntoQL.

3.1 Les types de données

⟨*data type*⟩ ::= ⟨*predefined type*⟩
| ⟨*reference type*⟩
| ⟨*collection type*⟩
⟨*predefined type*⟩ ::= ⟨*character string type*⟩
| ⟨*numeric type*⟩
| ⟨*boolean type*⟩
| ⟨*date type*⟩
⟨*character string type*⟩ ::= [MULTILINGUAL] STRING [(⟨*integer*⟩)]
| [MULTILINGUAL] VARCHAR (⟨*integer*⟩)
⟨*numeric type*⟩ ::= ⟨*exact numeric type*⟩
| ⟨*approximate numeric type*⟩
⟨*exact numeric type*⟩ ::= INT
| INTEGER
⟨*approximate numeric type*⟩ ::= FLOAT [(⟨*integer*⟩)]
| REAL
⟨*boolean type*⟩ ::= BOOLEAN
⟨*date type*⟩ ::= DATE
⟨*reference type*⟩ ::= REF (⟨*referenced type*⟩)
⟨*referenced type*⟩ ::= ⟨*class id*⟩
| ⟨*entity id*⟩
⟨*collection type*⟩ ::= ⟨*data type*⟩ ARRAY [[⟨*integer*⟩]]

3.2 Les valeurs

Les règles suivantes définissent la syntaxe des éléments définissant des valeurs des types de données présentés précédemment.

⟨*value expression*⟩ ::= ⟨*numeric value expression*⟩
| ⟨*string value expression*⟩
| ⟨*collection value expression*⟩
| ⟨*boolean value expression*⟩

Les valeurs entières :

⟨*numeric value expression*⟩ ::= ⟨*term*⟩
| ⟨*numeric value expression*⟩ + ⟨*term*⟩
| ⟨*numeric value expression*⟩ - ⟨*term*⟩
⟨*term*⟩ ::= ⟨*factor*⟩
| ⟨*term*⟩ * ⟨*factor*⟩
| ⟨*term*⟩ / ⟨*factor*⟩
⟨*factor*⟩ ::= [-] ⟨*numeric primary*⟩
⟨*numeric primary*⟩ ::= ⟨*unsigned numeric literal*⟩
| ⟨*value expression primary*⟩
| ⟨*numeric value function*⟩
⟨*numeric value function*⟩ ::= ⟨*cardinality expression*⟩
| ⟨*absolute value expression*⟩
| ⟨*modulus expression*⟩
| ⟨*natural logarithm*⟩
| ⟨*exponential function*⟩
| ⟨*power function*⟩
| ⟨*square root*⟩
| ⟨*floor function*⟩
⟨*cardinality expression*⟩ ::= CARDINALITY (⟨*collection value expression*⟩)
⟨*absolute value expression*⟩ ::= ABS (⟨*numeric value expression*⟩)
⟨*modulus expression*⟩ ::= MOD (⟨*numeric value expression*⟩, ⟨*numeric value expression*⟩)
⟨*natural logarithm*⟩ ::= LN (⟨*numeric value expression*⟩)
⟨*exponential function*⟩ ::= EXP (⟨*numeric value expression*⟩)
⟨*power function*⟩ ::= POWER (⟨*numeric value expression*⟩,
⟨*numeric value expression*⟩)
⟨*square root*⟩ ::= SQRT (⟨*numeric value expression*⟩)
⟨*floor function*⟩ ::= FLOOR (⟨*numeric value expression*⟩)

Les valeurs de type chaînes de caractères :

⟨*string value expression*⟩ ::= ⟨*concatenation*⟩ | ⟨*character factor*⟩
⟨*concatenation*⟩ ::= ⟨*string value expression*⟩ || ⟨*character factor*⟩
⟨*character factor*⟩ ::= ⟨*character primary*⟩
⟨*character primary*⟩ ::= ⟨*character string literal*⟩
| ⟨*value expression primary*⟩
| ⟨*string value function*⟩

⟨*string value function*⟩ ::= ⟨*character substring function*⟩
| ⟨*regexpr substring function*⟩
| ⟨*fold*⟩
⟨*character substring function*⟩ ::= `SUBSTRING (` ⟨*string value expression*⟩ `FROM` ⟨*numeric value expression*⟩ [`FOR` ⟨*numeric value expression*⟩] `)`
⟨*regexpr substring function*⟩ ::= `SUBSTRING (` ⟨*string value expression*⟩ `SIMILAR` ⟨*string value expression*⟩ `ESCAPE` ⟨*string value expression*⟩ `)`
⟨*fold*⟩ ::= ⟨*fold op*⟩ `(` ⟨*string value expression*⟩ `)`
⟨*fold op*⟩ ::= `UPPER` | `LOWER`

Les valeurs booléennes :

⟨*boolean value expression*⟩ ::= ⟨*boolean term*⟩
| ⟨*boolean value expression*⟩ `OR` ⟨*boolean term*⟩
⟨*boolean term*⟩ ::= ⟨*boolean factor*⟩
| ⟨*boolean term*⟩ `AND` ⟨*boolean factor*⟩
⟨*boolean factor*⟩ ::= [`NOT`] ⟨*boolean test*⟩
⟨*boolean test*⟩ ::= ⟨*predicate*⟩ | ⟨*boolean predicand*⟩
⟨*predicate*⟩ ::= ⟨*comparison predicate*⟩
| ⟨*between predicate*⟩
| ⟨*in predicate*⟩
| ⟨*like predicate*⟩
| ⟨*null predicate*⟩
| ⟨*quantified predicate*⟩
| ⟨*exists predicate*⟩
| ⟨*type predicate*⟩
⟨*comparison predicate*⟩ ::= ⟨*value expression*⟩ ⟨*equality op*⟩ ⟨*value expression*⟩
⟨*equality op*⟩ ::= = | <> | > | >= | < | <=
⟨*between predicate*⟩ ::= ⟨*value expression*⟩ [`NOT`] `BETWEEN` ⟨*value expression*⟩ `AND` ⟨*value expression*⟩
⟨*in predicate*⟩ ::= [`NOT`] `IN` ⟨*in predicate value*⟩
⟨*in predicate value*⟩ ::= ⟨*subquery*⟩ | `(` ⟨*value expression list*⟩ `)`
⟨*like predicate*⟩ ::= ⟨*value expression*⟩ [`NOT`] `LIKE` ⟨*value expression*⟩
⟨*null predicate*⟩ ::= ⟨*value expression*⟩ `IS` [`NOT`] `NULL`
⟨*quantified predicate*⟩ ::= ⟨*value expression*⟩ ⟨*quantifier op*⟩ `(` ⟨*sub query*⟩ `)`
⟨*quantifier op*⟩ ::= `ALL` | `SOME` | `ANY`
⟨*exists predicate*⟩ ::= `EXISTS` ⟨*subquery*⟩
⟨*type predicate*⟩ ::= ⟨*value expression*⟩ `IS` [`NOT`] `OF (` ⟨*type list*⟩ `)`
⟨*type list*⟩ ::= [`ONLY`] ⟨*is of type*⟩ { `,` [`ONLY`] ⟨*is of type*⟩ }
⟨*is of type*⟩ ::= ⟨*reference type*⟩ | ⟨*class id*⟩ | ⟨*entity id*⟩

⟨*boolean predicand*⟩ ::= (⟨*boolean value expression*⟩)
  | ⟨*boolean literal*⟩
  | ⟨*value expression primary*⟩

Les valeurs de type collection :

⟨*collection value expression*⟩ ::= ⟨*array concatenation*⟩
  | ⟨*array primary*⟩
⟨*array concatenation*⟩ ::= ⟨*collection value expression*⟩ || ⟨*array primary*⟩
⟨*array primary*⟩ ::= ⟨*value expression primary*⟩
  | ⟨*array value constructor*⟩
⟨*array value constructor*⟩ ::= ⟨*array value constructor by enumeration*⟩
  | ⟨*array value constructor by query*⟩
⟨*array value constructor by enumeration*⟩ ::= ARRAY [⟨*value expression list*⟩]
⟨*array value constructor by query*⟩ ::= ARRAY (⟨*query expression*⟩)

Expressions produisant des valeurs de différents types :

⟨*value expression primary*⟩ ::= ⟨*par value expression*⟩
  | ⟨*nonpar value expression primary*⟩
⟨*par value expression*⟩ ::= (⟨*value expression*⟩)
⟨*nonpar value expression primary*⟩ ::= ⟨*description reference*⟩
  | ⟨*scalar subquery*⟩
  | ⟨*function call*⟩
  | ⟨*aggregate function*⟩
  | ⟨*case expression*⟩
  | ⟨*cast specification*⟩
  | ⟨*subtype treatment*⟩
  | ⟨*typeof treatment*⟩
  | ⟨*reference resolution*⟩
  | ⟨*null specification*⟩
⟨*description reference*⟩ ::= [⟨*identifier*⟩ .] ⟨*qualified description*⟩
⟨*qualified description*⟩ ::= ⟨*column name*⟩
  | ⟨*property path expression*⟩
  | ⟨*attribute path expression*⟩
  | ⟨*identifier*⟩
⟨*property path expression*⟩ ::= ⟨*property id*⟩ { . ⟨*property id*⟩ }
⟨*attribute path expression*⟩ ::= ⟨*attribute id*⟩ { . ⟨*attribute id*⟩ }
⟨*scalar subquery*⟩ ::= ⟨*subquery*⟩
⟨*function call*⟩ ::= ⟨*function name*⟩ ([⟨*value expression list*⟩])
⟨*aggregate function*⟩ ::= COUNT (*) | ⟨*general set function*⟩
⟨*general set function*⟩ ::= ⟨*computational operation*⟩
  ([⟨*set quantifier*⟩] ⟨*value expression*⟩)

⟨*computational operation*⟩ ::= AVG | MAX | MIN | SUM | COUNT

⟨*set quantifier*⟩ ::= DISTINCT | ALL

⟨*case expression*⟩ ::= ⟨*case abbreviation*⟩
| ⟨*case specification*⟩

⟨*case abbreviation*⟩ ::= NULLIF (⟨*value expression*⟩ , ⟨*value expression*⟩)
| COALESCE (⟨*value expression list*⟩)

⟨*case specification*⟩ ::= ⟨*simple case*⟩
| ⟨*searched case*⟩

⟨*simple case*⟩ ::= CASE ⟨*value expression*⟩ ⟨*simple when clause list*⟩
[⟨*else clause*⟩] END

⟨*simple when clause*⟩ ::= WHEN ⟨*value expression*⟩ THEN ⟨*value expression*⟩

⟨*else clause*⟩ ::= ELSE ⟨*value expression*⟩

⟨*searched case*⟩ ::= CASE ⟨*searched when clause list*⟩ [⟨*else clause*⟩] END

⟨*searched when clause*⟩ ::= WHEN ⟨*search condition*⟩ THEN ⟨*value expression*⟩

⟨*search condition*⟩ ::= ⟨*boolean value expression*⟩

⟨*cast specification*⟩ ::= CAST (⟨*value expression*⟩ AS ⟨*data type*⟩)

⟨*subtype treatment*⟩ ::= TREAT (⟨*value expression*⟩ AS ⟨*target subtype*⟩)

⟨*target subtype*⟩ ::= ⟨*reference type*⟩ | ⟨*class id*⟩ | ⟨*entity id*⟩

⟨*typeof treatment*⟩ ::= TYPEOF (⟨*value expression*⟩)

⟨*reference resolution*⟩ ::= DEREF (⟨*value expression*⟩)

⟨*null specification*⟩ ::= NULL

4 Langage de définition de données : LDD et LDO

Les règles suivantes définissent la syntaxe des langages de définition de données du langage OntoQL. Le LDD au niveau logique :

⟨*table definition*⟩ ::= CREATE TABLE ⟨*table name*⟩ ⟨*table element list*⟩

⟨*table element*⟩ ::= ⟨*column definition*⟩
| ⟨*table constraint definition*⟩

⟨*column definition*⟩ ::= ⟨*column name*⟩ ⟨*data type*⟩
{ ⟨*column constraint definition*⟩ }

⟨*column constraint definition*⟩ ::= NOT NULL
| ⟨*unique specification*⟩
| ⟨*references specification*⟩
| ⟨*check constraint definition*⟩

⟨*unique specification*⟩ ::= UNIQUE | PRIMARY KEY

⟨*references specification*⟩ ::= REFERENCES ⟨*table name*⟩ [(⟨*column name list*⟩)]

⟨*check constraint definition*⟩ ::= CHECK (⟨*search condition*⟩)

⟨*table constraint definition*⟩ ::= [⟨*constraint name definition*⟩] ⟨*table constraint*⟩

⟨*constraint name definition*⟩ ::= `CONSTRAINT` ⟨*constraint name*⟩

⟨*table constraint*⟩ ::= ⟨*unique constraint definition*⟩
| ⟨*referential constraint definition*⟩
| ⟨*check constraint definition*⟩

⟨*unique constraint definition*⟩ ::= ⟨*unique specification*⟩ `(` ⟨*column name list*⟩ `)`

⟨*referential constraint definition*⟩ ::= `FOREIGN KEY (` ⟨*column name list*⟩ `)` ⟨*references specification*⟩

⟨*alter table statement*⟩ ::= `ALTER TABLE` ⟨*table name*⟩ ⟨*alter table action*⟩

⟨*alter table action*⟩ ::= ⟨*add column definition*⟩
| ⟨*drop column definition*⟩
| ⟨*add table constraint definition*⟩
| ⟨*drop table constraint definition*⟩

⟨*add column definition*⟩ ::= `ADD` ⟨*column definition*⟩

⟨*drop column definition*⟩ ::= `DROP` ⟨*column name*⟩

⟨*add table constraint definition*⟩ ::= `ADD` ⟨*table constraint definition*⟩

⟨*drop table constraint definition*⟩ ::= `DROP CONSTRAINT` ⟨*constraint name*⟩

⟨*drop table statement*⟩ ::= `DROP TABLE` ⟨*table name*⟩

Le LDD au niveau ontologique :

⟨*class definition*⟩ ::= `CREATE` ⟨*entity id*⟩ ⟨*class id*⟩ [⟨*view clause*⟩] [⟨*under clause*⟩]
[⟨*descriptor clause*⟩] [⟨*properties clause list*⟩]

⟨*view clause*⟩ ::= `AS VIEW`

⟨*under clause*⟩ ::= `UNDER` ⟨*class id list*⟩

⟨*descriptor clause*⟩ ::= `DESCRIPTOR (` ⟨*attribute value list*⟩ `)`

⟨*attribute value*⟩ ::= ⟨*attribute id*⟩ = ⟨*value expression*⟩

⟨*properties clause*⟩ ::= ⟨*entity id*⟩ `(` ⟨*property definition list*⟩ `)`

⟨*property definition*⟩ ::= ⟨*prop id*⟩ ⟨*datatype*⟩ [⟨*descriptor clause*⟩]

⟨*alter class statement*⟩ ::= `ALTER` ⟨*class id*⟩ [⟨*descriptor clause*⟩]
[⟨*alter class action*⟩]

⟨*alter class action*⟩ ::= ⟨*add property definition*⟩
| ⟨*drop property definition*⟩

⟨*add property definition*⟩ ::= `ADD` [⟨*entity id*⟩] ⟨*property definition*⟩ [⟨*descriptor clause*⟩]

⟨*drop property definition*⟩ ::= `DROP` ⟨*property id*⟩

⟨*drop class definition*⟩ ::= `DROP` ⟨*class id*⟩

Il permet également de définir les extensions des classes :

⟨*extension definition*⟩ ::= `CREATE EXTENT OF` ⟨*class id*⟩ `(` ⟨*property id list*⟩ `)` [⟨*logical clause*⟩]

⟨*logical clause*⟩ ::= `TABLE` [⟨*table and column name*⟩]

⟨*table and column name*⟩ ::= ⟨*table name*⟩ [`(` ⟨*column name list*⟩ `)`]

⟨*alter extension statement*⟩ ::= `ALTER EXTENT OF` ⟨*class id*⟩ ⟨*alter extent action*⟩

⟨*alter extension action*⟩ ::= ⟨*add property definition*⟩
| ⟨*drop property definition*⟩
⟨*add property definition*⟩ ::= ADD [PROPERTY] ⟨*property id*⟩ [COLUMN ⟨*column name*⟩]
⟨*drop property definition*⟩ ::= DROP [PROPERTY] ⟨*property id*⟩
⟨*drop extension statement*⟩ ::= DROP EXTENT OF ⟨*class id*⟩

Le LDO pour les ontologies :

⟨*entity definition*⟩ ::= CREATE ENTITY ⟨*entity id*⟩ [⟨*under clause*⟩] ⟨*attribute clause*⟩
⟨*under clause*⟩ ::= UNDER ⟨*entity id list*⟩
⟨*attribute clause*⟩ ::= ⟨*attribute definition list*⟩
⟨*attribute definition*⟩ ::= ⟨*attribute id*⟩ ⟨*datatype*⟩ [⟨*derived clause*⟩]
⟨*derived clause*⟩ ::= DERIVED BY ⟨*function name*⟩
⟨*alter entity statement*⟩ ::= ALTER ENTITY ⟨*entity id*⟩ ⟨*alter entity action*⟩
⟨*alter entity action*⟩ ::= ⟨*add attribute definition*⟩
| ⟨*drop attribute definition*⟩
⟨*add attribute definition*⟩ ::= ADD [ATTRIBUTE] ⟨*attribute definition*⟩
⟨*drop attribute definition*⟩ ::= DROP [ATTRIBUTE] ⟨*attribute id*⟩
⟨*drop entity statement*⟩ ::= DROP ENTITY ⟨*entity id*⟩

5 Langage de manipulation de données : LMD et LMO

Les règles suivantes définissent la syntaxe des langages de manipulation de données du langage OntoQL, c'est-à-dire le LMD niveau logique et ontologique et le LMO. La syntaxe de ces différents langages est uniforme. Ainsi, ces différents langages sont définis par des règles communes :

⟨*insert statement*⟩ ::= INSERT INTO ⟨*category id*⟩ ⟨*insert description and source*⟩
⟨*insert description and source*⟩ ::= ⟨*from subquery*⟩ | ⟨*from constructor*⟩
⟨*from subquery*⟩ ::= [(⟨*insert description list*⟩)] ⟨*query expression*⟩
⟨*insert description list*⟩ ::= ⟨*column name list*⟩
| ⟨*property id list*⟩
| ⟨*attribute id list*⟩
⟨*from constructor*⟩ ::= [(⟨*insert description list*⟩)] ⟨*values clause*⟩
⟨*values clause*⟩ ::= VALUES (⟨*values expression list*⟩)
⟨*update statement*⟩ ::= UPDATE ⟨*category id polymorph*⟩ SET ⟨*set clause list*⟩
[WHERE ⟨*search condition*⟩]
⟨*set clause*⟩ ::= ⟨*description id*⟩ = ⟨*value expression*⟩
⟨*delete statement*⟩ ::= DELETE FROM ⟨*category id polymorph*⟩
[WHERE ⟨*search condition*⟩]

6 Langage d'interrogation de données : LID et LIO

Les règles suivantes définissent la syntaxe des langages d'interrogation du langage OntoQL, c'est-à-dire le LID niveau logique et ontologique et le LIO. A nouveau, nous avons défini des règles communes pour ces différents langages. Par contre, nous avons décomposé cette syntaxe pour en faciliter la compréhension.

La forme générale d'une requête :

⟨query expression⟩ ::= *⟨query term⟩*
| *⟨query expression⟩* `UNION` *⟨set quantifier⟩* *⟨query term⟩*
| *⟨query expression⟩* `EXCEPT` *⟨set quantifier⟩* *⟨query term⟩*

⟨query term⟩ ::= *⟨query primary⟩*
| *⟨query term⟩* `INTERSECT` *⟨set quantifier⟩* *⟨query primary⟩*

⟨query primary⟩ ::= *⟨query specification⟩* | `(` *⟨query expression⟩* `)`

⟨query specification⟩ ::= *⟨select clause⟩* *⟨from clause⟩* [*⟨where clause⟩*]
[*⟨group by clause⟩*] [*⟨having clause⟩*] [*⟨order by clause⟩*]
[*⟨namespace clause⟩*] [*⟨language clause⟩*]

La clause `SELECT` :

⟨select clause⟩ ::= `SELECT` [*⟨set quantifier⟩*] *⟨select list⟩*

⟨select list⟩ ::= * | *⟨select sublist⟩* `{ ,` *⟨select sublist⟩* }

⟨select sublist⟩ ::= *⟨value expression⟩* [*⟨as clause⟩*]

⟨as clause⟩ ::= [`AS`] *⟨alias name⟩*

La clause `FROM` :

⟨from clause⟩ ::= `FROM` *⟨category reference list⟩*

⟨category reference⟩ ::= *⟨category primary⟩*
| *⟨joined category⟩*

⟨category primary⟩ ::= *⟨category or subquery⟩* [[`AS`] *⟨alias name⟩*]
| *⟨collection derived category⟩* [`AS`] *⟨alias name⟩*

⟨category or subquery⟩ ::= *⟨category id polymorph⟩*
| *⟨dynamic iterator⟩*
| *⟨subquery⟩*

⟨dynamic iterator⟩ ::= *⟨identifier⟩* | `ONLY (` *⟨identifier⟩* `)`

⟨subquery⟩ ::= `(` *⟨query expression⟩* `)`

⟨collection derived category⟩ ::= `UNNEST (` *⟨collection value expression⟩* `)`

⟨joined category⟩ ::= *⟨cross join⟩*
| *⟨qualified join⟩*
| *⟨natural join⟩*

⟨cross join⟩ ::= *⟨category reference⟩* `CROSS JOIN` *⟨category primary⟩*

⟨qualified join⟩ ::= *⟨category reference⟩*[*⟨join type⟩*] `JOIN`
⟨categroy reference⟩ *⟨join specification⟩*

⟨*join type*⟩	::=	`INNER` | ⟨*outer join type*⟩ [`OUTER`]
⟨*outer join type*⟩	::=	`LEFT` | `RIGHT` | `FULL`
⟨*join specification*⟩	::=	⟨*join condition*⟩
	|	⟨*named columns join*⟩
⟨*join condition*⟩	::=	`ON` ⟨*search condition*⟩
⟨*named columns join*⟩	::=	`USING` `(` ⟨*description id list*⟩ `)`
⟨*natural join*⟩	::=	⟨*category reference*⟩ `NATURAL` [⟨*join type*⟩] `JOIN` ⟨*category primary*⟩

Les clauses `WHERE`, `GROUP BY`, `HAVING` et `ORDER BY` :

⟨*where clause*⟩	::=	`WHERE` ⟨*search condition*⟩
⟨*group by clause*⟩	::=	`GROUP BY` [⟨*set quantifier*⟩] ⟨*description id list*⟩
⟨*having clause*⟩	::=	`HAVING` ⟨*search condition*⟩
⟨*order by*⟩	::=	`ORDER BY` ⟨*sort specification list*⟩
⟨*sort specification*⟩	::=	⟨*sort key*⟩ [⟨*ordering specification*⟩]
⟨*sort key*⟩	::=	⟨*value expression*⟩
⟨*ordering specification*⟩	::=	`ASC` | `DESC`

Les clauses `NAMESPACE` et `LANGUAGE` :

⟨*namespace clause*⟩	::=	`USING NAMESPACE` ⟨*namespace definition list*⟩
⟨*namespace definition*⟩	::=	[⟨*namespace alias*⟩ =] ⟨*namespace id*⟩
⟨*language clause*⟩	::=	`USING LANGUAGE` ⟨*language id*⟩

7 Langage de définition de vues : LDV

Les règles suivantes définissent la syntaxe pour définir des vues avec le langage OntoQL.

⟨*view definition*⟩	::=	`CREATE VIEW` ⟨*table name*⟩ ⟨*view specification*⟩ `AS` ⟨*query expression*⟩
⟨*view specification*⟩	::=	⟨*regular view specification*⟩
	|	⟨*referenceable view specification*⟩
⟨*regular view specification*⟩	::=	[`(` ⟨*column name list*⟩ `)`]
⟨*referenceable view specification*⟩	::=	`OF` ⟨*class id*⟩ [⟨*property id list*⟩]

8 Paramétrage du langage

Le langage OntoQL est paramétré par l'espace de noms dans lequel il doit rechercher les éléments d'une ontologie ainsi que par la langue naturelle dans laquelle il doit reconnaître les noms des différents éléments manipulés. La syntaxe permettant de paramétrer le langage OntoQL est la suivante :

⟨*global namespace definition*⟩ ::= SET NAMESPACE [⟨*namespace alias*⟩ =] ⟨*namespace specification*⟩

⟨*namespace specification*⟩ ::= ⟨*namespace id*⟩ | NONE

⟨*global language definition*⟩ ::= SET LANGUAGE ⟨*language specification*⟩

⟨*language specification*⟩ ::= ⟨*language id*⟩ | NONE

Annexe B

Comparaison du pouvoir d'expression de OntoQL avec des langages conçus pour RDF/RDF-Schema

Une comparaison du pouvoir d'expression de plusieurs langages pour RDF/RDF-Schema a été proposée dans [Haase et al., 2004] sur un échantillon de requêtes. Cette annexe présente l'ajout du langage OntoQL aux langages comparés.

1 L'exemple utilisé

La figure B.1 présente l'ontologie utilisée pour faire cette comparaison. Nous avons seulement représenté sur cette figure les éléments nécessaires pour exprimer les requêtes proposées. Nous avons également adapté cette ontologie aux différences entre le modèle de données de RDF-Schema et celui de OntoQL. Nous avons ainsi fait les modifications suivantes :

– la propriété `isAbout` a pour domaine `Publication`. Dans l'exemple proposé, le domaine de cette propriété n'était pas défini ;
– `labelP` et `labelT` sont deux propriétés. Dans l'exemple proposé, l'attribut `rdfs:label` était utilisé ;
– la propriété `namespace` a été ajoutée à la classe racine. Dans l'exemple proposé, toutes les instances ont par défaut un espace de noms ;
– `subTopic` est défini sur la classe `Topic` de cette ontologie. Dans l'exemple proposé, cette propriété était définie sur une classe d'un autre espace de noms. Nous avons fait cette modification uniquement pour simplifier la représentation de l'ontologie.

Cette ontologie décrit des concepts du domaine de la recherche scientifique. Elle définit la classe `Publication` décrite par les propriétés `title` et `labelP`. Chaque publication est faite sur un thème particulier. La propriété `isAbout` dont le codomaine est la classe `Topic` permet d'indiquer le thème auquel une publication est associée. Les thèmes sont organisés hiérarchiquement par la propriété `subTopic`. Une publication est également décrite par des auteurs. Cette relation est représentée par la propriété `author` de type collection (une séquence RDF-Schema) dont le codomaine est la classe `Person`. Cette classe est le domaine des propriétés `name` et `email`.

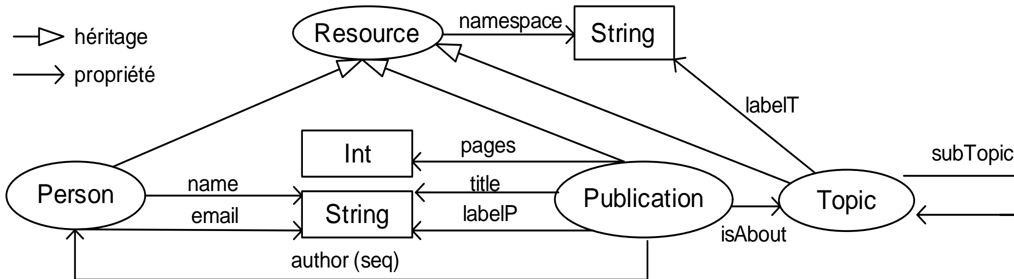


FIG. B.1 – Extrait de l'ontologie utilisée

2 Cas d'utilisation

Pour simplifier, l'expression des requêtes correspondant aux cas d'utilisation, l'espace de noms de l'ontologie exemple est définie comme étant celui par défaut grâce à l'instruction suivante :

```
SET NAMESPACE http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/sample.rdf
```

Use case 1 (*Chemin*)

Retourner les noms des auteurs de la publication X. Les auteurs dont le nom n'est pas connu ne devront pas être retournés.

```
SELECT auteur.name
  FROM Publication AS publi,
       UNNEST(publi.authors) AS auteur
 WHERE publi.oid = X AND
       auteur.name IS NOT NULL
```

Use case 2 (*Chemin optionnel*)

Quels sont le nom et, si elles sont connues, les adresses email des auteurs de toutes les publications disponibles ?

```
SELECT auteur.name, auteur.email,
  FROM Publication AS publi,
       UNNEST(publi.author) AS auteur
 WHERE publi.oid = X AND
       auteur.name IS NOT NULL
```

Cette requête retourne la valeur NULL lorsque l'adrese email de l'auteur retourné n'est pas connue.

Use case 3 (*Union*)

Retourner les libellés de tous les thèmes (`Topic`*) et (union) les titres de toutes les publications.*

```
(SELECT labelT FROM Topic)
 UNION
(SELECT title FROM Publication)
```

Use case 4 (*Différence*)

Retourner les libellés de tous les thèmes (topics) qui ne sont pas des titres de publications.

```
(SELECT labelT FROM Topic)
INTERSECT
(SELECT title FROM Publication)
```

Use case 5 (*Quantification*)

Retourner les personnes qui sont les auteurs de toutes les publications.

```
SELECT person.oid
  FROM Person AS person
 WHERE NOT EXISTS
        (SELECT publi.oid,
           FROM Publication AS publi
          WHERE person.oid <> ANY(publi.author))
```

Use case 6 (*Aggrégat*)

Compter le nombre d'auteurs des publications.

```
SELECT CARDINALIRY(author) FROM Publication
```

Aucun cas d'utilisation proposé ne comprend les opérateurs de groupement et de tri car aucun des langages testés ne les supporte. OntoQL proposant ces opérateurs, nous avons ajouté le cas d'utilisation suivant.

Use case 7 (*Groupement et tri*)

Retourner le nom des auteurs qui ont participé à plus de 10 publications. Retourner ces noms par ordre alphabétique décroissant.

```
  SELECT person.name
    FROM Publication AS publi,
         Person AS person
   WHERE person.oid = ANY(publi.author)
GROUP BY person.oid, person.name
HAVING COUNT(publi.oid) > 10
ORDER BY person.name DESC
```

Use case 8 (*Récursion*)

> *Retourner tous les sous-thèmes du thème* `Information Système` *de manière récursive.*

OntoQL ne supporte pas l'opérateur `WITH RECURSIVE` de SQL. Il ne permet donc pas d'exprimer cette requête.

Use case 9 (*Réïfication*)

> *Retourner la personne qui a classifié la publication X.*

Le but est ici de tester le support de la réïfication. La réïfication permet en RDF de définir un triplet dont le sujet est un autre triplet. OntoQL ne la supporte pas.

Use case 10 (*Collection*)

> *Retourner le premier auteur de la publication X.*

```
SELECT author[1] FROM Publication
```

Use case 11 (*Namespaces*)

> *Retourner toutes les ressources (éléments des ontologies et instances) dont l'espace de noms commence par* `http://www.aifb.unikarlshruhe.de`.

```
(SELECT #oid
  FROM #concept
 WHERE #definedBy.#namespace LIKE "http://www.aifb.unikarlshruhe.de%")
UNION
(SELECT oid
  FROM Resource
 WHERE namespace LIKE "http://www.aifb.unikarlshruhe.de%")
```

Nous considérons que OntoQL ne répond que partiellement à ce cas d'utilisation car il requiert que la propriété `namespace` soit ajoutée à la classe `Resource` alors que ce n'est pas le cas des autres langages.

Use case 12 (*Langage*)

> *Retourner le libellé allemand du thème dont le libellé anglais est* `'Database Management'`.

```
SELECT labelP[DE] FROM Publication WHERE labelP[EN] = 'Database Management'
```

Use case 13 (*Littéraux*)

> *Retourner toutes les publications dont le nombre de page est la valeur lexicale* `'08'`.

```
SELECT * FROM Publication WHERE pages = '08'
```

Dans cette requête, OntoQL, comme SQL, réalise la conversion automatique de la chaîne de caractères `'08'` en entier.

Use case 14 (*Type de données*)

Retourner toutes les publications dont le nombre de page est la valeur entière `08`*.*

```
SELECT * FROM Publication WHERE pages = 08
```

Use case 15 (*Implication*)

Retourner toutes les instances directes et indirectes de la classe `Publication`*.*

```
SELECT * FROM Publication
```

Le but de ce cas d'utilisation était de tester si les langages supportent la transitivité de la relation de subsomption.

3 Synthèse

Le tableau B.1 présente la synthèse faite dans [Haase et al., 2004][41] à laquelle nous avons ajouté les résultats proposés dans la section précédente.

Use case	RDQL	Triple	SeRQL	Versa	N3	RQL	OntoQL
Chemin	●	●	●	●	●	●	●
Chemin optionnel	-	-	●	●	-	○	●
Union	-	●	-	●	●	●	●
Difference	-	-	-	○	-	●	●
Quantification	-	-	-	-	-	●	●
Aggrégat	-	-	-	●	●	●	●
Groupement et tri	-	-	-	-	-	-	●
Récursivité	-	●	-	●	●	-	-
Reïfication	○	○	●	○	-	○	-
Collection	○	○	○	○	○	○	●
Namespace	○	-	●	-	●	●	○
Langage	-	-	●	-	-	-	●
Littéraux	●	●	●	●	●	●	●
Type de données	○	-	●	-	-	-	●
Implication	-	○	●	-	○	●	●

TAB. B.1 – Comparaison des langages étudiés sur les cas d'utilisation

[41]Nous avons conservé les résultats publiés dans [Haase et al., 2004]. Le tableau présenté à l'URL `http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/` présente des résultats légèrement différents.

Annexe C

Détails sur la sémantique du langage OntoQL

1 Les opérateurs de *OntoAlgebra* permettant d'interroger les données d'une BDBO

OntoProject

Signature :

$$\mathtt{ADT_C} \times 2^{\mathtt{V_C}} \times 2^{\mathtt{String} \times \mathtt{Function}} \rightarrow \mathtt{TUPLE}[< (\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C}) >] \times 2^{\mathtt{V_C}}$$

Sémantique :

$$\begin{aligned}&\mathtt{OntoProject}(\mathtt{T}, \mathtt{I_t}, \{(\mathtt{A_1}, \mathtt{f_1}), \ldots, (\mathtt{A_n}, \mathtt{f_n})\}) = \\ &\quad (\mathtt{TUPLE}[< (\mathtt{A_1}, \mathtt{range}(\mathtt{f_1})), \ldots, (\mathtt{A_n}, \mathtt{range}(\mathtt{f_n})) >], \\ &\quad\quad \{< \mathtt{A_1} : \mathtt{OntoVal}(\mathtt{i}, \mathtt{f_1}), \ldots, \mathtt{A_n} : \mathtt{OntoVal}(\mathtt{i}, \mathtt{f_n}) > | \, \mathtt{i} \in \mathtt{I_t}\})\end{aligned}$$

OntoSelect

Signature :

$$\begin{aligned}&\mathtt{TUPLE}[< (\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C}) >] \times 2^{\mathtt{V_C}} \times \mathtt{Predicate} \\ &\quad \rightarrow \mathtt{TUPLE}[< (\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C}) >] \times 2^{\mathtt{V_C}}\end{aligned}$$

Sémantique :

$$\mathtt{OntoSelect}(\mathtt{T}, \mathtt{I_t}, \mathtt{pred}) = (\mathtt{T}, \{\mathtt{i} \,|\, \mathtt{i} \in \mathtt{I_t} \wedge \mathtt{pred}(\mathtt{i})\})$$

OntoOJoin

Signature :

$$\begin{aligned}&\mathtt{TUPLE}[< (\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C}) >] \times 2^{\mathtt{V_C}} \times \\ &\quad \mathtt{TUPLE}[< (\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C}) >] \times 2^{\mathtt{V_C}} \times \mathtt{Predicate} \\ &\quad\quad \rightarrow \mathtt{TUPLE}[< (\mathtt{String}, \mathtt{ADT_C}), \ldots, (\mathtt{String}, \mathtt{ADT_C}) >] \times 2^{\mathtt{V_C}}\end{aligned}$$

Sémantique :

$$\begin{aligned}&\texttt{OntoOJoin}(\texttt{TUPLE}[<(A_1,T_1),\ldots,(A_m,T_m)>], I_t,\\&\quad\texttt{TUPLE}[<(A_{m+1},T_{m+1}),\ldots,(A_n,T_n)], I_r, pred) =\\&(\texttt{TUPLE}[<(A_1,T_1),\ldots,(A_n,T_n)>],\\&\{<(A_1,t.A_1),\ldots,(A_m,t.A_m),(A_{m+1},r.A_{m+1})\ldots,(A_n,r.A_n)> \mid t\in I_t \wedge r\in I_r \wedge pred(t,r)\})\end{aligned}$$

OntoLeftOuterOJoin

Signature :

$$\begin{aligned}&\texttt{TUPLE}[<(\texttt{String},ADT_C),\ldots,(\texttt{String},ADT_C)>]\times 2^{V_C}\times\\&\quad\texttt{TUPLE}[<(\texttt{String},ADT_C),\ldots,(\texttt{String},ADT_C)>]\times 2^{V_C}\times \texttt{Predicate}\\&\quad\quad\rightarrow \texttt{TUPLE}[<(\texttt{String},ADT_C),\ldots,(\texttt{String},ADT_C)>]\times 2^{V_C}\end{aligned}$$

Sémantique :

$$\begin{aligned}&\texttt{OntoLeftOuterOJoin}(\texttt{TUPLE}[<(A_1,T_1),\ldots,(A_m,T_m)>], I_t,\\&\quad\texttt{TUPLE}[<(A_{m+1},T_{m+1}),\ldots,(A_n,T_n)], I_r, pred) =\\&(\texttt{TUPLE}[<(A_1,T_1),\ldots,(A_n,T_n)>],\\&\{<(A_1,t.A_1),\ldots,(A_m,t.A_m),(A_{m+1},r.A_{m+1}),\ldots,(A_n,r.A_n)> \mid t\in I_t \wedge r\in I_r \wedge pred(t,r)\}\cup\\&\{<(A_1,t.A_1),\ldots,(A_m,t.A_m),(A_{m+1},NULL),\ldots,(A_n,NULL)> \mid t\in I_t \wedge \forall r\in I_r.(\neg pred(t,r))\})\end{aligned}$$

OntoRightOuterOJoin

Signature :

$$\begin{aligned}&\texttt{TUPLE}[<(\texttt{String},ADT_C),\ldots,(\texttt{String},ADT_C)>]\times 2^{V_C}\times\\&\quad\texttt{TUPLE}[<(\texttt{String},ADT_C),\ldots,(\texttt{String},ADT_C)>]\times 2^{V_C}\times \texttt{Predicate}\\&\quad\quad\rightarrow \texttt{TUPLE}[<(\texttt{String},ADT_C),\ldots,(\texttt{String},ADT_C)>]\times 2^{V_C}\end{aligned}$$

Sémantique :

$$\begin{aligned}&\texttt{OntoRightOuterOJoin}(\texttt{TUPLE}[<(A_1,T_1),\ldots,(A_m,T_m)>], I_t,\\&\quad\texttt{TUPLE}[<(A_{m+1},T_{m+1}),\ldots,(A_n,T_n)], I_r, pred) =\\&(\texttt{TUPLE}[<(A_1,T_1),\ldots,(A_n,T_n)>],\\&\{<(A_1,t.A_1),\ldots,(A_m,t.A_m),(A_{m+1},r.A_{m+1}),\ldots,(A_n,r.A_n)> \mid t\in I_t \wedge r\in I_r \wedge pred(t,r)\}\cup\\&\{<(A_1,NULL),\ldots,(A_m,NULL),(A_{m+1},r.A_1),\ldots,(A_n,r.A_n)> \mid r\in I_r \wedge \forall t\in I_t.(\neg pred(t,r))\})\end{aligned}$$

OntoFullOuterOJoin

Signature :

$$\begin{aligned}&\texttt{TUPLE}[<(\texttt{String},ADT_C),\ldots,(\texttt{String},ADT_C)>]\times 2^{V_C}\times\\&\quad\texttt{TUPLE}[<(\texttt{String},ADT_C),\ldots,(\texttt{String},ADT_C)>]\times 2^{V_C}\times \texttt{Predicate}\\&\quad\quad\rightarrow \texttt{TUPLE}[<(\texttt{String},ADT_C),\ldots,(\texttt{String},ADT_C)>]\times 2^{V_C}\end{aligned}$$

Sémantique :

$$\begin{aligned}
&\mathtt{OntoFullOuterOJoin}(\mathtt{TUPLE}[<(\mathtt{A}_1,\mathtt{T}_1),\ldots,(\mathtt{A}_m,\mathtt{T}_m)>],\mathtt{I}_t,\\
&\qquad\mathtt{TUPLE}[<(\mathtt{A}_{m+1},\mathtt{T}_{m+1}),\ldots,(\mathtt{A}_n,\mathtt{T}_n)],\mathtt{I}_r,\mathtt{pred}) =\\
&(\mathtt{TUPLE}[<(\mathtt{A}_1,\mathtt{T}_1),\ldots,(\mathtt{A}_n,\mathtt{T}_n)>],\\
&\{<(\mathtt{A}_1,\mathtt{t.A}_1),\ldots,(\mathtt{A}_m,\mathtt{t.A}_m),(\mathtt{A}_{m+1},\mathtt{r.A}_{m+1}),\ldots,(\mathtt{A}_n,\mathtt{r.A}_n)> \mid \mathtt{t}\in\mathtt{I}_t\wedge\mathtt{r}\in\mathtt{I}_r\wedge\mathtt{pred}(\mathtt{t},\mathtt{r})\}\cup\\
&\{<(\mathtt{A}_1,\mathtt{t.A}_1),\ldots,(\mathtt{A}_m,\mathtt{t.A}_m),(\mathtt{A}_{m+1},\mathtt{NULL}),\ldots,(\mathtt{A}_n,\mathtt{NULL})> \mid \mathtt{t}\in\mathtt{I}_t\wedge\forall\mathtt{r}\in\mathtt{I}_r.(\neg\mathtt{pred}(\mathtt{t},\mathtt{r}))\})\cup\\
&\{<(\mathtt{A}_1,\mathtt{NULL}),\ldots,(\mathtt{A}_m,\mathtt{NULL}),(\mathtt{A}_{m+1},\mathtt{r.A}_1),\ldots,(\mathtt{A}_n,\mathtt{r.A}_n)> \mid \mathtt{r}\in\mathtt{I}_r\wedge\forall\mathtt{t}\in\mathtt{I}_t.(\neg\mathtt{pred}(\mathtt{t},\mathtt{r}))\})
\end{aligned}$$

OntoNest

Signature :

$$\begin{aligned}
&\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_c}\times\mathtt{String}\\
&\qquad\rightarrow\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_c}
\end{aligned}$$

Sémantique :

$$\begin{aligned}
&\mathtt{OntoNest}(\mathtt{TUPLE}[<(\mathtt{A}_1,\mathtt{T}_1),\ldots,(\mathtt{A}_i,\mathtt{T}_i),\ldots,(\mathtt{A}_n,\mathtt{T}_n)>],\mathtt{I}_t,\mathtt{A}_i) =\\
&\qquad(\mathtt{TUPLE}[<(\mathtt{A}_1,\mathtt{T}_1),\ldots,(\mathtt{A}_i,\mathtt{SET}[\mathtt{T}_i]),\ldots,(\mathtt{A}_n,\mathtt{T}_n)>],\\
&\qquad\{<\mathtt{A}_1:\mathtt{s.A}_1,\ldots,\mathtt{A}_i:\mathtt{t},\ldots,\mathtt{A}_n:\mathtt{s.A}_n> \mid \forall\mathtt{r}\in\mathtt{t}\;\exists\mathtt{s}\in\mathtt{I}_t.(\mathtt{s.A}_i=\mathtt{r})\})
\end{aligned}$$

Cet opérateur peut également prendre en paramètre une collection d'attributs de groupement.

OntoUnNest

Signature :

$$\begin{aligned}
&\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_c}\times\mathtt{String}\times\mathtt{String}\\
&\qquad\rightarrow\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_c}
\end{aligned}$$

Sémantique :

$$\begin{aligned}
&\mathtt{OntoUnNest}(\mathtt{TUPLE}[<(\mathtt{A}_1,\mathtt{T}_1),\ldots,(\mathtt{A}_i,\mathtt{SET}[\mathtt{T}_i]),\ldots,(\mathtt{A}_n,\mathtt{T}_n)>],\mathtt{I}_t,\mathtt{A}_i) =\\
&\qquad(\mathtt{TUPLE}[<(\mathtt{A}_1,\mathtt{T}_1),\ldots,(\mathtt{A}_i,\mathtt{T}_i),\ldots,(\mathtt{A}_n,\mathtt{T}_n)>],\\
&\qquad\{<\mathtt{A}_1:\mathtt{s.A}_1,\ldots,\mathtt{A}_i:\mathtt{t},\ldots,\mathtt{A}_n:\mathtt{s.A}_n> \mid \mathtt{s}\in\mathtt{I}_t\wedge\mathtt{t}\in\mathtt{s.A}_i\})
\end{aligned}$$

OntoUnion

Signature :

$$\begin{aligned}
&\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_c}\times\\
&\qquad\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_c}\\
&\qquad\qquad\rightarrow\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_c}
\end{aligned}$$

Sémantique :

$$\begin{aligned}
&\mathtt{OntoUnion}(\mathtt{TUPLE}[<(\mathtt{A}_1,\mathtt{T}_1),\ldots,(\mathtt{A}_n,\mathtt{T}_n)>],\mathtt{I}_t,\mathtt{TUPLE}[<(\mathtt{B}_1,\mathtt{T}_1),\ldots,(\mathtt{B}_n,\mathtt{T}_n)],\mathtt{I}_r) =\\
&\qquad(\mathtt{TUPLE}[<(\mathtt{A}_1,\mathtt{T}_1),\ldots,(\mathtt{A}_n,\mathtt{T}_n)>],\\
&\qquad\{<(\mathtt{A}_1,\mathtt{t.A}_1),\ldots,(\mathtt{A}_n,\mathtt{t.A}_n)>\cup<(\mathtt{A}_1,\mathtt{r.B}_1),\ldots,(\mathtt{A}_n,\mathtt{r.B}_n)> \mid \mathtt{t}\in\mathtt{I}_t\wedge\mathtt{r}\in\mathtt{I}_r\})
\end{aligned}$$

OntoIntersection

Signature :

$$\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_C}\times \mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_C} \rightarrow \mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_C}$$

Sémantique :

$$\mathtt{OntoIntersection}(\mathtt{TUPLE}[<(A_1,T_1),\ldots,(A_n,T_n)>],I_t,\mathtt{TUPLE}[<(B_1,T_1),\ldots,(B_n,T_n)],I_r) = (\mathtt{TUPLE}[<(A_1,T_1),\ldots,(A_n,T_n)>], \{<(A_1,t.A_1),\ldots,(A_n,t.A_n)> \cap <(A_1,r.B_1),\ldots,(A_n,r.B_n)> \mid t\in I_t \wedge r\in I_r\})$$

OntoDifference

Signature :

$$\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_C}\times \mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_C} \rightarrow \mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_C}$$

Sémantique :

$$\mathtt{OntoDifference}(\mathtt{TUPLE}[<(A_1,T_1),\ldots,(A_n,T_n)>],I_t,\mathtt{TUPLE}[<(B_1,T_1),\ldots,(B_n,T_n)],I_r) = (\mathtt{TUPLE}[<(A_1,T_1),\ldots,(A_n,T_n)>], \{<(A_1,t.A_1),\ldots,(A_n,t.A_n)> - <(A_1,r.B_1),\ldots,(A_n,r.B_n)> \mid t\in I_t \wedge r\in I_r\})$$

OntoDupEliminate

Signature :

$$\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_C} \rightarrow \mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_C}$$

Sémantique :

$$\mathtt{OntoDupEliminate}(T,I_t) = (T,I_r)$$

I_r est une collection sans doublon (`Set[T]`) construite à partir de la collection I_t.

OntoSort

Signature :

$$\mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_C}\times \mathtt{Function} \rightarrow \mathtt{TUPLE}[<(\mathtt{String},\mathtt{ADT_C}),\ldots,(\mathtt{String},\mathtt{ADT_C})>]\times 2^{V_C}$$

Sémantique :

$$\mathtt{OntoSort}(T,I_t,f) = (T,I_r)$$

$\mathtt{I_r}$ est une tableau (ARRAY[T]) construit à partir de la collection $\mathtt{I_t}$ qui respecte la condition suivante :

$\forall \mathtt{i},\mathtt{j}$ tels que $\mathtt{i} < \mathtt{j} < \mathtt{cardinality}(\mathtt{I_r})$ alors $\mathtt{f}(\mathtt{I_r}[\mathtt{i}]) < \mathtt{f}(\mathtt{I_r}[\mathtt{j}])$

Cet opérateur peut également prendre en paramètre une collection d'attributs selon lesquels le tri doit être effectué. Il peut également prendre en paramètre le sens du tri (ascendant ou descendant).

2 Expression algébrique d'une requête OntoQL

Le tableau suivant présente les règles permettant de construire l'expression algébrique d'une requête OntoQL sur les données. Dans ce tableau, C désigne une classe, $p_1, \ldots, p_n$ sont des propriétés et $f_1, \ldots, f_n$ sont des fonctions. Une expression OntoQL peut contenir une autre expression OntoQL notée $\underline{exp_i}$. $\underline{exp_1}$ et $\underline{exp_2}$ sont deux relations composées respectivement des propriétés p_{1i} et p_{2i}. Elles ont en commun les propriétés allant jusqu'à l'indice k.

Expressions OntoQL (exp)	**Expressions algébriques**
C ou FROM C	OntoProject $(C, ext^*(C), \{(p_1, p_1), \ldots, (p_n, p_n)\})$
ONLY(C) ou FROM ONLY(C)	OntoProject $(C, ext(C), \{(p_1, p_1), \ldots, (p_n, p_n)\})$
FROM $\underline{exp_1}$, $\underline{exp_2}$	OntoOJoin $(\underline{exp_1}, \underline{exp_2}, true)$
FROM $\underline{exp_1}$ CROSS JOIN $\underline{exp_2}$	OntoOJoin $(\underline{exp_1}, \underline{exp_2}, true)$
FROM $\underline{exp_1}$ INNER JOIN $\underline{exp_2}$ ON pred	OntoOJoin $(\underline{exp_1}, \underline{exp_2}, pred)$
FROM $\underline{exp_1}$ LEFT OUTER JOIN $\underline{exp_2}$ ON pred	OntoLeftOuterOJoin $(\underline{exp_1}, \underline{exp_2}, pred)$
FROM $\underline{exp_1}$ RIGHT OUTER JOIN $\underline{exp_2}$ ON pred	OntoRightOuterOJoin $(\underline{exp_1}, \underline{exp_2}, pred)$
FROM $\underline{exp_1}$ FULL OUTER JOIN $\underline{exp_2}$ ON pred	OntoFullOuterOJoin $(\underline{exp_1}, \underline{exp_2}, pred)$
FROM $\underline{exp_1}$ NATURAL JOIN $\underline{exp_2}$	OntoOJoin $(\underline{exp_1}, \underline{exp_2}, p_{11} = p_{21} \wedge \cdots \wedge p_{1k} = p_{2k})$
FROM $\underline{exp}$ UNNEST $\underline{exp}.p_1$	OntoUnNest $(\underline{exp}, p_1)$
SELECT * $\underline{exp}$	OntoProject $(\underline{exp}, \{(p_1, p_1), \ldots, (p_n, p_n)\})$
SELECT $f_1, \ldots, f_n$ $\underline{exp}$	OntoProject $(\underline{exp}, \{(f_1, f_1), \ldots, (f_n, f_n)\})$
DISTINCT $\underline{exp}$	OntoDupEliminate $(\underline{exp})$
$\underline{exp}$ WHERE pred	OntoSelect $(\underline{exp}, pred)$
$\underline{exp}$ GROUP BY $p_1, \ldots, p_k$	OntoNest $(\underline{exp}, \{p_{k+1}, \ldots, p_n\})$
$\underline{exp}$ HAVING pred	OntoSelect $(\underline{exp}, pred)$
$\underline{exp}$ ORDER BY $f_1, \ldots, f_n$	OntoSort $(\underline{exp}, \{f_1, \ldots, f_n\}, true)$
$\underline{exp}$ ORDER BY $f_1, \ldots, f_n$ ASC	OntoSort $(\underline{exp}, \{f_1, \ldots, f_n\}, true)$
$\underline{exp}$ ORDER BY $f_1, \ldots, f_n$ DESC	OntoSort $(\underline{exp}, \{f_1, \ldots, f_n\}, false)$
$\underline{exp_1}$ UNION $\underline{exp_2}$	OntoUnion $(\underline{exp_1}, \underline{exp_2})$
$\underline{exp_1}$ INTERSECT $\underline{exp_2}$	OntoIntersection $(\underline{exp_1}, \underline{exp_2})$
$\underline{exp_1}$ EXCEPT $\underline{exp_2}$	OntoDifference $(\underline{exp_1}, \underline{exp_2})$

Tab. C.1 – Règles permettant de construire l'expression algébrique d'une requête OntoQL

3 Règles d'équivalence sur l'algèbre *OntoAlgebra*

Voici une liste de règles d'équivalence que nous avons établies sur l'algèbre *OntoAlgebra* à partir de celles définies pour l'algèbre *Encore*. Pour définir ces règles, nous utilisons les éléments suivants :

- T et R sont deux types TUPLE dont les attributs sont nommés respectivement $A_1, \ldots, A_m$ et $A_m, \ldots, A_n$;
- I_t et I_r sont deux collections de tuples de type respectif T et R ;
- $pred_t$ est un prédicat portant sur les tuples de T. pred, $pred_1$ et $pred_2$ portent sur les tuples de T et de R.

La règle suivante permet de réaliser une sélection avant une jointure :

$$\mathtt{OntoSelect}(\mathtt{OntoOJoin}(T, I_t, R, I_r, pred), pred_t) \Leftrightarrow \mathtt{OntoOJoin}(\mathtt{OntoSelect}(T, I_t, pred_t), R, I_r, pred)$$

La règle suivante permet de combiner une sélection et une jointure en une seule jointure :

$$\mathtt{OntoSelect}(\mathtt{OntoOJoin}(T, I_t, R, I_r, pred_1), pred_2) \Leftrightarrow \mathtt{OntoOJoin}(T, I_t, R, I_r, pred_1 \wedge pred_2)$$

La règle suivante permet d'extraire une sélection d'un prédicat de jointure :

$$\mathtt{OntoOJoin}(T, I_t, R, I_r, pred_t \wedge pred) \Leftrightarrow \mathtt{OntoOJoin}(\mathtt{OntoSelect}(T, I_t, pred_t), R, I_r, pred)$$

La règle suivante permet de transformer une requête imbriquée en une jointure :

$$\mathtt{OntoSelect}(T, I_t, \lambda t_\bullet \exists r \in I_r.(pred(t, r))) \Leftrightarrow \mathtt{OntoProject}(\mathtt{OntoOJoin}(T, I_t, R, I_r, \lambda t \lambda r_\bullet pred(t, r)), \lambda tr_\bullet \{(A_1, tr.A_1), \ldots, (A_m, tr.A_m)\})$$

La règle suivante permet d'inverser le sens d'une requête imbriquée :

$$\mathtt{OntoProject}(T, I_t, \lambda t_\bullet \{(R, \mathtt{OntoSelect}(R, I_r, \lambda r_\bullet r.prop = t.prop))\}) \Leftrightarrow \mathtt{OntoSelect}(R, I_r, \lambda r_\bullet \exists t\,(t \in I_t \wedge r.prop = t.prop))$$

La règle suivante permet de remplacer une jointure par le dégroupage d'une requête imbriquée :

$$\mathtt{OntoOJoin}(T, I_t, R, I_r, pred) \Leftrightarrow \mathtt{OntoSelect}(\mathtt{OntoUnNest}(\mathtt{OntoProject}(T, I_t, \lambda t\{(A_1, t.A_1), \ldots, (A_m, t.A_m), (R, I_r)\}), R, R), pred)$$

La règle suivante est similaire à la précédente mais réalise d'abord la sélection avant le dégroupage de la requête imbriquée :

$$\mathtt{OntoOJoin}(T, I_t, R, I_r, pred) \Leftrightarrow \mathtt{OntoUnNest}(\mathtt{OntoProject}(T, I_t, \lambda t\{(A_1, t.A_1), \ldots, (A_m, t.A_m), (R, \mathtt{OntoSelect}(R, I_r, pred))\}), R, R)$$

Enfin, la règle suivante permet de déplier une requête polymorphique en utilisant l'opérateur `OntoUnion` :

$$\theta_1(\ldots\theta_n(\mathtt{OntoProject}(\mathtt{C},\mathtt{ext}^*(\mathtt{C}),\ldots))) \Leftrightarrow \mathtt{OntoUnion}(\\
\ldots\mathtt{OntoUnion}(\theta_1(\ldots\theta_n(\mathtt{OntoProject}(\mathtt{C},\mathtt{ext}(\mathtt{C}),\ldots))),\\
\theta_1(\ldots\theta_n(\mathtt{OntoProject}(\mathtt{C}_1,\mathtt{ext}^*(\mathtt{C}_1),\ldots))))\ \ldots\\
,\theta_1(\ldots\theta_n(\mathtt{OntoProject}(\mathtt{C}_n,\mathtt{ext}^*(\mathtt{C}_n),\ldots)))$$

Annexe D

Détails sur l'implantation du langage OntoQL

1 Traduction d'une expression algébrique d'une requête OntoQL en une expression de l'algèbre relationnelle étendue

Les règles présentées dans le tableau D.1 permettent de traduire une expression *OntoAlgebra*, qui correspond à une requête OntoQL sur les données, en une expression de l'algèbre relationnelle étendue.

	Expressions OntoAlgebra	**Expressions de l'algèbre relationnelle étendue**
1	$\mathtt{OntoProject}\ (C, \mathtt{ext}(C), \{(p_1, p_1), \ldots, (p_n, p_n)\}$	$\pi_{Pp_1_rids, Pp_2_rid, Pp_3, \ldots, Pp_u, NULL \rightarrow Pp_{u+1}, \ldots, NULL \rightarrow Pp_n}(EC)$
2	$\mathtt{OntoProject}\ (C, \mathtt{ext}^*(C), \{(p_1, p_1), \ldots, (p_n, p_n)\}$	$\underline{\mathtt{OntoProject}\ (C, \mathtt{ext}(C), \{(p_1, p_1), \ldots, (p_n, p_n)\})} \cup$ $\underline{\mathtt{OntoProject}\ (C_1, \mathtt{ext}^*(C_1), \{(p_1, p_1), \ldots, (p_n, p_n)\})} \cup$ $\cdots \cup$ $\underline{\mathtt{OntoProject}\ (C_n, \mathtt{ext}^*(C_n), \{(p_1, p_1), \ldots, (p_n, p_n)\})}$
3	$\mathtt{OntoOJoin}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2}, \mathtt{true})$	$\underline{\mathtt{exp}_1} \times \underline{\mathtt{exp}_2}$
4	$\mathtt{OntoOJoin}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2}, \mathtt{pred})$	$\underline{\mathtt{exp}_1} \underset{\mathtt{pred}}{\bowtie} \underline{\mathtt{exp}_2}$
5	$\mathtt{OntoLeftOuterOJoin}$ $(\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2}, \mathtt{pred})$	$\underline{\mathtt{exp}_1} \underset{\mathtt{pred}}{⟕} \underline{\mathtt{exp}_2}$
6	$\mathtt{OntoRightOuterOJoin}$ $(\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2}, \mathtt{pred})$	$\underline{\mathtt{exp}_1} \underset{\mathtt{pred}}{⟖} \underline{\mathtt{exp}_2}$
7	$\mathtt{OntoFullOuterOJoin}$ $(\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2}, \mathtt{pred})$	$\underline{\mathtt{exp}_1} \underset{\mathtt{pred}}{⟗} \underline{\mathtt{exp}_2}$
8	$\mathtt{OntoUnNest}\ (\underline{\mathtt{exp}}, p_1, p)$	$\underline{\mathtt{exp}} \underset{EC_1.p = ANY(\underline{exp}.Pp_1_rids)}{\bowtie} \underline{\mathtt{OntoProject}\ (C_1, \mathtt{ext}^*(C_1),}$ $\underline{\{(p, \mathtt{oid})\})}$
9	$\mathtt{OntoProject}\ (\underline{\mathtt{exp}}, \{(p_1, p_1), \ldots, (p_n, p_n)\})$	$\pi_{Pp_1_rids, Pp_2_rid, Pp_3, \ldots, Pp_n}(\underline{\mathtt{exp}})$
10	$\mathtt{OntoDupEliminate}\ (\underline{\mathtt{exp}})$	$\delta(\underline{\mathtt{exp}})$
11	$\mathtt{OntoSelect}\ (\underline{\mathtt{exp}}, \mathtt{pred})$	$\sigma_{\mathtt{pred}}(\underline{\mathtt{exp}})$
12	$\mathtt{OntoNest}\ (\underline{\mathtt{exp}}, \{\ p_{k+1}, \ldots, p_n\})$	$\gamma_{Pp_1_rids, Pp_2_rid, Pp_3, , \ldots, p_k}(\underline{\mathtt{exp}})$

13	$\mathtt{OntoSort}\ (\underline{\mathtt{exp}}, \{\ \mathtt{p}_1, \ldots, \mathtt{p}_n\})$	$\tau_{\mathtt{Pp_1_rids}, \mathtt{Pp_2_rid}, \mathtt{Pp_3}, \ldots, \mathtt{Pp_n}}(\underline{\mathtt{exp}})$
14	$\mathtt{OntoUnion}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2})$	$\underline{\mathtt{exp}_1} \cup \underline{\mathtt{exp}_2}$
15	$\mathtt{OntoIntersection}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2})$	$\underline{\mathtt{exp}_1} \cap \underline{\mathtt{exp}_2}$
16	$\mathtt{OntoDifference}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2})$	$\underline{\mathtt{exp}_1} - \underline{\mathtt{exp}_2}$

Tab. D.1: Règles permettant de traduire une requête OntoQL sur les données en une requête SQL

Les règles présentées dans le tableau D.2 permettent de traduire une expression *OntoAlgebra*, qui correspond à une requête OntoQL sur les ontologies, en une expression de l'algèbre relationnelle étendue.

	Expressions OntoAlgebra	**Expressions de l'algèbre relationnelle étendue**
1	$\mathtt{OntoProject}\ (\#\mathtt{E}, \mathtt{ext}(\#\mathtt{E}), \{(\#\mathtt{a}_1, \#\mathtt{a}_1), \ldots, (\#\mathtt{a}_n, \#\mathtt{a}_n)\}$	$\mathtt{ONLY(E_e)}$
2	$\mathtt{OntoProject}\ (\#\mathtt{E}, \mathtt{ext}^*(\#\mathtt{E}), \{(\#\mathtt{a}_1, \#\mathtt{a}_1), \ldots, (\#\mathtt{a}_n, \#\mathtt{a}_n)\}$	$\mathtt{E_e}$
3	$\mathtt{OntoOJoin}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2}, \mathtt{true})$	$\underline{\mathtt{exp}_1} \times \underline{\mathtt{exp}_2}$
4	$\mathtt{OntoOJoin}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2}, \mathtt{pred})$	$\underline{\mathtt{exp}_1} \underset{\mathtt{pred}}{\bowtie} \underline{\mathtt{exp}_2}$
5	$\mathtt{OntoLeftOuterOJoin}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2}, \mathtt{pred})$	$\underline{\mathtt{exp}_1} \underset{\mathtt{pred}}{\ltimes} \underline{\mathtt{exp}_2}$
6	$\mathtt{OntoRightOuterOJoin}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2}, \mathtt{pred})$	$\underline{\mathtt{exp}_1} \underset{\mathtt{pred}}{\rtimes} \underline{\mathtt{exp}_2}$
7	$\mathtt{OntoFullOuterOJoin}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2}, \mathtt{pred})$	$\underline{\mathtt{exp}_1} \underset{\mathtt{pred}}{\times} \underline{\mathtt{exp}_2}$
8	$\mathtt{OntoUnNest}\ (\underline{\mathtt{exp}}, \#\mathtt{a}_3, \mathtt{a})$	$\pi_{\mathtt{rid_s} \rightarrow \mathtt{a}}\ (\ \mathtt{E_2_a}_3 \underset{\mathtt{rid=ANY(a_3)}}{\bowtie} \underline{\mathtt{exp}}\ )$
9	$\mathtt{OntoProject}\ (\underline{\mathtt{exp}}, \{(\#\mathtt{a}_1, \#\mathtt{a}_1)\})$	$\pi_{\mathtt{a}_1}(\ \underline{\mathtt{exp}}\ )$
10	$\mathtt{OntoProject}\ (\underline{\mathtt{exp}}, \{(\#\mathtt{a}_2, \#\mathtt{a}_2)\})$	$\pi_{\mathtt{rid_s}}(\ \underline{\mathtt{exp}} \underset{\mathtt{a_2=rid}}{\bowtie} \mathtt{E_2_a}_2\ )$
11	$\mathtt{OntoProject}\ (\underline{\mathtt{exp}}, \{(\#\mathtt{a}_3, \#\mathtt{a}_3)\})$	$\pi_{\mathtt{Array}(\pi_{\mathtt{rid_s}}(\sigma_{\mathtt{rid=ANY}(\underline{\mathtt{exp}}.\mathtt{a}_3)}(\mathtt{E_2_a}_3)))}(\underline{\mathtt{exp}})$
14	$\mathtt{OntoProject}\ (\underline{\mathtt{exp}}, \{(\#\mathtt{a}_4, \#\mathtt{a}_4)\})$	$\pi_{\mathtt{a}_4}\ (\underline{\mathtt{exp}} \underset{\mathtt{a_{lien}=rid}}{\bowtie} \mathtt{E_2_a}_{\mathtt{lien}} \underset{\mathtt{rid_s=rid}}{\bowtie} \mathtt{E}_{\mathtt{def}}\mathtt{_e})$
12	$\mathtt{OntoDupEliminate}\ (\underline{\mathtt{exp}})$	$\delta(\underline{\mathtt{exp}})$
13	$\mathtt{OntoSelect}\ (\underline{\mathtt{exp}}, \mathtt{pred})$	$\sigma_{\mathtt{pred}}(\underline{\mathtt{exp}})$
14	$\mathtt{OntoNest}\ (\underline{\mathtt{exp}}, \{\ \#\mathtt{a}_1, \ldots, \#\mathtt{a}_4\})$	$\gamma_{\mathtt{a}_5, \ldots, \mathtt{a}_n}(\underline{\mathtt{exp}})$
15	$\mathtt{OntoSort}\ (\underline{\mathtt{exp}}, \{\ \#\mathtt{a}_5, \ldots, \#\mathtt{a}_n\})$	$\tau_{\mathtt{a}_5, \ldots, \mathtt{a}_n}(\underline{\mathtt{exp}})$
16	$\mathtt{OntoUnion}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2})$	$\underline{\mathtt{exp}_1} \cup \underline{\mathtt{exp}_2}$
17	$\mathtt{OntoIntersection}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2})$	$\underline{\mathtt{exp}_1} \cap \underline{\mathtt{exp}_2}$
18	$\mathtt{OntoDifference}\ (\underline{\mathtt{exp}_1}, \underline{\mathtt{exp}_2})$	$\underline{\mathtt{exp}_1} - \underline{\mathtt{exp}_2}$

Tab. D.2 – Règles permettant de traduire une requête OntoQL sur les ontologies en une requête SQL

2 Correspondances entre le modèle d'ontologies noyau de OntoQL et le modèle PLIB

Le tableau D.3 présente les correspondances entre les entités et attributs du modèle noyau de OntoQL et ceux du modèle PLIB.

Entités et attributs OntoQL	Entités et attributs PLIB
Ontology _namespace	Supplier_element _code (Supplier_bsu)
Concept _code _name _definition _definedBy	Class_and_Property_elements _code (Basic_semantic_unit) _preferred_name (Item_names) _definition _defined_by (Class_bsu)
Class _directSuperClasses	Class _its_superclass ∪ is_case_of (Item_class_case_of)
Property _scope _range	Property_det _name_scope (Property_bsu) _domain
Datatype	Datatype
PrimitiveType	Simple_type
BooleanType	Boolean_type
NumberType	Number_type
StringType	String_type
IntType	Int_type
RealType	Real_type
RefType _onClass	Class_instance_type _domain
CollectionType _ofDatatype	Entity_instance_type_for_aggregate _value_type (Aggregate_type)

Tab. D.3 – Correspondances entre les entités et attributs du modèle noyau de OntoQL et ceux de PLIB

3 Correspondances entre le modèle RDF-Schema et le modèle d'ontologies noyau de OntoQL

Le tableau D.4 présente le mapping entre le modèle RDF-Schema et le modèle d'ontologies noyau de OntoQL.

Entités et attributs RDF-Schema	**Entités et attributs du modèle noyau de OntoQL**
rdfs:Resource _rdfs:label _rdfs:comment _rdfs:seeAlso _rdfs:isDefinedBy _rdfs:member _rdfs:value	#Concept _#name _#definition _ _ _ _
rdfs:Class _rdfs:subClassOf	#Class _#superClasses
rdfs:Property _rdfs:domain _rdfs:range _rdfs:subPropertyOf	#Property _#scope _#range _
rdfs:Datatype	#Datatype
rdfs:Literal	#SimpleType
rdfs:XMLLiteral	
rdfs:Container	#CollectionType
rdf:Bag	
rdf:Seq	
rdf:Alt	
rdf:List _rdfs:first _rdfs:rest	 _ _
rdfs:Statement _rdfs:subject _rdfs:predicate _rdfs:object	 _ _ _

TAB. D.4 – Correspondances entre les entités et attributs du modèle RDF-Schema et ceux du modèle noyau de OntoQL

Table des figures

1.1 OCC locales sur le domaine des vins 19
1.2 OCNC intégrée sur le domaine des vins 20
1.3 Le modèle en oignon pour les ontologies de domaine 23
1.4 Utilisation d'ontologies pour l'échange canonique de données 24
1.5 Un extrait de l'ontologie SIOC représentée sous forme graphique 28
1.6 Un extrait de l'ontologie SIOC représentée dans le format RDF/XML 28
1.7 Extrait de la représentation générique de l'ontologie SIOC 43
1.8 Extrait de la représentation spécifique de l'ontologie SIOC 44
1.9 Représentation verticale des données décrites par l'ontologie SIOC 46
1.10 Représentation binaire des données décrites par l'ontologie SIOC 46
1.11 Représentation horizontale des données décrites par l'ontologie SIOC 47
1.12 Notre proposition d'architecture de bases de données 49

2.1 Positionnement des exigences par rapport à l'architecture de bases de données proposée . 57

3.1 Illustration du modèle de données du niveau ontologique 101
3.2 Exemple d'opération `UNNEST` 118
3.3 Exemples de création de classes non canoniques par des expressions booléennes de OWL 124
3.4 Exemples de création de classes non canoniques par des restrictions OWL 125

4.1 Le modèle d'ontologies noyau du langage OntoQL 137
4.2 Exemple d'extensions du modèle d'ontologies noyau du langage OntoQL 138
4.3 Automatisation du calcul de l'extension des classes non canoniques `HasValue` 155

5.1 Exemple de formalisation de la classe `Item` de l'ontologie SIOC 168

5.2 Exemple de génération de l'expression algébrique *OntoAlgebra* d'une requête OntoQL . 178
5.3 Exemple d'optimisation traditionnelle d'une requête OntoQL 185
5.4 Exemple d'optimisation d'une requête OntoQL par évaluation partielle 187

6.1 Architecture OntoDB 192
6.2 Exemple de représentation des données dans OntoDB 193
6.3 Exemple de représentation des ontologies dans OntoDB 195
6.4 Exemple de représentation des données conformes au modèle noyau de OntoQL dans OntoDB 197
6.5 Exemple de représentation de données utilisant des extensions du modèle noyau de OntoQL dans OntoDB 198
6.6 Les différentes étapes du traitement d'une requête OntoQL 203
6.7 Exemple de traitement d'une requête OntoQL 204
6.8 Exemple de traduction de l'opérateur `OntoProject` 207
6.9 Exemple de traduction de l'opérateur `OntoUnnest` 207
6.10 Exemple de traduction d'une requête sur les ontologies 211
6.11 Implantation des itérateurs dynamiques 212
6.12 Implantation des projections impliquant des propriétés déterminées à l'exécution 213
6.13 Exemple montrant l'implantation de l'opérateur `TYPEOF` 213
6.14 OntoQL*Plus, un éditeur d'instructions OntoQL en ligne de commandes 214
6.15 OntoQBE, un éditeur graphique de requêtes OntoQL 216
6.16 Extrait de l'interface JOBDBC 217
6.17 Extrait de l'interface OntoAPI pour l'accès aux ontologies 218
6.18 Extrait de l'interface OntoAPI pour l'accès aux données 220
6.19 Interface de recherche de classes et de propriétés par mots clés 222

B.1 Extrait de l'ontologie utilisée 266

Liste des tableaux

1.1 Synthèse des constructeurs offerts par les modèles d'ontologies 39

2.1 Analyse du langage SPARQL par rapport aux exigences définies 66

2.2 Analyse du langage RQL par rapport aux exigences définies 76

2.3 Analyse du langage CQL par rapport aux exigences définies 82

2.4 Analyse du langage SOQA-QL par rapport aux exigences définies 86

3.1 Support du modèle de données de la norme SQL92 par les SGBD usuels 95

3.2 Support du modèle de données de la norme SQL2003 par les SGBD usuels 95

3.3 Constructeurs du modèle de données de l'architecture ANSI/SPARC étendue au niveau logique 96

3.4 Support des principaux opérateurs relationnels par les SGBD usuels 97

3.5 Principaux opérateurs correspondant aux constructeurs relationnels-objets introduits dans la norme SQL99 98

3.6 Opérateurs de SQL disponibles dans le langage OntoQL au niveau logique 99

3.7 Correspondances entre le modèle de données relationnel-objet et le modèle de données du niveau ontologique 102

3.8 Constructeurs non canoniques proposés par les modèles d'ontologies 119

4.1 Attributs applicables sur les principales entités du modèle noyau 143

4.2 Les attributs dérivés prédéfinis 147

5.1 Extrait des règles permettant de construire l'expression algébrique d'une requête OntoQL 178

6.1 Correspondances entre les types du modèle noyau de OntoQL et ceux de PLIB 197

6.2 Extrait des correspondances entre les entités et attributs du modèle noyau de OntoQL et ceux de PLIB 199

6.3 Extrait des règles permettant de traduire une requête OntoQL sur les données en une requête SQL 205

6.4 Extrait des règles permettant de traduire une requête OntoQL sur les ontologies en une requête SQL 210

6.5 Traduction des fonctions de SPARQL en OntoQL 226

6.6 Extrait du mapping entre les entités et attributs du modèle RDF-Schema et ceux du modèle noyau de OntoQL 230

B.1 Comparaison des langages étudiés sur les cas d'utilisation 269

C.1 Règles permettant de construire l'expression algébrique d'une requête OntoQL 275

D.1 Règles permettant de traduire une requête OntoQL sur les données en une requête SQL . 280

D.2 Règles permettant de traduire une requête OntoQL sur les ontologies en une requête SQL 280

D.3 Correspondances entre les entités et attributs du modèle noyau de OntoQL et ceux de PLIB 281

D.4 Correspondances entre les entités et attributs du modèle RDF-Schema et ceux du modèle noyau de OntoQL 282

Glossaire

API : Application Programming Interface

BDBO : Base de Données à Base Ontologique
BDOO : Base de Données Orientée-Objet
BDRO : Base de Données Relationnelle-Objet
BSU : Basic Semantic Unit

F-Logic : Frame Logic

IEC : International Electrotechnical Commission
ISO : International Organization for Standardization

JDBC : Java DataBase Connectivity

LDD : Langage de Définition de Données
LDO : Langage de Définition des Ontologies
LDV : Langage de Définition de Vues
LID : Langage d'Interrogation de Données
LIO : Langage d'Interrogation des Ontologies
LMD : Langage de Manipulation de Données
LMO : Langage de Manipulation des Ontologies

MOF : Meta Object Facility

OC : Ontologie Conceptuelle
OCC : Ontologie Conceptuelle Canonique
OCNC : Ontologie Conceptuelle Non Canonique
OL : Ontologie Linguistique
OWL : Web Ontology Language

PLIB : Parts Library - Norme ISO 13584

QBE : Query-By-Example

RDF : Ressource Description Framework

SGBD : Système de Gestion de Bases de Données
STEP : Standard for the Exchange of Product Model Data Norme ISO 10303

UML : Unified Modeling Language
URI : Uniform Resource Identifier

XML : Extensible Markup Language

Résumé

Avec le développement d'Internet et des Intranets, l'échange et l'intégration de données contenues dans les bases de données sont devenus des besoins cruciaux. Au coeur de ces problèmes se situe la nécessité d'expliciter la sémantique des données d'une base de données. En tant que modèle permettant de représenter la sémantique des concepts d'un domaine, les ontologies sont apparues comme une solution possible à ces problèmes. Les approches proposées utilisent différents types d'ontologies (conceptuelles, linguistiques) et différents modèles d'ontologies (PLIB, F-Logic, etc.). Dans le contexte du Web Sémantique, des bases de données permettant de stocker à la fois ontologies et données sont apparues. Nous les appelons des Bases de Données à Base Ontologique (BDBO). Ces BDBO sont associées à des langages qui permettent d'interroger les données et les ontologies qu'elles contiennent. Cependant, conçus pour permettre la persistance et l'interrogation des données Web, ces BDBO et ces langages sont spécifiques aux modèles d'ontologies Web (RDF-Schema, OWL), ils se focalisent sur les ontologies conceptuelles et ils ne prennent pas en compte la structure relationnelle inhérente à des données contenues dans une base de données. C'est ce triple problème que vise à résoudre le travail présenté dans cette thèse. Nous proposons le langage OntoQL permettant d'exploiter une architecture de BDBO construite comme extension de l'architecture ANSI/SPARC pour permettre de représenter dans une base de données le modèle conceptuel des données et les ontologies qui en décrivent le sens. Le langage OntoQL répond au triple problème évoqué précédemment en présentant trois caractéristiques essentielles qui le distinguent des autres langages proposés : (1) le langage OntoQL est indépendant d'un modèle d'ontologies particulier. En effet, ce langage est basé sur un noyau commun aux différents modèles d'ontologies et des instructions de ce langage permettent de l'étendre, (2) le langage OntoQL exploite la couche linguistique qui peut être associée à une ontologie conceptuelle pour permettre d'exprimer des instructions dans différentes langues naturelles et (3) le langage OntoQL est compatible avec le langage SQL, permettant ainsi d'exploiter les données au niveau logique d'une BDBO, et il étend ce langage pour permettre d'accéder aux données au niveau ontologique indépendamment de la représentation logique des données tout en permettant d'en manipuler la structure. Nous avons validé le langage OntoQL d'une part en lui associant une sémantique formelle constituée d'une algèbre d'opérateurs, et, d'autre part, en l'implantant sur la BDBO OntoDB que nous avons équipée de différents outils facilitant l'exploitation des ontologies et des données qu'elle contient.

Mots-clés : Langage de requête, Base de données, Ontologie, Base de données à base ontologique, PLIB, RDF-Schema, OWL, F-Logic.

OntoQL, un langage d'exploitation des bases de données à base ontologique

Présenté par :

Stéphane JEAN

Directeurs de Thèse :

Yamine AIT-AMEUR et Guy PIERRA

Résumé. Avec le développement d'Internet et des Intranets, l'échange et l'intégration de données contenues dans les bases de données sont devenus des besoins cruciaux. Au cœur de ces problèmes se situe la nécessité d'expliciter la sémantique des données d'une base de données. En tant que modèle permettant de représenter la sémantique des concepts d'un domaine, les ontologies sont apparues comme une solution possible à ces problèmes. Les approches proposées utilisent différents types d'ontologies (conceptuelles, linguistiques) et différents modèles d'ontologies (PLIB, F-Logic, etc.). Dans le contexte du Web Sémantique, des bases de données permettant de stocker à la fois ontologies et données sont apparues. Nous les appelons des *Bases de Données à Base Ontologique* (BDBO). Ces BDBO sont associées à des langages qui permettent d'interroger les données et les ontologies qu'elles contiennent. Cependant, conçus pour permettre la persistance et l'interrogation des données Web, ces BDBO et ces langages sont spécifiques aux modèles d'ontologies Web (RDF-Schema, OWL), ils se focalisent sur les ontologies conceptuelles et ils ne prennent pas en compte la structure relationnelle inhérente à des données contenues dans une base de données. C'est ce triple problème que vise à résoudre le travail présenté dans cette thèse. Nous proposons le langage OntoQL permettant d'exploiter une architecture de BDBO construite comme extension de l'architecture ANSI/SPARC pour permettre de représenter dans une base de données le modèle conceptuel des données et les ontologies qui en décrivent le sens. Le langage OntoQL répond au triple problème évoqué précédemment en présentant trois caractéristiques essentielles qui le distinguent des autres langages proposés : (1) le langage OntoQL est indépendant d'un modèle d'ontologies particulier. En effet, ce langage est basé sur un noyau commun aux différents modèles d'ontologies et des instructions de ce langage permettent de l'étendre, (2) le langage OntoQL exploite la couche linguistique qui peut être associée à une ontologie conceptuelle pour permettre d'exprimer des instructions dans différentes langues naturelles et (3) le langage OntoQL est compatible avec le langage SQL, permettant ainsi d'exploiter les données au niveau logique d'une BDBO, et il étend ce langage pour permettre d'accéder aux données au niveau ontologique indépendamment de la représentation logique des données tout en permettant d'en manipuler la structure. Nous avons validé le langage OntoQL, d'une part, en lui associant une sémantique formelle constituée d'une algèbre d'opérateurs, et, d'autre part, en l'implantant sur la BDBO OntoDB que nous avons équipée de différents outils facilitant l'exploitation des ontologies et des données qu'elle contient.

Mots-clés : Langage de requête, Base de données, Ontologie, Base de données à base ontologique, PLIB, RDF-Schema, OWL, F-Logic.