OntoDB2 : un système flexible et efficient de base de données à base ontologique pour le web sémantique et les données techniques

Chimène Fankam

Ecole Nationale Supérieure de Mécanique et d'Aérotechnique

Ecole Doctorale des Sciences et l'Ingénierie pour l'Information

THESE

pour l'obtention du grade de

DOCTEUR DE L'ECOLE NATIONALE SUPERIEURE DE MECANIQUE ET D'AEROTECHNIQUE

(Diplôme National — Arrêté du 7 Août 2006)

Secteur de Recherche : INFORMATIQUE et APPLICATIONS

Présentée par :

Chimène FANKAM

****

OntoDB2 : un système flexible et efficient de Base de Données à Base Ontologique pour le Web sémantique et les données techniques

****

Directeurs de Thèse

Guy PIERRA et Ladjel BELLATRECHE

****

Soutenu le 10 Décembre 2009
Devant la Commission d'Examen

****

JURY

Président :	Danielle BOULANGER	Professeur, Université de Lyon3
Rapporteurs :	Nadine CULLOT	Professeur, Université de Bourgogne
	Hacid MOHAND-SAÏD	Professeur, Université Claude Bernard Lyon 1
Examinateurs :		
	Jean CHARLET	Chercheur, INSERM
	Guy PIERRA	Professeur, ENSMA, Futuroscope
	Ladjel BELLATRECHE	Maître de Conférences, ENSMA, Futuroscope

Remerciements

Mes remerciements les plus sincères s'adressent à :

Guy PIERRA, mon directeur de thèse, pour avoir bien voulu m'accueillir dans le laboratoire, dans son équipe de recherche et m'avoir encadré durant ces quatre années. Je le remercie pour la confiance qu'il m'a témoignée, pour son entière disponibilité, ses encouragements et pour m'avoir transmis un peu de son savoir faire et pour toutes les lumières qu'il a apporté dans cette thèse.

Ladjel BELLATRECHE co-directeur de cette thèse, pour son encadrement, son enthousiasme, sa passion de la recherche et surtout pour l'aide qu'il m'a apporté dans mes travaux.

Yamine AIT-AMEUR, Directeur du LISI et membre de l'équipe d'Ingénierie De Données du LISI pour tous ses bons conseils et bonnes remarques.

Nadine Cullot et Mohand-Saïd Hacid qui ont eu la lourde tâche de rapporter ma thèse, ainsi que les autres membres du jury Jean CHARLET et Danielle BOULANGER, lesquels m'ont fait l'honneur d'accepter d'être examinateurs. Je les remercie pour l'intérêt qu'ils portent à mes travaux.

Sybille CAFFIAU pour son amitié, ainsi que toute la famille CAFFIAU (annie, hervé, laurianne et baptiste) qui m'ont accueilli dans leur famille. Je tiens à les remercier pour leur gentillesse et pour les merveilleux moments passés en leur compagnie.

Tout le personnel du LISI, et plus particulièrement, Loé, Stéphane J., Christian, Henry-Valery, Ahmed, Nabil, Idir, Eric, Dago, Malo, Youcef, Michaël B, Kamel, Dilek, Claudine, Frédéric, Laura, Youness, François, Michaël R, Medhi et Chedlia, pour leur présence et leur soutien cordial.

Ma famille, et plus particulièrement mes parents, qui ont su m'encourager jour après jour et qui ont toujours cru en moi. Sans leur soutien constant, tant affectif que matériel, je n'aurai jamais pu accomplir mes études et envisager cette thèse.

A tous ceux qui me sont chers :
mon père, ma mère,
mes frères (Joël, Livingstone, Armstrong),
mes soeurs (Marcelle, Cédine, Priscilla),
Patrick et notre fille bien aimée Phebe.

Table des matières

Partie II Notre proposition d'architecture

Partie III Validation

Annexe

Résumé

Le besoin d'expliciter la sémantique des données dans différents domaines scientifiques (biologie, médecine, géographie, ingénierie, etc.) s'est traduit par la définition de données faisant référence à des ontologies, encore appelées données à base ontologique. Avec la multiplication des ontologies de domaine, et le volume important de données à manipuler, est apparu le besoin de systèmes susceptibles de gérer des données à base ontologique de grande taille. De tels systèmes sont appelés des systèmes de gestion de Bases de Données à Base Ontologique (BDBO).

Les principales limitations des systèmes de gestion de BDBO existants sont (1) leur rigidité, due à la prise en compte des constructions d'un unique formalisme d'expression d'ontologies, (2) l'absence de support pour les données non standard (spatiales, temporelles, etc.) et, (3) leur manque d'efficacité pour gérer les données de grande taille. Nous proposons dans cette thèse un nouveau système de gestion de BDBO permettant (1) de supporter des ontologies basées sur différents formalismes d'ontologies, (2) l'extension de son formalisme d'ontologie pour répondre aux besoins spécifiques des applications, et (3) une gestion originale des données facilitant le passage à grande échelle.

Le système que nous proposons dans cette thèse, OntoDB2, se fonde sur l'existence d'un ensemble de constructions communes aux différents formalismes d'expression d'ontologies, susceptible de constituer une ontologie noyau, et sur les techniques de gestion des modèles pour permettre l'extension flexible de ce noyau. Nous proposons également une approche originale de gestion des données à base ontologique. Cette approche part du fait que les données à base ontologique peuvent se classifier en données canoniques (instances de classes primitives) et non-canoniques (instances de classes définies). Les instances de classes définies peuvent, sous certaines hypothèses, s'exprimer en termes d'instances de classes primitives. Nous proposons donc de ne représenter que les données canoniques, en transformant sous certaines conditions, toute donnée non-canonique en donnée canonique. Enfin, nous proposons d'exploiter l'interpréteur de requêtes ontologiques pour permettre (1) l'accès aux données non-canoniques ainsi transformées et, (2) d'indexer et pré-calculer les raisonnements en se basant sur les mécanismes du SGBD support. L'ensemble de ces propositions est validé (1) à travers une implémentation sur le SGBD PostgreSQL basée sur les formalismes d'ontologies PLIB, RDFS et OWL Lite, (2) des tests de performances sur des ensembles de données issus de la géographie et du Web.

Mots-clés: Base de données, ontologie, formalisme d'ontologies, bases de données à base ontologique, méta-modélisation, ingénierie dirigée par les modèles, ingénierie des données, interrogation de données, raisonnement, PLIB, OWL

.

Introduction générale

Contexte

Les ontologies sont des structures qui permettent de représenter explicitement la sémantique d'un domaine par des modèles objets consensuels dont chaque concept (classe ou propriété) est associé à un identificateur universel permettant de référencer la sémantique qui lui correspond.

Les ontologies sont aujourd'hui utilisées dans un nombre croissant d'applications, par exemple pour faciliter la recherche d'information dans le domaine du Web en annotant les documents par des instances ontologiques, ou, dans le domaine technique, pour représenter des catalogues de composants industriels. Par instance ontologique, nous entendons un objet dont le sens est défini par son appartenance à une classe ontologique et par les valeurs d'un certains nombre de propriétés définies dans la même ontologie. Nous appellerons données à base ontologiques (DBO) un ensemble d'instances ontologiques.

Avec l'utilisation croissante des ontologies, un certain nombre de formalismes d'ontologie ont été proposés : RDF [39], RDFS [10], OWL [5], PLIB [57], FLIGHT [18], etc. Chacun de ces formalismes cible un domaine d'application particulier et introduit, pour ce faire, des primitives de modélisation particulières. Par exemple, le formalisme PLIB est largement utilisé dans le domaine de l'ingénierie [41], les formalismes RDFS et OWL sont utilisés dans le domaine du Web.

Initialement, les données à base ontologiques étaient gérées par des outils en mémoire centrale. Avec la multiplication des ontologies de domaine, et le volume important de données à manipuler, est apparu le besoin de systèmes susceptibles de gérer des ensembles de données à base ontologique de grande taille. De tels systèmes sont appelés des systèmes de gestion de bases de données à base ontologique (BDBO). Différentes BDBO ont ainsi été proposées telles que OntoDB [20], Ontobroker [23], RDFSuite [4], SESAME [11], OntoMS [53] et SOR [44]. La principale caractéristique de ces BDBO est qu'elles sont chacune construites pour supporter un seul formalisme d'ontologie (OWL, PLIB ou FLIGHT).

Problèmes

Les organisations évoluent dans un environnement hétérogène et recouvrent différents domaines. De plus, les organisations définissent, pour faciliter l'échange d'information, des ontologies modulaires définies à partir d'ontologies préexistantes. Les organisations étant amenées à évoluer, les ontologies qu'elles manipulent sont également amenées à évoluer afin de s'adapter aux nouveaux besoins. Par exemple, les ontologies doivent changer pour répondre aux changements de la conceptualisation ou au changement de métier de l'entreprise. Les ontologies et les forma-

lismes de description d'ontologies ont donc besoin de pouvoir évoluer en intégrant les primitives de modélisation spécifiques selon les besoins.

Aujourd'hui, les ontologies sont utilisées dans bien des domaines tels que la biologie, la médecine ou encore la géographie, où de plus en plus de données sont également définies par référence à une ontologie. Cette forte croissante des DBO nécessite de disposer de BDBO capables de passer à l'échelle c'est à dire de traiter de grand volumes de données comme savent le faire les bases de données traditionnelles.

Un problème important, pour passer ainsi à l'échelle, est que les formalismes d'ontologie modélisent l'information de façon très différente de la modélisation effectuée dans une base de données traditionnelle. En effet, dans les bases de données chaque information doit être représentée de manière unique. Cette représentation est dite *canonique*. Pour répondre à une requête sur une information, les bases de données ont donc à rechercher en un endroit unique. Au contraire, la plupart des formalismes d'ontologie permettent pour le même objet d'une part une représentation canonique, mais d'autre part un nombre quelconque d'autres représentations, dites *non canoniques*. Par exemple, il est possible de définir une personne de sexe féminin de manière canonique comme appartenant à la classe *Personne* et ayant la valeur *féminin* pour la propriété *genre*. Mais on peut également la définir sous une deuxième forme, non canonique, comme appartenant à la classe *Femme*.

L'avantage de cette approche de modélisation offerte par les formalismes d'ontologie est de pouvoir offrir plusieurs alternatives de désignation du même objet et, d'assurer l'équivalence entre ces différentes désignations. Cette approche est utile par exemple dans le domaine de la recherche documentaire, où de nombreux termes synonymes sont utilisés dans les documents pour référencer le même concept. L'inconvénient de cette approche, si elle est traitée en l'état dans les bases de données, est que pour répondre à une requête sur une information, les bases de données auront à rechercher à plusieurs endroits. Pour éviter ce problème auquel les logiciels de gestion de bases de données ne sont pas préparés, deux méthodes sont utilisées pour gérer les informations modélisées de façon non canonique. La première méthode, dite par *saturation*, consiste à calculer toutes les descriptions possibles des différents objets et à les stocker dans la base de données. Le système peut alors rechercher à n'importe quel endroit où une information peut être rangée et il la trouvera toute entière. L'inconvénient de cette méthode est qu'elle génère un grand volume de données et un temps de re-calcul des relations souvent important lors des mises à jour. La seconde méthode, par *raisonnement*, consiste à réaliser des inférences lors de l'interrogation des données. L'inconvénient de cette méthode outre le fait qu'elle n'existe pas dans les SGBD classiques, est qu'elle est relativement lente et ne passe pas à l'échelle pour les grands volumes de données.

Intérêt initial du laboratoire : le domaine technique et PLIB

Les travaux présentés dans ce mémoire ont été réalisés dans le cadre des recherches de l'équipe Ingénierie de Données (IDD) du laboratoire LISI. Les thèmes principaux de cette équipe portent sur la modélisation à base ontologique, sur l'intégration, la gestion persistante et l'échange de données avec comme domaine d'application privilégié le secteur de l'ingénierie et des données techniques. Ces recherches s'appuient sur des problématiques concrètes à traiter dans le cadre

de différents projets de recherche tels que les projets ANR e-Wok Hub et DAFOE. Elles s'appuient également souvent sur un formalisme d'ontologie, développé principalement pour le domaine technique et publié sous forme d'une série de normes ISO, le modèle PLIB (ISO 13584).

Au début des années 2000, un travail était lancé, au LISI, pour réaliser un système de BDBO supportant complètement le formalisme d'ontologie PLIB. Une architecture de BDBO appelée OntoDB, permettant de gérer de grandes tailles de données à base ontologique PLIB, a été finalisée en 2004. Compte tenu de la complexité du modèle objet PLIB, composé de 217 entités et de 118 types, sa traduction en base de données a également abouti à un système complexe comportant 568 tables de représentation pour le niveau ontologique. La complexité de PLIB rend sa compréhension relativement difficile et le système obtenu est également difficilement extensible sauf pour l'équipe l'ayant développé ou pour une personne ayant une très bonne connaissance de PLIB. De plus, le formalisme d'ontologie PLIB étant orienté vers l'échange, il ne définit pas de concept non canonique. le système OntoDB ne gère de ce fait que des données canoniques.

OntoDB est aujourd'hui largement utilisé en dehors du domaine de l'ingénierie. Il est utilisé sur des problématiques concrètes à résoudre dans le cas de différents projets de recherches menés au LISI, notamment dans les projets e-Wok Hub et DAFOE. Or, chacune de ces applications a mis en évidence le besoin de pouvoir compléter OntoDB en intégrant des mécanismes adaptés aux besoins spécifiques propres à chaque application. Dans le domaine de l'indexation de document, le besoin est apparu de pouvoir utiliser des types de données spécifiques, et de pouvoir collecter des termes différents décrivant le même concept. Dans le domaine médical, le besoin est apparu de permettre à des médecins de différentes spécialités de partager des informations sur le même patient. Or, les langages courants employés par les spécialistes et les généralistes sont parfois très différents. Néanmoins, ces derniers devant accéder à toutes les informations disponibles sur un patient, il est nécessaire de supporter à la fois plusieurs langages de description, et donc une description canonique, mais aussi des représentations non canoniques dans OntoDB.

Objectifs

Ce sont les difficultés mentionnées ci-dessus qui ont amené à définir les objectifs du présent travail. Il s'agit d'élaborer des propositions pour les trois problématiques de recherche suivantes :

1. **Support d'un formalisme d'ontologie flexibilité et efficace**. Il s'agit de définir un système de BDBO doté d'un formalisme d'ontologie capable de s'adapter aux évolutions du formalisme d'ontologie. Pour cela, le formalisme d'ontologie supporté doit non seulement couvrir le formalisme d'ontologie PLIB, mais il doit aussi être capable de supporter toutes ou certaines des primitives de modélisation d'autres formalismes d'ontologie (RDFS, OWL, FLIGHT, etc.). Cette flexibilité devra en particulier, permettre d'intégrer les mécanismes identifiés comme importants pour les applications des projets menés au LISI. On s'intéresse également à l'efficacité de ce formalisme d'ontologie. Ce dernier doit en particulier supporter de facon efficace l'accès aux ontologies PLIB et il doit avoir un accès simple pour les non spécialistes de PLIB, et ceci de façon simplifiée ; c'est à dire (1) modifiable par un utilisateur non spécialiste de PLIB et (2) efficace en temps de traitement.
2. **Introduction de types de données spécifiques**. Les types de données de la BDBO doivent pouvoir être étendus afin de permettre la représentation de domaines de valeurs

qui n'étaient pas prévus initialement par le système.

3. **Gestion simultanée des données canoniques et non canoniques sans saturation ni raisonnement**. Une approche doit être proposée pour que la BDBO puisse lire et gérer à la fois des données canoniques et non canoniques sans utiliser une des méthodes classique.

Notre proposition

Les objectifs mentionnés ci-dessus ne pouvant être développés dans OntoDB, nous nous sommes intéressé, dans cette thèse, à la définition d'une architecture de BDBO nouvelle que nous appelons OntoDB2. Nous proposons dans OntoDB2 :

1. d'utiliser les technique d'IDM pour définir une représentation fortement simplifiée du formalisme d'ontologie supporté, de pouvoir d'expression au moins égal à PLIB, mais aisément modifiable et beaucoup plus rapide ;
2. d'utiliser les techniques d'ingénierie dirigée par les modèles (IDM) associées à une architecture de BDBO de type MOF[1] pour supporter et représenter aisément les modifications du système de type de données et en particulier, offrir le support des types de données géographiques ;
3. d'élaborer une nouvelle approche de gestion des instances consistant à convertir les données non canoniques en données canoniques (migration d'instances) puis, à représenter et à interroger (1) les classes canoniques complétées en utilisant les mécanismes usuels des bases de données et (2) les classes non canoniques par des vues construites sur les classes canoniques.

Organisation du mémoire

Ce manuscrit s'organise en trois parties comportant chacune deux chapitres. La partie 1 présente l'état de l'art et la problématique telle qu'elle est traitée dans une étude bibliographique. La partie 2 décrit les solutions que nous avons proposées et développées afin de résoudre les problèmes identifiés. Enfin, la Partie 3 présente la validation du système de BDBO OntoDB2 que nous avons conçu.

Partie 1 : État de l'art

Cette partie présente un état de l'art sur les ontologies et les systèmes de gestion de BDBO.

Le chapitre 1 présente une analyse du concept d'ontologie. Après une définition, nous y présentons deux familles de formalismes d'expressions d'ontologies : (1) les formalismes d'ontologie orientés gestion en échange de données PLIB et RDFS et (2) les formalismes orientées inférences, où nous discutons en particulier du formalisme OWL. La principale contribution de ce chapitre est de proposer une comparaison de ces différents formalismes en termes des similitudes et des différences qu'ils comportent suivant les trois couches du modèle en oignon de classification des ontologies.

[1] Meta Object Flexibility

Le chapitre 2 présente une étude sur les systèmes de bases de données à base ontologique. Nous commençons par donner une définition des termes DBO et BDBO. Ensuite nous proposons une classification des architectures de BDBO suivant la structuration qu'ils utilisent pour la représentation des ontologies. Nous présentons ensuite les approches adoptées dans ces architectures pour la représentation des DBO et pour la gestion de la montée en charge et l'interrogation de ces DBO. Nous discutons ensuite dans ce chapitre des besoins actuels des applications et, montrons au travers de l'étude d'un certain nombre de systèmes représentatifs de l'offre actuel que ces besoins restent encore à être satisfaits. L'objectif visé dans ce chapitre est de mettre en évidence dans les systèmes existants, les besoins qui n'étaient pas bien couverts.

A partir des limitations exposées précédemment, la partie 2 présente les exigences que doit satisfaire la BDBO OntoDB2 que nous proposons.

Partie 2 le système de BDBO OntoDB2

La deuxième partie débute par la description du système OntoDB2 au chapitre 3. Nous discutons tout d'abord dans ce chapitre, du choix du type d'architecture adopté pour OntoDB2 et des constructions choisies pour constituer le formalisme noyau du système OntoDB2. Nous y présentons également les hypothèses fondamentales qui ont justifié le choix de ces constructions. Nous décrivons ensuite les extensions de ce formalisme d'ontologie en présentant dans chaque cas des exemples de constructions d'utilité générale qui pourront être intégrées au formalisme noyau. Passant à la gestion des DBO, nous proposons ensuite une solution de gestion et d'interrogation efficace des DBO par représentation de DBO non canoniques en DBO canoniques par migration d'instances dans OntoDB2. Nous illustrons à travers des exemples, comment est réalisée cette transformation, puis comment sont interrogées les DBO par exploitation à la fois du langage de requête ontologique et des mécanismes usuels des bases de données.

Le chapitre 4 est consacré à l'implémentation du prototype de BDBO que nous avons développé sur le SGBD relationnel objet PostgreSQL. Nous y présentons l'implémentation des différentes composantes de la BDBO OntoDB2. Nous commençons par la présentation de l'ensemble de l'environnement dans lequel notre travail de thèse a été développé. Tout d'abord, nous définissons la notion d'ingénierie dirigée par les modèles (IDM) que nous avons largement utilisé et qui consiste à générer les codes (programmes) à partir des modèles. Nous présentons alors le langage EXPRESS et l'ensemble de la technologie qui lui associée. Nous présentons en particulier l'environnement d'IDM ECCO (EXPRESS Compiler Compiler) que nous avons utilisé pour implémenter de manière générique nos différents modules.

Nous présentons la structure *Flatlib* et la structure en *Peigne* du formalisme d'ontologie définies pour faciliter respectivement la représentation des ontologies dans les bases de données et leur manipulations par les applications. Des règles de transformations sont définies entre ces deux structures en exploitant la bibliothèque de mapping Hibernate. Ces correspondances sont utilisées pour la génération respectivement de la structure des tables et des API Java d'accès de la partie ontologie et de la partie méta-schéma de notre architecture.

Partie 3 : Validation du système de BDBO OntoDB2

Le chapitre 5 présente notre approche d'extension du système de types de données et du formalisme d'ontologie de la BDBO OntoDB2. Nous y présentons comment les types géométriques sont intégrés dans OntoDB2 et comment le formalisme d'ontologie est étendu pour permettre le support des relations d'ordre et des propriétés propagées par une relation d'ordre.

Dans le chapitre 6, nous présentons les expérimentations réalisées sur un ensemble de données de tests. Nous présentons également une évaluation des résultats et mettons en évidence la faisabilité et l'efficacité des solutions proposées pour les constructions supportées et implémentées. Le premier essai permet de relever les apports de la structure de représentation choisie pour le niveau ontologique. Nous présentons ensuite les performances de l'approche de substitution de certains raisonnements déductifs par des requêtes numériques et alphanumériques. Enfin, nous présentons les premiers résultats obtenus dans l'intégration d'instances non canoniques par migration d'instances.

Pour conclure les travaux présentés ci-dessus, dans le chapitre Conclusion, nous faisons un résumé de la problématique, de nos propositions et des principales approches suivies. Nous discutons également des limites et des points restant à approfondir dans le futur.

Ce manuscrit comporte également en annexe, le modèle EXPRESS du méta-schéma EXPRESS que nous avons utilisé. La liste suivante représente les publications concernant le travail dans cette thèse.

- *Chimène Fankam, Yamine Ait-Ameur and Guy Pierra*, **Exploitation of Ontology Languages for both Persistence and reasoning Purposes : Mapping PLIB, OWL and Flight ontology models**. Third International Conference on Web Information Systems and Technologies(WEBIST), Edited by : Joaquim Filipe, José Cordeiro, Bruno Encarnação and Vitor Pedrosa. , INSTICC Press, March, 2007, pp. 254-262.

- *Chimene Fankam*, **Prise en compte des ontologies non canoniques dans les BDBO : le modèle ONTODB2**, Ph D. présentation, XXVème Congrès INFORSID (INFORSID'07), Perros-Guirec France, Mai, 2007, pp 561-562.

- *Chimène Fankam, Stéphane Jean, Guy Pierra and Ladjel Bellatreche*, **Enrichissement de l'architecture ANSI/SPARC pour expliciter la sémantique des données : une approche fondée sur les ontologies**, Actes de la 2ème Conférence francophone sur les Architectures Logicielles (CAL'08), edited by Revue RNTI, mars, 2008, pp. 47-61.

- *Chimene Fankam, Ladjel Bellatreche and Guy Pierra*, **OntoDB2 : Support of Multiple Ontology Models within Ontology Based Database**, 11th International Conference on Extending Database Technology (EDBT'08) Ph.D. Workshop, Mars, 2008,. Nantes France, pp 21-27.

- *Nabil Belaid , Idir Aït Sadoune, Chimene Fankam, Stephane Jean, Yamine Aït Ameur, Guy Pierra, and Jean-Francois Rainaud*, **Une architecture orientée services pour la gestion sémantiques des données géologiques pour le stockage de CO2**, 26eme Congrès INFORSID Systèmes d'Information et de Décision pour l'Environnement, Fontai-

nebleau 27 Mai 2008.

- *Chimène Fankam, Stéphane Jean and Guy Pierra*, **Numeric reasoning in the Semantic Web**, ESWC - SEMMA : First International Workshop on Semantic Metadata Management and Applications, SeMMA 2008, Located at the Fifth European Semantic Web Conference, vol. 346,, CEUR Workshop Proceedings, edited by Khalid Belhajjame and Mathieu d'Aquin and Peter Haase and Paolo Missier, Tenerife, Spain, edited by CEUR-WS.org, June, 2008, pp. 84 - 103.

- *Yamine Aït Ameur, Nabil Belaid, Mohammed Bennis, Olivier Corby, Rose Dieng-Kuntz, Jérémie Doucy, Priscille Durville, Chimène Fankam, Fabien L. Gandon, Alain Giboin, Patrick Giroux, Sandrine Grataloup, Bruno Grilheres, Florian Husson, Stéphane Jean, Joel Langlois, Phuc-Hiep Luong, Laura Silveira Mastella, Olivier Morel, Michel Perrin, Guy Pierra, Jean-François Rainaud, Idir Aït-Sadoune, Eric Sardet, Francois Tertre and João Francisco Valiati*, **Semantic Hubs for Geological Projects**, Workshop on Semantic Metadata Management and Applications (SeMMA 2008), June, 2008, pp. 3-17.

- *Chimène Fankam, Stéphane Jean, Ladjel Bellatreche and Yamine Aït Ameur*, **Extending the ANSI/SPARC Architecture Database with Explicit Data Semantics : An Ontology-Based Approach**, Second European Conference on Software Architecture(ECSA), edited by LNCS Springer, September, 2008, pp. 318-321.

- *Chimène Fankam, Stéphane Jean and Guy Pierra*, **Raisonnement Numérique sur les Relations d'Ordre pour le Web Sémantique**, Actes de la deuxième édition des Journées Francophones sur les Ontologies (JFO 2008), edited by ACM, Décembre, 2008, pp. 4-15.

- *Chimène Fankam, Ladjel Bellatreche, Hondjack Dehainsala, Yamine Ait Ameur and Guy Pierra*, **SISRO : conception de bases de données à partir d'ontologies de domaine**, Technique et science informatiques (TSI), vol. 28, 2009.

- *Chimène Fankam, Stéphane Jean, Guy Pierra, Ladjel Bellatreche and Yamine Ait Ameur*, **Towards Connecting Database Applications to Ontologies**, First International Conference on Advances in Databases, Knowledge, and Data Applications, edited by IEEE Computer Society, Conference Publishing Service, March 2009, pp. 131-137.

- *Selma Khouri, Ladjel Bellatreche and Chimène Fankam*, **SISROM2C : Un outil de modélisation conceptuelle à base ontologique d'un entrepôt de données**, 5èmes Journées francophones sur les Entrepôts de Données et l'Analyse en ligne (EDA'09), edited by RNTI, Juin, 2009, Toulouse, France, pp 123-138.

Première partie

État de l'art

Chapitre 1

Ontologie et formalismes d'ontologies

Sommaire

Introduction

Parallèlement à l'explosion de la quantité d'information numérique dans de nombreux domaines au cours des dernières années, de nombreux travaux ont été menés pour développer des méthodes permettant de représenter explicitement la signification de ces données sous des formes échangeables et exploitables par des ordinateurs.

Les ontologies, définies par Gruber [26] comme *une spécification explicite d'une conceptualisation*, se sont imposées comme un moyen pour expliciter la sémantique des données. Elles permettent aux programmes d'échanger cette sémantique et, le cas échéant, de réaliser des raisonnements et des traitements intelligents sur les données dans des domaines aussi variés que

l'intégration des sources de données hétérogènes, ou la recherche d'information sur le Web. Un problème important est qu'un domaine donné peut être couvert par plusieurs ontologies qui peuvent être décrites en utilisant des langages différents et basées sur différentes logiques. De ce fait, l'intégration des données requiert souvent non seulement l'intégration des schémas ou modèles, mais aussi l'intégration des langages d'ontologies sous-jacents et en conséquence des logiques d'ontologies sous-jacentes.

Ce problème s'étant souvent posé dans les applications que traite le laboratoire, pouvoir faciliter la coopération voire l'intégration des différents formalismes d'ontologie au sein du système OntoDB2 fait partie des objectifs de notre travail. Ceci suppose d'analyser les différents formalismes d'ontologie, tout au moins ceux qui nous intéresse directement, afin de voir leurs points communs et leurs différences. Ceci nous permettra alors d'envisager des approches leur permettant de coopérer.

Le plan de ce chapitre est le suivant. Dans la section 1, nous définissons la notion d'ontologie et les problèmes liés à leur représentation dans les bases de données. Dans la section 2, nous présentons deux catégories de formalismes d'expression des ontologies et les problèmes qu'ils visent à résoudre au travers des constructeurs qu'ils offrent. Ces deux catégories de formalismes sont PLIB, d'une part, adapté au domaine technique, et OWL d'autre part développé pour faciliter l'accès aux données du Web. Ces formalismes sont les plus utilisés. Nous présentons ensuite, dans la section 3 une comparaison de ces formalismes. Cette comparaison nous permet de montrer la complémentarité des différents formalismes et nous amène à proposer une classification des ontologies que nous utilisons dans la suite de ce travail, pour faire coopérer différents formalismes d'ontologie.

1 La notion d'ontologie

La notion d'ontologie est apparue en informatique dans les années 90. Une ontologie est essentiellement une représentation explicite de la conceptualisation d'un domaine, telle qu'elle est perçue par une communauté donnée [26]. Les ontologies sont aujourd'hui utilisées dans divers domaines et on retrouve dans la littérature [2] de nombreuses définitions du terme ontologie en fonction du secteur d'activité visé [27] (indexation de document, traitement automatique de la langue naturelle (TALN), intégration de données). En ce qui nous concerne, nous adoptons la définition suivante :

1.1 Définition

Une ontologie est *une représentation formelle, explicite, référençable et consensuelle de l'ensemble des concepts partagés d'un domaine en terme de classes d'appartenance et de propriétés caractéristiques* [37].

Cette définition met en avant trois caractéristiques qui distinguent une ontologie de domaine des autres modèles informatiques tels que les modèles conceptuels et les modèles de connaissance. Une ontologie est une représentation :

[2] http ://websemantique.org/Ontologie

- *formelle*, exprimée dans un langage de syntaxe et de sémantique formalisé (RDF Schéma [10], DAML+OIL [15], OWL[5], PLIB[35], etc.) permettant ainsi des raisonnements automatiques ayant pour objet soit d'effectuer des vérifications de consistance, soit d'inférer de nouveaux faits ;
- *consensuelle*, c'est-à-dire admise par l'ensemble des membres (et des systèmes) d'une communauté et,
- *référençable*, c'est-à-dire que toute entité ou relation décrite dans l'ontologie peut être directement référencée par un symbole (« identifiant »), à partir de n'importe quel contexte, afin d'expliciter la sémantique de l'élément référençant.

1.2 Ontologies en Informatique

D'un point de vue pratique, une ontologie informatique est formalisée en utilisant cinq sortes de composants :

1. Les **classes** (catégories d'objets modélisés qui ont une existence propre dans le domaine modélisé) : une classe est une collection d'objets définie en intention. Exemple : Territoire, Document, Personne, etc.
2. Les **propriétés** (attributs des objets modélisés qui permettent de caractériser une instance d'une classe) : une propriété est une relation binaire entre deux classes ou entre une classe et un domaine de valeurs.
3. Les **domaines de valeurs** (ou **types de données**) : un domaine de valeurs est un ensemble mathématique défini en extension ou en intention. Exemple : réels, caractères, booléen, mais aussi les types structurés : liste, tableau, ...
4. Les **individus (ou instances)** : un individu du domaine modélisé est représenté de façon ontologique comme une instance appartenant à une classe définie dans l'ontologie. Par exemple, L'individu *Anne Dupont* peut être défini ontologiquement comme une instance de la classe des personnes, avec la valeur *1990* pour la propriété *année de naissance* et, la valeur *féminin* pour la propriété *genre*, si la classe des personnes et les propriétés année de naissance et genre sont définies dans l'ontologie.
5. Les **axiomes** qui sont des prédicats s'appliquant sur les relations, les classes ou les instances et permettent d'assurer l'intégrité des données ou de faire des inférences. Exemple : l'âge d'une personne doit être une valeur positive, une instance ne peut être caractérisée que par les propriétés applicables à sa classe

1.3 Concepts primitifs et concepts définis

La conception d'une ontologie se fait en exploitant un formalisme d'expression d'ontologies. Ces formalismes utilisent des représentations orientées objet. Ils permettent de définir des constructeurs en vue de créer des ontologies au travers des concepts énumérés ci-dessus. Depuis leur émergence en informatique, plusieurs courants ont été suivis pour définir les ontologies. On peut distinguer deux principaux courant :

1. Les formalismes d'ontologie orientés gestion et échange de données qui visent à représenter la sémantique de données d'un domaine d'application de manière précise et unique de façon

à permettre le partage et l'échange d'information.

2. Les formalismes d'ontologies orientés inférence qui visent à permettre certains raisonnements sur un domaine d'application pour résoudre certains problèmes, en exploitant des connaissances relatives au domaine étudié.

Les formalismes d'ontologie orientés gestion et échange de données définissent des constructeurs qui vont permettre de décrire les concepts des ontologies de manière à faciliter l'échange d'information. De ce fait, ces ontologies permettent de représenter chaque information de manière unique ou *canonique*. Nous caractérisons les ontologies définies par ces formalismes d'ontologie *canoniques*.

Au contraire, les formalismes d'ontologie orientés inférence vont définir plusieurs constructeurs qui vont permettent de définir le même concept (classes, propriétés et instances) de l'ontologie. Nous appelons ces constructeurs *les constructeurs d'équivalence conceptuelle* car ils offrent le moyen de représenter de différentes manières le même concept ou la même information. Par exemple, l'individu *Anne Dupont* peut être décrit comme une instance de la classe *Personne* avec les caractéristiques *nom* et *genre*, ou encore comme une instance de la classe *Femme* si cette dernière est également définie au niveau de l'ontologie comme équivalente à l'ensemble des personnes pour lesquelles la valeur de la propriété *genre* est *féminin*. ainsi, *Anne Dupont* peut être décrite de deux manières équivalentes qui permettent de faire automatiquement des inférences entre ces représentations. Les ontologies possédant de tels constructions sont dites *non canoniques*.

Un problème fondamental qui résulte de ces différentes façons de modéliser l'information par les formalismes d'ontologie est leur représentation au sein des bases de données. En effet, les bases de données sont utilisées par un très grand nombre d'applications car elles fournissent un environnement de persistance robuste, sécurisé et efficace des données. Il apparaît cependant qu'à la différence des bases de données qui représente l'information de manière unique, et donc canonique, les données définies à partir d'ontologies non canoniques peuvent être redondantes et décrites de plusieurs manières. Si l'on considère par exemple une ontologie qui comporte les classes Personne et Femme ; la classe Femme étant définie comme égale à la restriction de la classe Personne aux individus dont la valeur pour la propriété genre est égale à féminin (*Personne [genre = 'féminin']*), la représentation dans une base de données d'un ensemble d'instances de personnes comportant les femmes et les hommes nécessite :

- soit que toutes les instances de Femme soient toujours représentées comme des instances de la classe Personne et dans ce cas, elles seraient immédiatement représentable en base de données,
- soit que la base de données soit capable, lorsque l'on demande de lister toutes les femmes, de faire l'union de celles représentées comme *Personne[genre = 'féminin']* et de celles représentées comme *Femme*,
- soit que des mécanismes soient fournis pour transformer les instances de la classe Femme en instances de la classe Personne.

La première solution bien qu'immédiate est assez restrictive car elle restreint le langage autorisé pour définir les instances. Ceci est tout à fait acceptable si l'objectif de l'ontologie est de pouvoir échanger de l'information. C'est par contre, moins acceptable par exemple si l'on veut

identifier des instances dans des documents écrits. Dans ce cas, en effet, un grand nombre de formules équivalentes sont en général utilisées pour les mêmes concepts. La deuxième forme viole le principe de canonicité des bases de données. Elle exigera soit des mécanismes de raisonnement extérieurs (« raisonneurs »), soit de dupliquer l'information, ce qui est contradictoire avec la théorie des bases de données et posera les problèmes classiques de mise à jour et de cohérence dans la base de données résultante. C'est la solution souvent utilisée pour gérer les instances des classes des ontologies que ce soit en mémoire centrale [50] ou en mémoire secondaire [36]. La troisième solution ne restreint pas le langage utilisable pour décrire l'information à représenter, mais elle requiert la mise en œuvre de mécanismes spécifiques pour permettre la transformation des données décrites à partir d'ontologie avant de les charger dans la base de données. C'est l'approche que nous proposerons dans cette thèse pour lier la puissance d'expression du langage de représentation de l'information et la cohérence de la base de données résultante

Mettre en œuvre de tels mécanismes nécessite d'identifier les constructeurs offerts par les différents formalismes d'ontologies, et de distinguer les constructeurs d'équivalence conceptuelle des autres constructeurs. Cette distinction permet de définir deux catégories de concepts dans les ontologies :

1. *les concepts primitifs* ou *canoniques* qui sont des concepts " pour lesquels nous ne sommes pas capables de donner une définition axiomatique complète " [26]. La définition de ces concepts s'appuie sur une documentation textuelle et un savoir partagé avec les utilisateurs mais ne peut se réduire avec d'autres concepts. 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 par équivalence, si besoin est. 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, on ne peut certifier ni la complétude de l'ensemble des concepts consensuels décrits, ni la consistance (il n'y a pas de contradiction entre les définitions informelles et les relations formelles que l'on définit entre concepts) des définitions fournies par une ontologie ;
2. *les concepts définis* ou *non canoniques* qui 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 »[26]. 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. Elles n'introduisent donc pas de nouvelles instances 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. Dans ce type de formalisme d'ontologie ou les langages de modélisation permettent non seulement de décrire des concepts primitifs, mais comportent également différents opérateurs permettant de composer les concepts pour construire des concepts définis, cette caractéristique est la base des mécanismes d'inférence. Par exemple, un système d'inférence permettra de décider qu'une *Femme* est une *Personne* et qu'une *Personne* de *genre* = *'féminin'* est une *Femme*. Il permettra également d'effectuer des classifications automatiques, c'est-à-dire de déduire automatiquement des relations de subsomption entre concepts des relations

d'équivalence conceptuelle. Il permettra également de calculer l'appartenance d'instances à certaines classes, à partir de la définition axiomatique complète des concepts définis.

Nous présentons dans les sections suivantes quelques formalismes qui permettent de construire les ontologies.

2 Exemples de formalismes d'ontologies

Les ontologies sont exprimées à partir de formalismes d'ontologies. Ces formalismes offrent des constructeurs permettant de définir les différents concepts (classes, propriétés, types de données, individus, ...) que l'on retrouve dans une ontologie. Les ontologies sont utilisées dans différentes disciplines (bases de données, TALN, recherche d'information, ...) et ce, avec des objectifs différents. Il existe donc différents formalismes de définition d'ontologies suivant les objectifs spécifiques visés par les différentes spécialités. Nous introduisons dans cette section, les principaux formalismes d'ontologies. L'objectif est de ressortir (1) ce qu'ils ont en commun et (2) ce qui les distinguent. Nous allons ainsi pouvoir déterminer, dans quelle mesure il est possible, au moins partiellement, d'intégrer leur capacité.

2.1 Formalismes d'ontologies orientés gestion et échange de données

Dans cette section, nous présentons les formalismes d'ontologie RDF-Schéma et PLIB. Ces formalismes offrent respectivement dans le domaine du Web et de l'ingénierie, des constructeurs permettant de représenter les informations de l'univers du discours de manière à faciliter le partage et l'échange des ontologies et des instances associées.

2.1.1 RDF/RDF-Schéma

Il s'agit ici de deux formalismes développés pour expliciter par annotation (une partie) de la sémantique du Web. RDF définit à la fois une syntaxe, utilisable ensuite pour tous les formalismes du Web (dont RDF-Schéma et OWL) et un mécanisme d'annotation permettant ensuite d'annoter les éléments existants du Web, qualifiés de ressources. Il ne s'agit donc pas d'un formalisme d'ontologie.

RDF-Schéma étend RDF pour permettre de définir des classes et des propriétés devenant donc un formalisme simple de définition d'ontologies. RDF-Schéma peut utiliser la syntaxe RDF. Il utilise aussi RDF pour annoter les ressources du Web qui seront représentées comme instances des classes de l'ontologie RDF-Schéma.

2.1.1.1 RDF

RDF (*Resource Description Framework*) est le premier langage apparu pour définir la sémantique sur le Web. A l'origine, il était essentiellement destiné à associer aux documents du Web des annotations sémantiques (titre, auteur, ...) exploitables par machine. Puis, l'utilisation des annotations a été étendue à toute information pouvant être référencée sur le Web (site Web complet, page Web, ou encore un élément particulier d'une page Web) et, l'information que l'on voulait représenter a été étendue à la signification de ressource Web. La syntaxe du formalisme RDF est

utilisée par les autres formalismes d'ontologies Web en particulier pour décrire les individus des classes des ontologies.

Le développement de RDF a été motivé par la perspective de :

- manipuler des méta-données Web, afin de fournir des informations sur les ressources Web et les systèmes qui les utilisent ;
- faciliter la recherche et le traitement automatique de l'information du Web par la coopération (indexation, classement, diffusion, ...) des agents logiciels qui exploitent ces méta-données.

Un modèle RDF est défini à partir de quatre ensembles :

1. Les *Ressources*. Une ressource est tout élément que l'on peut référencer par un identifiant appelé URI.
2. Les *Littéraux* qui sont des valeurs éventuellement typées par un des types de données primitif défini par XML Schéma.
3. Les *prédicats*. Un prédicat est une propriété, un aspect, une caractéristique, un attribut ou une relation spécifique que l'on peut utiliser pour décrire une ressource.
4. Les *déclarations* qui permettent de décrire tout élément selon un mécanisme particulièrement simple. Une déclaration est un triplet de la forme : *sujet*, *prédicat*, *objet* ; où *sujet* est une ressource, *prédicat* est une propriété, et *objet* est soit une ressource, soit un littéral. Par exemple : *La France est un Territoire* est une déclaration RDF possible.

Un modèle RDF est un graphe orienté étiqueté (car l'objet d'une déclaration peut être le sujet d'une autre) dans lequel les nœuds sont des ressources (ou des littéraux) et les arcs représentent les prédicats.

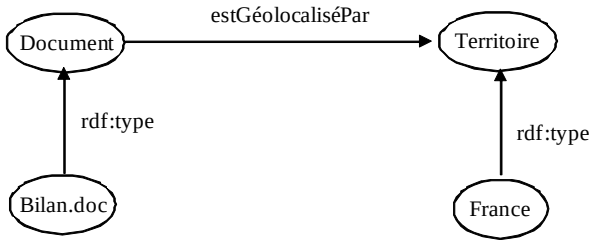


FIG. 1.1 – Exemple d'un graphe RDF.

Exemple : La figure 1.1 illustre un exemple de graphe RDF. Les deux noeuds *Territoire* et *Document* représentent des ressources, et le nœud est un littéral. Les arcs représentent les prédicats. Chaque arc est orienté du sujet vers l'objet de la déclaration. Cet exemple sera utilisé et enrichi tout au long de ce mémoire.

RDF représente toute information par un ensemble de déclarations. Il ne permet cependant pas de définir des vocabulaires permettant de formuler des contraintes sémantiques plus riches (par exemple, de définir l'ensemble des valeurs permises pour une propriété) ou de faire des raisonnements. De plus, RDF ne permet pas de catégoriser le domaine modélisé en termes de classes et de propriétés. Ce n'est donc pas un formalisme d'ontologie. Il fournit par contre une syntaxe et un langage simple pour annoter les ressources du Web. Ce langage est notamment utilisé :

- pour représenter les constructions de l'ensemble des langages d'annotation du Web (XML, HTML, ...),
- par les langages d'ontologies du Web (DAML+OIL, RDF-Schéma, OWL, ...) pour représenter les caractéristiques des ressources en représentant celles-ci comme des instances de classe d'ontologie.

Afin de caractériser l'ensemble de ressources du Web, le formalisme RDFS-Schéma a été proposé.

2.1.1.2 RDFS-Schéma

RDFS-Schéma, aussi appelé RDFS est le premier formalisme d'ontologie défini sur le Web. RDFS-Schéma fournit les prédicats essentiels pour représenter (en RDF) une ontologie. Ces prédicats prédéfinis pourront alors être utilisés afin de définir des ontologies RDFS (aussi appelé vocabulaire RDF) et caractériser ainsi les ressources du Web. RDFS-Schéma est un des piliers du Web sémantique. Grâce à RDFS-Schéma, il est par exemple possible de définir que le concept de *Territoire* dans le vocabulaire intitulé « espace géographique », représente une zone géographique. Une fois que ce vocabulaire est formellement défini grâce à RDFS-Schéma, n'importe quel outil peut désormais utiliser le fait que *Territoire* est un cas particulier de zone géographique. Les données de tels outils pourront être publiées sur Internet et faire l'objet d'une indexation par un autre outil connaissant ce vocabulaire : les utilisateurs de ce dernier outil pourront donc par exemple, lister tous les documents qui font référence à (sont géolocalisés par) la zone géographique France. RDF-Schéma est un système de typage pour RDF. L'utilisation conjointe de RDF et RDF-Schéma dans le Web Sémantique permet donc à la fois de représenter (en RDF-Schéma) une ontologie et (en RDF) des instances définies en termes de cette ontologie.

RDF-Schéma est le plus simple formalisme d'ontologie. Il est doté du nombre minimum de constructeurs nécessaires à la définition d'une ontologie. Ces constructeurs vont être retrouvés dans tous les autres langages d'ontologie du Web (DAML+OIL, OWL).

Constructeurs de classes

Pour modéliser les ressources du Web, RDFS permet de :

- définir des classes (*rdfs :class*) : une classe est un ensemble défini en intension de plusieurs objets analogues d'un certain point de vue et que l'on souhaite regrouper. Exemple : la classe des *territoires*.
- organiser les classes en une hiérarchie de spécialisation (*rdfs :subclassOf*). Exemple : la classe *Commune* peut être définie comme une sous-classe de la classe *Territoire*.

Constructeurs de propriétés

RDFS permet aussi de :

- définir des propriétés (*rdfs :property*). Exemple : *code_*ISO est une propriété de la classe *Territoire*.
- organiser les propriétés en une hiérarchie de spécialisation((*rdfs :subProperty*)). Exemple : *a_pour_fils* est une sous-propriété de *a_pour_enfants*.

RDFS définit également :

- La ou les classes auxquelles ont peut affecter une propriété et, qui constituent le domaine [3] de la propriété (*rdfs :domain*) (par exemple : la classe *Document* peut être l'objet de la propriété *estGeolocalisePar*). Lorsque le domaine d'une propriété n'est pas défini, cette dernière peut être utilisée pour décrire n'importe quelle instance appartenant à l'univers du discours conceptualisé par l'ontologie.
- le co-domaine, ou le domaine de valeurs d'une propriété, (*rdfs :range*) (par exemple : la propriété *estGeolocalisePar* à pour valeur un élément de la classe *Territoire*). Comme la définition du domaine, la définition du co-domaine d'une propriété peut être défini par une ou plusieurs classes (avec le sens d'intersection) [4], les types de données peuvent également être utilisées.

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 de données prédéfinis de XML Schéma. Ceci permet par exemple de représenter des valeurs de types chaîne de caractères, numérique ou date. Par ailleurs, RDFS fournit également des types collections (*rdfs :Container*). RDFS utilise des constructeurs de collections définis dans RDF, en particulier les listes (*rdf :List*) et les sacs (*rdf :Bag*).

Constructeurs d'individus

Les instances des classes RDFS sont définies en RDF, par deux types de triplets.

1. Les triplets de la forme *(i, rdf :type, C)* indiquent que i est une instance de la classe C. RDFS supporte la multiinstanciation qui permet à une instance d'appartenir à plusieurs classes même si ces classes ne sont pas liées par une relation de subsumption (spécialisation).
2. Les autres triplets, de la forme *(i, p, v)*, caractérisent l'instance i par la valeur v pour la propriété p.

Axiomes

RDFS ne permet pas de définir de nouveaux axiomes en dehors de l'appartenance à une classe ou la subsumption de classe. Par contre, un modèle RDFS est lui-même basé sur certains axiomes :

1. RDFS permet la méta-modélisation. En effet, il n'impose pas de disjonction entre l'ensemble des classes et l'ensemble des instances ; ainsi, une information peut à la fois être représentée comme une classe et comme une instance en fonction de son rôle dans un contexte particulier.
2. RDFS ne permet pas la définition de références circulaires dans la définition de la subsumption des classes et des propriétés. Une propriété ne peut être sous-propriété d'elle-même ou d'une de ses sous-propriétés.

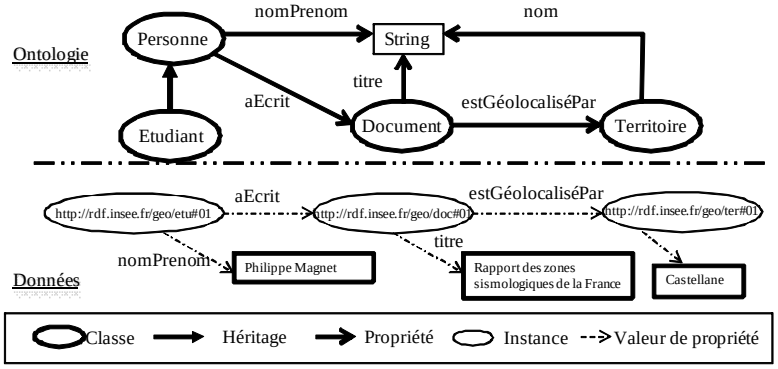
Autres concepts

[3] Lorsque plusieurs classes sont spécifiées, le domaine de la propriété correspond à l'intersection de ces classes

[4] Si plusieurs classes sont utilisées, seules les instances qui appartiennent à l'intersection de ces classes peuvent se voir affectées la propriété.

- RDFS définit la classe *ressource* comme classe mère de toute classe. Tout est une ressource dans le Web sémantique, sauf la notion de littéral. Et, toute classe est une sous-classe de la classe ressource.
- Les classes, les propriétés et les instances peuvent également être décrites de façon textuelle en utilisant les attributs *rdfs :label* et *rdfs :comment*, *rdfs :seeAlso*, *rdfs :isDefinedBy*.

La figure 1.2 montre un exemple d'ontologie présentée sur la forme d'un graphe. Cette ontologie comporte les classes *Personne*, *Etudiant* (sous classe de *Personne*), *Document* et *Territoire*. En dessous de cette ontologie, nous avons présenté des données de cette ontologie. Les URI des instances sont entourées par des ovales tandis que les valeurs littérales sont représentées dans des rectangles. Enfin, nous présentons un extrait de l'ontologie et des données suivant la syntaxe RDF/XML qui est une syntaxe XML pour les ontologies RDFS. Nous voyons notamment avec cette syntaxe que la propriété *nom* de la classe *Personne* représente le nom d'usage de la *Personne*. Cette précision est fournie par le constructeur *rdfs :label*. Notons que, comme le montre cette figure, il est possible de préciser avec le constructeurs *rdfs :label* un attribut *xml :lang* qui renseigne sur la langue utilisée dans la description du label.



Ontologie (cog="http://rdf.insee.fr/geo#)

```
<rdfs:class rdf:id= " cog:Personne" />
<rdfs:class rdf:id= " cog:Document" />
<rdfs:class rdf:id= " cog:Etudiant" >
        <rdfs:subclassof rdf:about= " cog:Personne" />
</rdfs:class>
<rdf:Property rdf:id= " cog:nomPrenom" >
        <rdfs:label xml:lang= "fr">nom d 'usage</rdfs:label>
        <rdfs:domain rdf:about= "cog:Personne" / >
        <rdfs:range rdf:resource= "xsd:string " />
</rdf:Property>
```

Données

```
<cog:Document rdf:about= " http://rdf.insee.fr/geo/doc#01"/ >
<cog:Etudiant rdf:about= " http://rdf.insee.fr/geo/pers#01" />
        <cog:nomPrenom> Philippe Magnet </cog:nomPrenom>
        <cog:aEcrit rdf:resource = " http://rdf.insee.fr/geo/doc#01"/>
</cog:Etudiant>
```

FIG. 1.2 – Exemple d'ontologie exprimée en RDF Schéma.

RDFS a étendu RDF par un ensemble de constructeurs permettant la définition d'ontologies sur le Web. Cependant, RDFS ne contient aucune primitive explicite permettant de décrire des équivalences conceptuelles. Les ontologies définies suivant le formalisme RDFS sont donc être composées de concepts canoniques uniquement. Dans la section suivante nous présentons le formalisme PLIB qui dans le domaine de l'ingénierie permet de définir des ontologies canoniques beaucoup plus complètes.

2.1.2 PLIB

PLIB (Parts LIBrary : series de normes ISO 13584), initié en 1987, est un formalisme d'ontologie conçu initialement dans le cadre du domaine technique pour échanger et modéliser avec la plus grande précision possible les différentes catégories (classes) de composants industriels et leurs instances telles qu'elles sont, par exemple, décrites dans les catalogues. Pour cela, une ontologie PLIB définit de façon très fine, les catégories et les propriétés qui caractérisent les objets d'un domaine du monde réel, ainsi que les abstractions que les différentes communautés peuvent en construire [56]. Enfin, PLIB fournit des opérateurs de modularité permettant d'intégrer dans un environnement homogène et cohérent les ontologies partiellement hétérogènes définies par différentes sources.

Nous présentons ci-dessous les différents constructeurs du formalisme d'ontologie PLIB.

2.1.2.1 Constructeurs de classes

Le modèle PLIB permet de déclarer des classes et de les organiser en des hiérarchies de subsumption. Une classe PLIB peut être définie comme étant :

- une classe de définition (*item_class*) qui contient les propriétés essentielles d'une classe, c'est a dire les propriétés communes qui caractérisent les instances de la classe pour tous les acteurs qui utilisent l'ontologie ;
- une classe de représentation (*functional_model_class*) qui contient les propriétés qui n'ont de sens que par rapport à un point de vue « métier »(par exemple le nombre d'instances qui existent en stock, pour le gestionnaire de stock, ou le taux de remise par quantité, pour le commercial) ;
- une classe de point de vue (*functional_view_class*) qui définit la perspective dans laquelle sont définies les propriétés des classes de représentation (par exemple : gestion de stock, conditions commerciales, ...).

La hiérarchie des classes PLIB est définie par :

- la relation sémantique de subsumption nommée *is_a* qui définit une hiérarchie (simple) avec factorisation/héritage des propriétés ;
- une deuxième relation sémantique nommée *is_case_of* qui permet également d'exprimer la subsumption entre classes. Celle-ci n'est cependant pas codée par le mécanisme d'héritage. La relation sémantique *is_case_of* permet d'indiquer qu'une classe est incluse dans une autre classe (subsumption) mais qu'elle souhaite, au niveau logique, n'importer explicitement qu'une partie des propriétés de cette dernière.

Le mécanisme *is_case_of* permet de construire des ontologies modulaires qui n'importent des autres ontologies du domaine que certaines classes et pour chacune de ces classes, que le sous-ensemble des propriétés nécessaires pour l'objectif visé. Ainsi, ce mécanisme permet la définition d'ontologies autonomes qui restent toutefois articulées aux autres ontologies du domaine par subsumption. Cette articulation formelle va ainsi permettre de partager et d'échanger ce qui est commun. La relation de subsumption avec importation sélective de propriétés *is_case_of* permet aux constructeurs de redéfinir entièrement la structure des classes en fonction des besoins particuliers que vise à résoudre l'ontologie mise en œuvre. Elle permet aussi d'assurer l'autonomie tant structurelle (superclasse, propriété) que temporelle (évolution éventuelle des autres ontologies du domaine) d'une ontologie. En effet, l'ontologie en cours de définition ne contient pas directement

les classes importées des autres ontologies. Elle contient seulement des classes spécifiques qui lui sont propres et qui sont reliées aux autres ontologies par des relations de subsumption. Ces classes importent explicitement certaines des propriétés de la classe subsumante qui s'avèrent pertinentes pour le système en cours de construction et ceci, dans une version temporelle bien définie. De plus, PLIB ne fournissant que des constructeurs canoniques, les ontologies PLIB ne comportent que des concepts primitifs et représentent donc des modèles canoniques d'échanges.

2.1.2.2 Constructeurs de propriétés

La modèle PLIB permet également de déclarer des propriétés en tenant compte du fait que la valeur d'une propriété peut dépendre d'une, ou plusieurs autres propriétés. Une propriété PLIB peut être définie comme étant :

- une propriété autonome (*non_ dependent_ p_ det*) ; sa valeur est indépendante de toute autre propriété ;
- un paramètre de contexte (*condition_ det*) ; qui décrit le contexte dans lequel peut être situé un objet (exemple : la température environnante) ;
- une propriété dépendante (*dependent_ p_ det*) dont la valeur dépend de paramètres de contexte (par exemple, la longueur d'une tige métallique dépend de la température environnante) ;
- une propriété de représentation définie dans une classe de point de vue ou représentation (*representation_ det*) ; La valeur d'une telle propriété peut changer sans que cela ne change l'objet décrit (exemple : le nombre d'instances existant en stock).

Toute propriété PLIB précise obligatoirement son domaine c'est-à-dire la classe dans laquelle l'affecter (*name_ scope*), ainsi que son domaine de valeurs ou co-domaine. C'est la notion de *typage fort* des propriétés. Ceci est différent de RDFS où domaine et co-domaine peuvent ne pas être précisés. Le co-domaine d'une propriété est défini par le constructeur *data_ type*. Ce dernier peut être une classe (*class_ instance_ type*), une collection (*set_ type, bag_ type, list_ type, array_ type*) ou un autre type de données.

En pratique, le domaine de valeurs d'une propriété est souvent composé de plusieurs classes qui peuvent se trouver dans plusieurs branches de la hiérarchie des classes. Pour résoudre ce problème, PLIB propose de qualifier les propriétés de la façon suivante.

- Une propriété est définie au plus haut niveau de la hiérarchie (is_a) où l'on peut la définir sans ambiguïté ; elle est dite *visible* pour tout le sous-arbre correspondant.
- Une propriété visible en un nœud peut y devenir *applicable* ; cela signifie qu'elle est rigide c'est à dire essentielle pour tout objet de la classe et pour toute sous-classe : toute instance réelle doit présenter une valeur ou une grandeur qui représente cette propriété. Ceci ne signifie pas que toutes les valeurs doivent être présentes dans la représentation informatique puisque ceci dépend uniquement des besoins de utilisateur.

2.1.2.3 Identification de concepts

Afin de pouvoir référencer de façon non ambigüe et multilingue n'importe lequel de ces concepts (classes et propriétés), PLIB comporte un schéma d'identification universel (GUI : Globally Unique Identifier). Chaque organisation qui est la source d'une ontologie est associée à un identificateur unique (en général préexistant pour toute organisation ou établissement ; par exemple en France

il peut en particulier, être construit sur les codes SIRET ou SIRENE). Le BSU (Basic Semantic Unit) dans PLIB est obtenu par concaténation d'un certain nombre d'attributs (*code, version, révision*, ...). Chaque source doit alors attribuer un code unique à chacune des classes qu'elle définit. Enfin le code d'une propriété doit être unique pour une classe et toutes ses sous-classes. La concaténation de ces informations permet alors d'identifier de façon unique et universelle chacun des concepts ci-dessus. C'est ce simple code, appelé un BSU, qu'il sera suffisant de référencer pour caractériser une classe ou une propriété PLIB. Ce code joue un rôle identique à l'URI de RDF/RDFS.

2.1.2.4 Types de données

Le schéma du formalisme d'ontologie PLIB fournit un système de types permettant de représenter :

- des types de données primitifs tels que les entiers, les réels, etc ;
- des types de données énumérés (*non_ quantitative_ code_ type, non_ quantitative_ int_ type*) ;
- des aggrégats (*list_ type, set_ type, bag_ type, array_ type*) ;
- des données qualifiées, par exemple des unités ou des monnaies (*int_ currency_ type*, *int_ measure_ type*, ...).

2.1.2.5 Constructeurs d'individus

PLIB propose pour définir les individus (ou instances) des classes, d'en énumérer les instances. Chaque instance est catégorisée par les classes (liées par la relation de subsumtion) auxquelles elle appartient et elle est caractérisée par les valeurs prises pour certaines des propriétés définies sur cette classe.

Notons que une ontologie décrivant les propriétés d'une classe indépendamment de toute utilisation dans un contexte particulier, les propriétés à utiliser pour décrire les instances de classe des ontologies PLIB dans tout contexte particulier sont choisies par l'utilisateur parmi les propriétés applicables. Il n'y a aucune contrainte sur les propriétés devant être utilisées. PLIB ne permet pas la multi-instanciation. A la place, PLIB offre le mécanisme d'agrégation d'instances. Pour cela, au travers des constructeurs de classes définis à la section 2.1.2.1,PLIB propose de distinguer les propriétés essentielles d'une classe de celles qui dépendent d'un point de vue sur le concept représenté. Ainsi, une instance peut appartenir non seulement à sa classe de base mais également aux classes de représentation attachées à cette classe de base. 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, sauf les propriétés définies comme communes qui permettent de réaliser la jointure.

Pour illustrer cette approche de gestion par le formalisme PLIB de la multi-instanciation, considérons par exemple la planète Mars. Cette dernière peut être décrite comme une instance de la classe Planète par les deux propriétés autonomes nom et couleur. Elle peut également être décrite suivant le point de vue d'un système d'information géographique (SIG) ou encore du point de vue d'un logiciel de cartographie.

- Du point de vue d'un SIG, la planète Mars peut être modélisée comme étant un vecteur de points correspondant à un réseau de triangles obtenu par triangularisation. Cette représentation serait ensuite utilisée pour réaliser des calculs mathématiques divers tels que son

volume, etc.
- Du point de vue d'un logiciel de cartographie, la planète Mars peut être vue comme une matrice d'altitudes obtenue par un balayage de sa surface. Cette représentation serait ensuite utilisée pour réaliser des aperçus graphiques ou encore pour déterminer la régularité de sa surface.

2.1.2.6 Axiomes

PLIB définit un grand nombre d'axiomes que doivent vérifier les ontologies et les individus associés pour être valides.
- Chaque concept d'une ontologie (classes et propriétés) est associé à un identifiant unique.
- Un individu appartient à une seule classe de base (et ses super-classes).
- Le graphe de subsumption *is_a* est constitué d'une forêt, il n'y a ni héritage multiple, ni cycle dans le graphe.
- Seules les propriétés applicables à une classe peuvent être utilisées pour décrire ses instances.
- La valeur de toute propriété appartient à son co-domaine.

Ces axiomes permettent de définir des contraintes d'intégrité qui seront exploitées par le système pour valider les données. Ainsi, si une instance est associée à une propriété qui n'est pas applicable à sa classe, le système l'interprète comme une erreur et signale une incompatibilité entre l'ontologie et les données.

2.1.2.7 Autres concepts

PLIB permet de construire des ontologies multilingues ; c'est à dire où le même identifiant de concepts est associé à des descriptions dans plusieurs langues. Une ontologie, tout comme les classes et les propriétés, peut être associée à des attributs. Les attributs permettent d'associer à chaque concept une définition, un nom préféré, un nom court, des noms synonymes, une note, une remarque, une image ou encore une source d'information extérieure qui peut être un document.

2.1.2.8 Synthèse sur PLIB

Le formalisme PLIB permet de définir de manière précise les concepts du domaine modélisé. Pour cela il fournit comme RDF-Schéma des constructeurs permettant de catégoriser les informations du domaine en termes de classes et de les caractériser par des propriétés. Toutefois, PLIB dispose d'un plus grand nombre de constructeurs (de classes et de propriétés) que RDFS. Ces constructeurs permettent en particulier d'exprimer le contexte dans lequel est évaluée l'information et de représenter explicitement une unité de mesure. PLIB fournit un opérateur de modularité qui permet de définir des articulations formelles entre ontologies. Contrairement à RDFS où les ensembles des classes et des individus ne sont pas disjoints, PLIB impose la distinction entre les concepts de l'ontologie (classes et propriétés) et les individus. Également, PLIB ne supporte pas la multi-instanciation ; à la place il adopte une représentation des données suivant un mécanisme de point de vue.

Notons que comme RDFS, les constructeurs offerts par PLIB sont des constructeurs canoniques, les seuls raisonnements qui peuvent être effectuées sur les ontologies RDFS et PLIB sont donc le test de subsumption des classes de l'ontologie et, le test d'instanciation des individus . L'objectif de

PLIB n'est pas de raisonner mais d'assurer la qualité des données. Dans la spécification formelle de PLIB, un grand nombre d'axiomes sont définis qui permettent d'identifier les erreurs de données. A contrario, et à la différence des formalismes orientés inférence que nous allons présenter ci-dessous, PLIB ne fait pas l'hypothèse que les données sont correctes de façon à pouvoir déduire de nouvelles connaissances de ces données. C'est précisément l'hypothèse sur laquelle se fondent les formalismes d'ontologies orientés inférence que nous présentons dans la section suivante.

2.2 Formalismes d'ontologies orientés inférence

Le souhait de pouvoir réaliser des inférences a conduit à la définition d'autres types de formalismes de modélisation d'ontologies. En particulier dans le domaine du Web, le pouvoir d'expression du formalisme d'ontologie RDFS a été étendu par un ensemble d'autres formalismes d'ontologies parmi lesquels OIL, DAML-ONT, DAML-OIL et OWL. Ceci afin de concevoir des ontologies permettant de réaliser davantage de raisonnements (en plus du test subsumption) sur les ontologies et les données.

OWL (Ontology Web Language), créé en 2001 par le W3C, s'est imposé comme le langage standard pour représenter des ontologies dans le Web. Après une brève description des formalismes OIL, DAML-ONT, DAML-OIL dans la section 2.2.1, nous nous intéresserons dans la section 2.2.2, au standard OWL développé tout particulièrement pour décrire les ressources du Web. Celui-ci introduit à la fois des équivalences conceptuelles et des hypothèses de correction des données permettant de réaliser des inférences.

2.2.1 DAML-ONT, OIL, DAML+OIL

Afin d'étendre le pouvoir d'expression de RDFS qui ne fournit que des mécanismes primitifs pour spécifier des classes, deux initiatives ont été lancées.

La première est une initiative américaine lancée par la DARPA (Defense Advanced Research Projects Agency), a permis la spécification du langage DAML-ONT (DARPA Agent Markup Language)[5], fondé sur RDF, et proche des langages objets.

La seconde est une initiative sponsorisée par la communauté européenne projet « On-To-Knowledge », a permis la spécification du formalisme OIL (Ontology Inference Layer) [17] , basé sur XML et, offrant des primitives inspirées des logiques de description.

La fusion de ces deux langages a donné naissance au formalisme DAML+OIL [15] qui cherche à combiner toutes les caractéristiques de DAML-ONT, d'OIL, de RDFS, et de RDF. DAML+OIL a servi de base pour la conception du formalisme OWL standardisé par le W3C.

2.2.2 OWL

OWL est une recommandation du W3C[6] pour indexer, à l'aide d'ontologies, les ressources du Web. L'objectif de cette approche est de permettre d'automatiser l'interprétation de la signification des ressources du Web en annotant ces ressources avec des termes ontologiques traitables en machine. OWL, permet à la fois d'annoter les données et de faire certains raisonnements sur

[5]http ://www.daml.org/

[6]World Wide Web Consortium. http ://www.w3.org/

les données. Pour cela, le formalisme OWL offre en plus des constructeurs du formalisme RDFS, d'autres constructeurs qui vont permettre de réaliser des raisonnements.

Un des objectifs importants de OWL est de pouvoir assurer que les raisonnements qui peuvent êtres réalisés sur les ontologies 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 fournis par une ontologie OWL peuvent être déduites d'une autre ontologie. L'impossibilité d'obtenir un formalisme à la fois compatible avec RDFS et dont les raisonnements associés soient décidables a poussé les concepteurs du formalisme d'ontologie OWL à en spécifier trois sous-formalismes de compatibilité et d'expressivité croissante mais, de décidabilité décroissante : OWL Lite, OWL DL et OWL Full.

2.2.2.1 Les influences de OWL

La conception du langage OWL a été influencée à la fois par (1) les langages préexistants sur le Web, en particulier RDF et RDFS, (2) la logique de description et (3) les langages de frames.

- La première et principale source d'influence de OWL est issue de ses prédécesseurs. Parmi les impératifs de conception du langage OWL figure la compatibilité avec RDF. Celle-ci se traduit par l'utilisation de RDF/XML comme syntaxe concrète de OWL. OWL a également été inspiré de DAML+OIL pour sa sémantique.
- Le domaine de la logique de description (Description Logics) est la seconde source d'influence de OWL. Elle a, d'une part, influencé le choix de spécifier la sémantique du langage par la théorie du modèle. D'autre part, les études sur la complexité menées dans ce domaine ont également orienté les choix des constructions du langage. En effet, un autre objectif important de OWL est de pouvoir assurer qu'il est possible de lui associer un moteur d'inférence décidable basé sur ce langage.
- Enfin, la troisième source d'influence est le domaine des Frames (Frames paradigm) qui consiste à regrouper l'ensemble des informations qui se rattachent à un même élément. Ceci a influencé la structuration de la syntaxe abstraite de OWL.

Nous décrivons, dans ce qui suit, les constructeurs offerts par OWL Lite qui est le sous-ensemble d'OWL le moins expressif. Les limitations apportées à OWL Lite par rapport à OWL DL et OWL Full, produisent un sous-ensemble pratique et minimal des caractéristiques du formalisme OWL dont la mise en œuvre est la plus simple pour les développeurs d'outils d'inférence et dont les possibilités semblent suffisantes pour beaucoup de cas d'applications.

2.2.2.2 Constructeurs de classes

OWL Lite permet de déclarer des classes, les organiser en une hiérarchie de subsumption comme RDFS et PLIB. Une classe OWL Lite peut être spécifiée comme étant :

- une classe nommée (en utilisant le constructeur *owl :Class* avec une URI) ;
- une expression de classe comme décrite ci-dessous, qui est une classe anonyme.

Une expression de classe est soit :

1. Une restriction de propriété. Il existe des restrictions de propriétés locales qui restreignent le co-domaine de certaines propriétés uniquement pour la classe où elles sont spécifiées.
   - $\exists$ P.C (*owl :someValuesFrom*) : au moins une des valeurs de P est une instance de la classe C.

– $\forall$ P.C (*owl :allValuesFrom*) : toutes les valeurs de P sont des instances de la classe C.

2. Une restriction de cardinalité qui restreint localement le nombre de valeurs distinctes d'une propriété P. On distingue :
   (a) Les restrictions non typées qui ne précisent pas le type du co-domaine.
   – ≥ 1 P (*owl :minCardinality*) : P possède au moins 1 valeur.
   – ≤ 1 P (*owl :maxCardinality*) : P possède au plus 1 valeur.
   – =1 P (*owl :Cardinality*) : P possède exactement 1 valeur.
   (b) Les restrictions typées qui précisent le type du co-domaine.
   – ≥ 1 P.C (*owl :minCardinalityQ*) : P possède au moins 1 valeur de type C.
   – ≤ 1 P.C (*owl :maxCardinalityQ*) : P possède au plus 1 valeur de type C.
   – $= 1$ P.C (*owl :CardinalityQ*) : P possède exactement 1 valeur de type C.
3. Un constructeur booléen de classes
   – $C_1 \cap C_2 \cap \ldots \cap C_n$ (*owl :intersectionOf*) : la sémantique de l'intersection est identique à celle de l'instanciation multiple (c est instance de l'intersection des C_j est équivalent à c est instance de chacune des classes C_j).

La hiérarchie des classes OWL Lite est définie par :

1. C $\subseteq$ D (*rdfs :subClassOf*) : qui équivaut à la subsumption en logique de description. Les seuls objets qui peuvent être instances de la sous-classe C sont ceux qui sont instances de la super-classe D.
2. C $\equiv$ D (*owl :EquivalentClass*) : cet axiome équivaut à ((C $\subseteq$ D) $\wedge$ (C $\supseteq$ D)).

2.2.2.3 Constructeurs de propriétés

OWL Lite permet aussi de déclarer des propriétés et de les organiser en hiérarchie de *sous-propriétés*. Une propriété OWL Lite peut être spécifiée comme étant :

1. Une propriété de type simple (*owl :datatypeProperty*) : son co-domaine est alors un type de données issu de la spécification XML Schema (string, int, real, boolean, ...).
2. Une propriété de type objet (*owl :ObjectProperty*) : son co-domaine est une classe de l'ontologie.

Contrairement à l'approche utilisée par les langages de programmation orientés objets, l'association des propriétés aux classes se fait lors de la définition du domaine de la propriété (*rdfs :domain*). Si aucune classe n'est spécifiée, la propriété s'applique à la classe à *owl :Thing* (c'est-à-dire à l'ensemble des classes de l'ontologie) qui est la classe racine de toute ontologie OWL. De la même manière, une propriété peut préciser explicitement son co-domaine (*rdfs :range*), sauf qu'en plus des classes, des types de données peuvent également être utilisés.

Comme pour les classes, OWL Lite définit des constructeurs pour hiérarchiser les propriétés :

1. $\subseteq$ (*rdfs :subPropertyOf*) : qui équivaut à la subsumption de propriétés (inclusion des extensions).
2. $\equiv$ (*owl :Equivalentproperty*) : qui spécifie que des propriétés ont la même extension.

En plus, OWL Lite supporte différentes caractéristiques (mathématiques) associées aux propriétés :

1. *owl :TransitiveProperty* : définit une propriété transitive, comme *« seSubdiviseEn »*.
2. *owl :SymmetricProperty* : définit une propriété symétrique, comme *« estVoisinDe »*.
3. *owl :InverseOf* : définit une propriété comme étant l'inverse d'une autre propriété : *« aPourPère »* et *« aPourEnfant »*.
4. *owl :FunctionalProperty* : la propriété ne pas avoir plus d'une valeur comme *« estNéLe »*.
5. *owl :InverseFunctional* : une unique ressource peut être associée à la valeur de cette propriété : *« aPourNoSecuriteSociale »*.

2.2.2.4 Identification des concepts

Une des caractéristiques principales de OWL est son mécanisme d'identification : pour supporter la multiplicité des ontologies sur le Web, un unique identificateur doit être associé à chaque ressource (classe, instance, propriété). Pour cela, OWL utilise les URI (Uniform Ressource Identifiers). L'utilisation des URI permet à tout constructeur d'ontologie de référencer et d'utiliser les ressources d'autres ontologies. Néanmoins, à la différence de PLIB, l'importation n'est pas modulaire. Les ontologies OWL doivent importer globalement toute autre ontologie dont elle référence au moins un concept.

2.2.2.5 Types de données

Les types de données supportés par OWL Lite sont :

1. Les types de données primitifs issus de la spécification XML Schema (*xsd :string*, *xsd :boolean*, ...).
2. Le type de donnée RDFS intégré *rdfs :Literal*.

2.2.2.6 Constructeurs d'individus

Le constructeur d'individus OWL est : *owl :individual*. Les instances d'une classe sont définies en explicitant la ou les classes auxquelles elles appartiennent et en indiquant les valeurs de propriétés sous la forme d'une liste de couples (propriété, valeur). OWL permet de définir les assertions suivantes sur individus :

1. i ∈ C (*rdf :type*) : permet de déclarer que l'individu i appartient à la classe C
2. $<i_1, i_2> \in$ P, $<i_1$, v$>$ ∈ P : définie la valeur d'une propriété de type objet (l'individu i_1 a comme valeur pour la propriété P l'individu i_2) et de type simple respectivement (i_1 a la valeur simple v pour la propriété P.
3. $i_1 \equiv i_2$ (*owl :sameAs*) : égalité entre les individus i_1 et i_2.
4. $i_1 \neq i_2$ (*owl :differentFrom*) : différence entre les individus i_1 et i_2.

2.2.2.7 Axiomes

Le formalisme OWL Lite définit un certain nombre d'axiomes qui sont des règles de typage. Il impose que [7] :

– les sujets des triplets *rdfs :subClassOf* et *owl :equivalentClass* soient des noms de classe et leurs objets des noms de classe ou des restrictions ;

[7] http ://www.w2.org/TR/owl-ref/

- les relations *owl :intersectionOf* soient seulement utilisées sur des listes dont la longueur est supérieure à un et qui ne contiennent que des noms de classe ou des restrictions ;
- les objets des triplets *owl :allValuesFrom* et *owl :someValuesFrom* soient des noms de classe ou des noms de type de données ;
- les objets des triplets *rdf :type* soient des noms de classe ou des restrictions ;
- les objets des triplets *rdfs :domain* soient des noms de classe, et
- les objets des triplets *rdfs :range* soient des noms de classe ou des noms de type de donnée.

Le formalisme OWL défini également des axiomes plus généraux. Comme nous l'avons vu à la section 2.2.2.1, le formalisme OWL a été influencé par les logiques de description. Ces derniers ont servis à offrir au formalisme OWL un certains nombre de constructeurs (de classes et de propriétés) à la base du raisonnement sur les ontologies OWL. Trois principaux choix ont été adoptés pour ouvrir les raisonnements possibles :

1. La définition d'équivalences conceptuelles entre classes nommées et/ou anonymes permettant de remplacer chaque classe définie par sa définition formelle. Par exemple, avec la description « *Femme* ≡ *Personne* ∩ =*genre 'féminin'* », pour déterminer l'ensemble des individus de la classe *Femme*, un interpréteur OWL retournera à la fois tous les individus explicitement définis comme des instances de la classe *Femme*, et tous les individus satisfaisant à la description « *Femme* ≡ *Personne* ∩ =*genre 'féminin'* », c'est à dire toutes les instances de la classe *Personne* dont le genre est féminin.
2. Le fait que plusieurs identifiants puissent être associés à un même objet. En effet, à la différence de PLIB, OWL ne fait pas l'hypothèse de l'unicité de nommage. Le fait que deux individus aient deux identifiants différents n'implique pas qu'ils soient des individus différents. Par exemple, si la propriété *estFilsDe* indique qu'une personne a au plus un père biologique, si l'on déclare que la personne *Paul Dupont* a pour père biologique les personnes *Magnet* et *Thomas*, un interpréteur OWL ne conclut pas à une erreur mais à l'égalité des individus *Magnet* et *Thomas*. Pour indiquer que deux individus sont différents, il faut le spécifier explicitement avec le constructeur *differentFrom*. Notons à la différence de PLIB, que si le fait de déclarer que *Magnet* est le père de Paul Dupont résulte d'une erreur, (*Magnet* est sa mère), cette erreur ne sera pas détectée. Au contraire, le raisonnement aura ajouté une deuxième erreur (*Magnet* = *Thomas*) qui pourra fausser encore d'autres raisonnements.
3. Le fait qu'une même instance puisse appartenir à plusieurs classes. Si par exemple, une instance est caractérisée par une propriété P qui n'est pas applicable à sa classe de base (et ses super-classes), l'interpréteur OWL conclut que cet individu appartient également à une autre classe (inconnue) qui fait partie du domaine de la propriété P. Ici également, s'il s'agit d'une erreur de donnée (par exemple on a confondu le code de deux propriétés), cette erreur n'est pas détectée mais une nouvelle erreur est introduite par inférence.

La richesse de raisonnement de OWLa donc comme contrepartie, en général, d'empêcher d'identifier une erreur de données et d'introduire par inférence, dans un tel cas, de nouvelles erreurs.

2.2.2.8 Autres concepts

Une ontologie peut importer (*owl :imports*) l'ensemble des concepts définis par une autre onto-

logie.

Une ontologie, tout comme les classes, propriétés et instances, peuvent être annotées ; ceci permet d'associer à un tel concept un mot (*rdfs :label*), un numéro de version (*owl :versionInfo*), un commentaire (*rdfs :comment*), des références vers d'autres concepts (*owl :seeAlso*) ou même son créateur (*owl :isDefinedBy*). Ces annotations sont néanmoins beaucoup moins nombreuses et précises que celles de PLIB. Par exemple, il n'existe pas de moyen, en OWL, de préciser l'unité d'une mesure ou de fournir un graphique.

2.2.2.9 Autres sous-formalismes de OWL

OWL fournit plusieurs sous-formalismes qui visent à répondre aux besoins de communautés et d'utilisateurs spécifiques. OWL Lite est le sous-formalisme OWL le plus simple ; c'est une version limitée qui en facilite son utilisation et son implémentation. OWL Lite est basé sur la logique de description SHIF et se destine principalement aux utilisateurs qui ont principalement besoin d'une hiérarchie de classification et de contraintes simples. Par exemple, alors qu'il supporte des contraintes de cardinalité, il autorise seulement des valeurs de cardinalité 0 ou 1. Deux autres sous-formalismes sont définis.

1. OWL DL inclut tous les constructeurs du langage OWL Lite, mais toutes les restrictions qui le concerne (par exemple les valeurs de cardinalité limitées à 0 et 1) sont supprimées, sauf celle qui rendrait le formalisme non décidable (par exemple, une classe ne peut pas être instance d'une autre classe). OWL DL constitue le sous-ensemble maximal du langage OWL par rapport auquel les travaux actuels peuvent garantir l'existence d'une procédure de raisonnement décidable pour un raisonneur. OWL DL est basé sur la logique de description SHOIN. Ce niveau de langage est décidable mais a l'inconvénient de ne pas être complètement compatible avec RDF.

   En plus, des expressions de classes définis par OWL Lite, OWL DL permet de définir une classe comme étant :
   - i_1, i_2, ... , i_n : une énumération d'instance (en utilisant le constructeur owl :oneOf). Exemple : *Genre* owl :oneOf(rdfs :Litteral('masculin'), rdfs :Litteral('féminin')) ;
   - $C_1 \cup C_2 \cup \ldots \cup C_n$ (*owl :unionOf*) : l'union de deux ou plusieurs autres classes.
   - $\neg$C (*owl :ComplementOf*) : le complémentaire d'une autre classe .

   OWL DL définit également un nouvel opérateur de comparaison de classes : C $\neq$ D (*owl :disjointWith*) : cet axiome équivaut à (C $\cap$ D) $\subseteq \perp$.

   OWL DL définit de nouvelles restrictions de propriété (*owl :hasValue* et *owl :valueNot*) et, relâche les restrictions de cardinalité existantes à une valeur entière quelconque.
   - i $\in$ P (*owl :hasValue*) : une des valeurs de la propriété P est l'individu *i*.
   - i $\notin$ P (*owl :valueNot*) : l'individu *i* n'est pas une valeur de P.
   - $\geq$ n P (*owl :minCardinality*) : P possède au moins n valeurs distinctes.
   - $\leq$ n P (*owl :maxCardinality*) : P possède au plus n valeurs distinctes.
   - =n P (*owl :Cardinality*) : P possède exactement n valeurs distinctes.
   - $\geq$ n P.C (*owl :minCardinalityQ*) : P possède au moins n valeurs distinctes de type C.
   - $\leq$ n P.C (*owl :maxCardinalityQ*) : P possède au plus n valeurs distinctes de type C.
   - = n P.C (*owl :CardinalityQ*) : P possède exactement valeurs n distinctes de type C.

OWL DL relâche également la définition du domaine et du co-domaine d'une propriété. En effet, contrairement à OWL Lite, le domaine et le co-domaine d'une propriété OWL DL peuvent être une classe anonyme ou multiple (composés de plusieurs classes). Dans ce dernier cas, ils sont interprétés comme une intersection. Ainsi, si plusieurs classes sont utilisées pour spécifier le domaine d'une propriété, le domaine est composé de l'intersection de ces classes. Il en est de même pour la définition du co-domaine.

OWL DL permet également d'associer la caractéristique reflexive à une propriété :
owl :ReflexiveProperty : définit une propriété réflexive, comme *« estAmiDe »*.

Enfin, OWL DL intègre la possibilité de définir des types énumérées (*owl :datarange*) à valeurs appartenant aux types primitifs XML schéma.

2. OWL Full, qui inclut OWL DL est, par contre, entièrement compatible avec RDFS. Il se destine, typiquement, aux utilisateurs qui veulent combiner l'expressivité du langage OWL à la souplesse et aux capacités de méta-modélisation de RDF. Ce niveau du formalisme OWL n'impose pas, par exemple, la distinction entre l'ensemble des propriétés d'objets et l'ensemble des propriétés simples. A contrario, OWL Full a l'inconvénient d'avoir un niveau d'expressivité qui le rend indécidable, contrairement à OWL DL et OWL Lite. Il semble donc assez peu utilisé.

2.2.2.10 Synthèse sur le formalisme OWL

OWL se différencie du couple RDF/RDFS en ceci qu'il apporte à RDFS, toute l'inférence basée sur l'hypothèse de correction de données. Si RDF et RDFS apportent à l'utilisateur la capacité de décrire le domaine à modéliser avec des classes et des propriétés, OWL intègre, en plus le support d'équivalences conceptuelles : comparaison des propriétés et des classes (identité, équivalence, contraire, disjonction), caractéristiques logiques des propriétés (symétrie, transitivité, etc), contraintes sur les propriétés (cardinalité, valeurs permises) et équivalence des termes. Ces différents constructeurs permettent de déduire de nouvelles informations (implicites) à partir de l'information préexistante (explicite).

Le formalisme OWL se décline en trois versions d'expressivité croissante OWL Lite, OWL DL et OWL Full (OWL Lite $\subset$ OWL DL $\subset$ OWL Full). Les raisonnements associés à OWL Lite et OWL DL sont décidables mais ne sont, par contre, pas compatibles avec RDFS 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 RDFS. En conséquence, toute ontologie RDFS n'est pas forcément une ontologie OWL Lite ou OWL DL. A l'opposé OWL Full est compatible avec RDFS mais les raisonnements associés ne sont pas forcément décidables. Notons que ces niveaux évoluent avec les nouveaux besoins rencontrés par les utilisateurs. Ainsi, OWL2 ajoute à OWL1 un support plus étendu des types de données permettant ainsi la définition de types de données utilisateurs. Ces extensions sont pour la plupart supportées par la plupart des outils et raisonneurs existants (Protégé 4.0 [8], Fact++, Pellet).

[8] http ://sourceforge/protege.org

3 Similitudes et différences des formalismes d'ontologie

Après l'étude des langages de définitions d'ontologies (RDFS, PLIB et OWL). On remarque que ceux-ci présentent des points en commun et des différences [22].

3.1 Similitudes

Pour ce qui concerne leurs points communs, quatre points méritent d'être soulignés.

1. Ils conceptualisent l'univers à l'aide de classes et propriétés. Les classes sont organisées en hiérarchie au moyen de relations de subsumption.
2. Ils associent chaque concept des ontologies (classes, propriétés, etc.) à un identifiant permettant de le référencer à partir de n'importe quel environnement ou système, et en particulier par une autre ontologie. En PLIB, cet identifiant est unique. Il est appelé BSU (Basic Semantic Unit) et, il est fondé sur l'identification de la source de l'information ontologique. En OWL, cet identifiant est appelé URI (Uniform Ressource Identifier), il est basé sur les identifiants Web, et n'est pas nécessairement unique.
3. Ils définissent des méta-attributs qui permettent de décrire textuellement et dans différentes langues la description des concepts.
4. Les autres constructeurs qu'ils offrent répondent aux besoins d'un problème précis et ont été influencés par des courants différents.

3.2 Différences

Pour ce qui concerne leurs différences, nous constatons les faits suivants :

1. Les formalismes orientés gestion et échange des données (RDFS, PLIB) définissent un langage canonique, ce qui les rend plus adaptés pour définir des bases de données (pas de redondance ou de duplication des données). De plus, les contraintes définies dans l'ontologie sont considérées comme des contraintes d'intégrité et permettront de détecter les erreurs de données.
2. Les formalismes orientés inférence (OWL) sont plus centrés sur des opérateurs de classes qui permettent de définir des classes définies (non primitives). Ces formalismes proposent un ensemble de constructeurs d'équivalence conceptuelle (union, intersection, équivalence, ...) qui peuvent être combinés pour définir de nouvelles classes, puis raisonner sur ces différentes classes. A contrario, ils ne permettent pas de vérifier qu'il n'y a pas d'erreurs dans les données.

Ces distinctions ont amené à proposer une classification des ontologies que nous présentons dans la section ci-dessous.

3.3 Le modèle en oignon

Les formalismes de modélisation d'ontologies permettent, à travers les constructeurs qu'ils offrent, de définir des ontologies. Cependant, ces ontologies ne sont pas identiques. Au sein du

laboratoire LISI[9], l'ensemble des approches de modélisation ontologiques ont été plus largement étudiés, en intégrant les approches linguistiques (dont les concepts sont représentés et identifiés par des termes). Cette étude a amené à proposer une classification des ontologies suivant un modèle en couches appelé *modèle en oignon* [37] (figure 1.3) et constitué de trois couches.

- les *Ontologies Conceptuelles Canoniques* (OCC) contiennent les ontologies qui visent à décrire des concepts et non les mots d'un langage et dont les définitions ne contiennent aucune redondance. Les OCC (Dublincore, IEC-61360-4 (International Electronic Commision)) adoptent une approche de structuration de l'information en termes de classes et de propriétés et leur associent des identifiants uniques réutilisables dans différents langages ;
- les *Ontologies Conceptuelles Non Canoniques* (OCNC) contiennent les ontologies qui décrivent également des concepts mais représentent non seulement des concepts primitifs (canoniques), mais aussi des concepts définis (non canoniques), c'est-à-dire qui peuvent être construits à partir de concept primitifs et/ou d'autres concepts définis, par exemple par les constructeurs d'équivalence définis dans OWL ;
- les *Ontologies Linguistiques* (OL) contiennent les ontologies qui définissent l'ensemble des termes qui apparaissent dans la description langagière d'un domaine. Outre des définitions textuelles, un certain nombre de relations linguistiques (synonyme, hyperonyme, hyponyme, etc.) sont utilisées pour capturer de façon approximative et semi-formelle les relations entre les mots. Un exemple d'OL est WordNet [10].

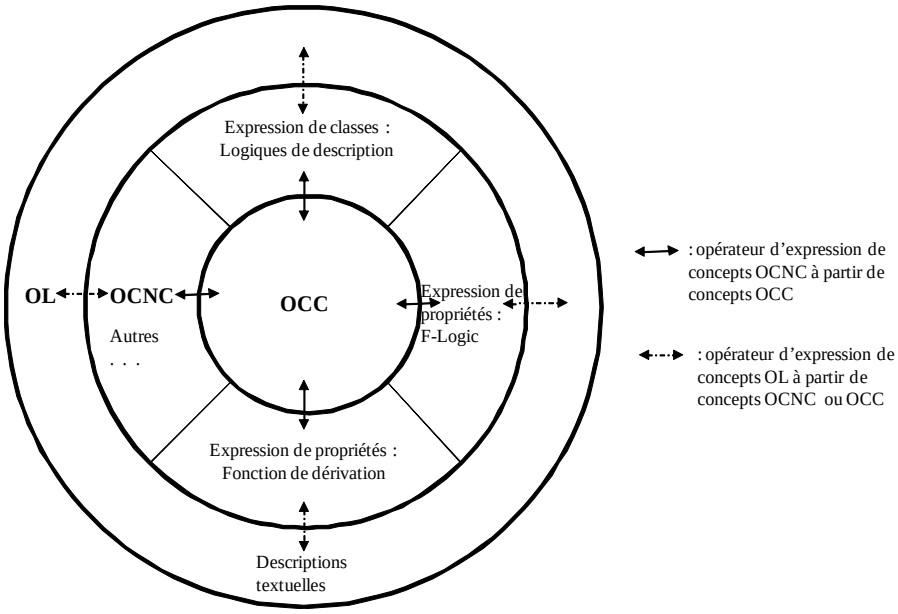


FIG. 1.3 – Le modèle en oignon

Malgré la diversité des formalismes d'ontologies, il apparaît que tous les formalismes possèdent un noyau commun. Nous proposons d'identifier pour chaque couche du modèle en oignon, les constructions offertes par les différents formalismes d'ontologies et, de faire une comparaison de ces constructions en termes de :

[9] http ://www.lisi.ensma.fr
[10] http ://wordnet.princeton.edu/

- ce qu'elles permettent d'exprimer en commun,
- ce qu'elles permettent d'exprimer de manière orthogonale ou complémentaires et enfin,
- ce qu'elles ne permettent pas d'exprimer simultanément et qui nécessitera toujours un choix.

3.3.1 Ontologies Conceptuelles canoniques

Tous les formalismes d'ontologies PLIB, RDFS et OWL offrent des constructeurs de classes et de propriétés canoniques. Ils offrent également des types de données simples (caractères, entier, réel, booléen). Ces différentes constructions permettent de caractériser de manière précise les informations du domaine à modéliser. Ainsi, ces formalismes permettent de définir avec ces constructeurs, des ontologies canoniques. Toutefois, chaque formalisme fournit également un ensemble de constructeurs spécifiques et, imposent des choix de modélisation. Certains de ces constructeurs spécifiques peuvent cohabiter sans contradiction dans le même environnement.

	PLIB	RDFS	OWL
Classes			
subsumption de classes	simple / modularité	multiple	multiple
Propriétés			
typage fort	+	-	± (Lite)
dépendance entre propriété	+	-	-
domaine / co-domaine multiple	-	+	± (Lite)
subsumption de propriétés	-	+	+
fonctionnelle	+	-	+
inverse fonctionnelle	-	-	+
cardinalité	+	-	±
Types de données			
primitifs	+	+	+
classe	+	+	+
collection	+	+	+
Individus			
identification universelle	-	+	+
multi-instanciation	-	+	+
point de vue	+	-	-
méta-modélisation	-	+	± (Full)
détection d'erreurs de données	+	-	-

TAB. 1.1 – Couche canonique des formalismes d'ontologies PLIB, RDFS et OWL

Le tableau 1.1 montre par exemple le formalisme PLIB offre la possibilité de définir le contexte d'évaluation d'une propriété grâce au constructeur de propriété dépendante. OWL permet de définir une relation de subsumption en les propriétés. Ces deux constructions spécifiques peuvent tout à fait être définies simultanément dans un même environnement, car ils sont orthogonaux. A contrario, le formalisme PLIB adopte une approche de représentation par point de vue pour modéliser certains individus. Cette représentation ne peut cependant pas coexister simultanément avec la multi-instanciation offert les formalismes RDFS et OWL, ceci car PLIB ne permet pas de rattacher une instance à plusieurs classes. Il est donc nécessaire, pour cette dernière alternative

de représentation des données, d'établir un choix pour chaque environnement où l'on souhaiterait définir des OCC à partir de constructions issues à la fois des formalismes PLIB, RDFS et OWL. Il en est également de même pour le constructeur de modularité offert par PLIB et la modélisation par héritage multiple disponible dans les formalismes RDFS et OWL.

3.3.2 Ontologies Conceptuelles Non canoniques

En plus des constructeurs canoniques, certains formalismes d'ontologie offrent des constructeurs d'équivalences conceptuelles ou des prédicats constructifs qui enrichissent la couche canonique des ontologies. Le tableau 1.2, montre les différentes constructions non canoniques offertes par les formalismes d'ontologie PLIB et OWL DL permettent par exemple de définir des types de données énumérées. OWL permet de définir une classe comme étant l'intersection de plusieurs autres classes. Cette constructions est interprétée comme la subsumption multiple, elle peut donc être représentée en utilisant l'approche modulaire défini par le formalisme PLIB en important explicitement toutes les propriétés des classes subsumantes. L'intersection de classes peut donc être implémentée dans ces différents formalismes. Les constructions telles que les caractéristiques

	PLIB	RDFS	OWL
Classes			
Intersection	±	-	+
Union	-	-	± (DL/Full)
Complément	-	-	± (DL/Full)
Énumération d'instances	±	-	± (DL/Full)
Restriction			
co-domaine	-	-	+
cardinalité	-	-	+
égalité/inégalité de valeur	-	-	± (DL/Full)
Prédicats constructifs sur les propriétés			
réflexivité, symétrie, transitivité, ...	-	-	+
inverse	-	-	+
Types de données			
Énumérés	+	-	± (DL/full)
Unité	+	-	-
Prédicats constructifs sur la déduction de faits	-	-	+
Prédicats constructifs sur la vérification d'intégrité	+	-	-

TAB. 1.2 – Couche non canonique des formalismes d'ontologies PLIB, RDFS et OWL

de propriétés offerts par le formalisme d'ontologie OWL, peuvent être implantées sur la couche canoniques des ontologies PLIB. Par contre, la définition d'une classe comme étant l'union de deux classes, possible en OWL DL est impossible en PLIB où une instance appartient seulement à une classe. De même, le fait qu'une classe soit une restriction d'une autre classe ne peut pas s'exprimer directement en PLIB, même si elle n'est pas contradictoire avec les primitives de PLIB.

3.3.3 Ontologies Linguistiques

Enfin, afin de permettent l'accès dans une interface langagière aux concepts des ontologies, les différents formalismes d'ontologies offrent la possibilité d'attacher à chaque concept, un ensemble

d'attributs (de descripteurs) prédéfinis et ce, dans différentes langues. Le tableau 1.3 présente les différents descripteurs disponibles dans les formalismes d'ontologies PLIB, RDFS et OWL. PLIB possède un plus grand nombre de descripteurs. Ils sont plus précis que les descripteurs OWL (qui se limite pour l'essentiel à, *label* et *commentaire*). Ils pourraient donc être utilisées pour enrichir plus spécifiquement la description des concepts OWL.

Le label est en général utilisé en OWL pour spécifier un nom préféré, alors que le commentaire est utilisé pour apporter davantage de précision (« définition ») sur un concept. Ainsi, dans tout environnement intégrant à la fois les constructions des formalismes PLIB et OWL par exemple, un choix doit être défini entre label et commentaire OWL et, nom préféré et définition PLIB.

OWL permet de rattacher à un concept une référence précisant des concepts ayant certaines relations avec un concept. Ces références peuvent être d'autres concepts de l'ontologie mais la nature de la relation n'est pas précisée. Il permet également de définir la source émettrice d'un concept. A contrario, les descriptions PLIB sont essentiellement textuelles et ne permettent pas de définir des relations avec d'autres concepts. Toutes les relations sont définies au niveau du formalisme lui-même.

	PLIB	RDFS	OWL
Classes / Propriétés			
commentaire	-	+	+
label	-	+	+
référence	-	-	+
source	-	+	+
définition	+	-	-
note	+	-	-
remarque	+	-	-
nom court	+	-	-
nom préféré	+	-	-
figure	+	-	-
document de définition	+	-	-
Individus			
commentaire	-	+	+
label	-	+	+
description multilingue	+	+	+

TAB. 1.3 – Couche linguistique des formalismes d'ontologies PLIB, RDFS et OWL

Conclusion

Nous avons présenté, dans ce chapitre, les deux principales familles de formalismes d'ontologie : OWL et PLIB. Ces deux formalismes sont différents tant du point de vue de leur domaine cible initial que du point de vue de leur objectif opérationnel.

Du point de vue domaine cible, OWL vise le domaine du langage dans la mesure où il est dérivé des logiques terminologiques (ou logique de description). PLIB, quant à lui, a été développé pour modéliser les informations techniques. Du point de vue opérationnel, PLIB a été développé pour l'échange et l'intégration d'information. Quant à OWL, il a été développé pour permettre de faire

des inférences. Ceci a amené à le subdiviser en différents sous-formalismes selon la décidabilité du raisonnement. Il n'est donc pas étonnant, comme nous l'avons vu dans ce chapitre que les langages résultants soient assez différents.

Un outil d'analyse pour les comparer les formalismes d'ontologie est de distinguer (1) leur partie canonique, qui leur permet de décrire de façon unique et non ambiguë les concepts d'un domaine, (2) leur partie non canonique, qui permet définir des modélisations équivalentes à la modélisation canonique et (3) leur partie linguistique qui permet d'enrichir les modélisations formelles, faites par les modèles précédents, par des informations textuelles destinées à clarifier ce que chaque concept dénote. En utilisant cette classification, nous avons pu distinguer, pour chacun des niveaux, les constructions identiques, les constructions orthogonales, et les constructions incompatibles entre lesquelles il faudrait choisir.

Au niveau central, la construction des instances définies par l'appartenance à une classe ontologique et des valeurs de propriétés ontologiques apparaît universel et appartiendra au modèle central d'interopérabilité. Inversement, le sens logique, les rapports entre prédicats et données devront être choisi entre le point de vue déductif de OWL qui admet qu'un prédicat n'est pas erroné mais permet de déduire des faits et PLIB pour lequel un prédicat est une contrainte d'intégrité dont la validation met en évidence une erreur des données.

Tous les éléments communs devront être représentés dans notre architecture. Concernant les éléments incompatibles, notre objectif étant d'avoir un système PLIB complet, nous ferons toujours le choix de l'approche retenue par PLIB. Par exemple pour exprimer qu'une propriété peut avoir une dépendance et une unité, ou de choisir entre multi-instanciation (OWL) et technique d'agrégat d'instances avec point de vue (PLIB). Concernant les constructions orthogonales, nous précisons au chapitre 3, les constructions que nous avons retenus et celles qui devront pouvoir être réalisées à titre d'extension en exploitant la flexibilité du modèle d'ontologie.

L'objectif de notre travail étant de déboucher sur une nouvelle architecture de base de données pour décrire des données associées à des ontologies, nous présentons, au chapitre suivant, un état de l'art sur les bases de données de ce type existantes.

Chapitre 2

Bases de Données à Base Ontologique

Sommaire

Introduction

Nous avons montré dans le chapitre précédent, les difficultés que soulève la représentation des données associées à des ontologies dans les bases de données. En particulier, le fait que les ontologies permettent de modéliser les domaines au travers non seulement de concepts primitifs, bien adaptés pour la représentation en bases de données, mais aussi au travers de concepts définis qui permettent d'exprimer de plusieurs façons le même concept, introduisant ainsi, à la fois, redondance et nécessité de raisonnement.

Le but de ce chapitre est tout d'abord de présenter comment peuvent être représentées, au sein des bases de données, les ontologies et les données associées. Nous appelons ces données des Données à Base Ontologique (DBO). La représentation des DBO et de leur sémantique définie par l'ontologie associée et le lien entre chaque donnée et l'élément ontologique qui en définit le sens, a nécessité la proposition de nouvelles architectures de bases de données. Celles-ci sont appelées Bases de Données à Base Ontologique (BDBO). Nous discutons en particulier les différentes techniques mises en œuvre pour gérer dans les BDBO, les DBO des classes canoniques et non canoniques. L'objectif est de tirer les conséquences de cette analyse dans la perspective d'offrir une architecture permettant de supporter des ontologies reconcilliant les constructions de plusieurs langages et représentant à la fois, au sein de la bases de données, des informations canoniques et certaines informations non canoniques.

L'organisation de ce chapitre est la suivante. Nous caractérisons dans la section 1 les Données à Base Ontologiques. La section 2 définit la notion de Bases de Données à Base Ontologique. Dans la section 3 nous présentons une analyse des différentes structures de représentation des ontologies dans les BDBO, nous proposons en nous basant sur les schémas utilisés pour la représentation des ontologies, une classification des BDBO. Nous présentons ensuite une analyse des différentes structures de représentation des ontologies et des données au sein des BDBO. La section 4 est consacrée à la gestion des DBO dans les BDBO existantes ; nous y analysons comment sont assurées la montée en charge et les interrogations des DBO. Dans la section 5, nous discutons les besoins actuels des applications pour les BDBO. Nous étudions succinctement, dans la section 6, un ensemble de systèmes de BDBO existants et représentatifs de l'offre actuelle, en les positionnant par rapport aux besoins des applications que nous avons identifiés. Enfin, nous concluons ce chapitre en positionnant les systèmes étudiées par rapport aux objectifs du système de BDBO que nous proposons.

1 Données à Base Ontologique

Une donnée à base ontologique est une donnée qui contient une référence à une classe d'une ontologie qui en définit la sémantique. Les ontologies étant, comme nous l'avons vu au chapitre 1, des formalismes de modélisation de type classe/propriété/instance, une DBO sera donc tout ou partie d'un individu d'une classe d'une ontologie caractérisée par un ensemble de valeurs de propriétés applicables de sa classe. Les références aux classes et propriétés seront faites par les identifiants universels dont disposent tous les formalismes d'ontologies.

Les DBO sont utilisées pour annoter les ressources du Web, en particulier les documents. Elles peuvent alors être semi-automatiquement produites et associées aux classes et propriétés des ontologies par les outils de TALN [11]. Cependant, la diffusion et la manipulation de connaissances se faisant de plus en plus à l'aide d'ontologies, l'utilisation des ontologies (PLIB, OWL), des éditeurs et des outils qui leurs sont dédiés ont dépassé le cadre de quelques domaines particuliers (Web sémantique ou ingénierie). Les DBO sont désormais utilisées dans bien d'autres domaines comme la géologie ou encore la médecine.

Initialement, les DBO étaient gérées en mémoire centrale par les systèmes informatiques comme les moteurs d'inférence. Cependant, au cours des dix dernières années, elles ont atteint des tailles qui rendaient leur gestion difficile, voire incompatible avec le traitement en mémoire centrale. De ce fait, pour satisfaire les besoins de performance requis pour de nombreuses applications, des efforts ont été fournis ces dernières années par de nombreux chercheurs pour proposer une structure permettant leur représentation dans des Systèmes de Gestion de Bases de Données (SGBD) afin de profiter des performances de ces dernières.

2 Définition d'une BDBO

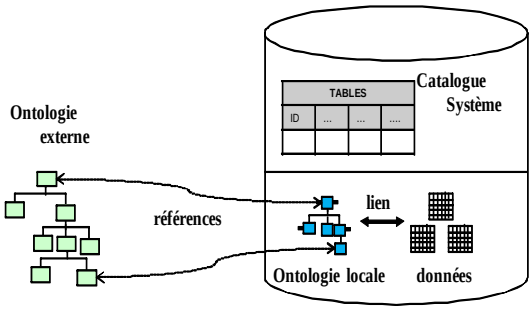


FIG. 2.1 – Architecture générale d'une BDBO

Nous appelons Base de Données à Base Ontologique, une base de données qui possède les trois caractéristiques suivantes :

1. Ontologie et données sont toutes deux représentées dans la même base de données et peuvent faire l'objet des mêmes traitements (insertion, mise à jour, requêtes, etc.).

[11] Traitement Automatique de la Langue Naturelle

2. Toute donnée est associée à un élément ontologique qui en définit le sens et inversement, tout élément ontologique permet d'accéder aux données qui lui correspondent.
3. L'ontologie locale à la base de données possède éventuellement des références vers des ontologies externes.

La figure 2.1 représente synthétiquement l'architecture d'une BDBO.

Avec l'accroissement du volume des DBO, différentes architectures de BDBO ont été proposées : OntoMS [53], Sesame [11], DLDB [51], KAON [9] OntoDB [20], RDFSuite [4], Jena [68], Oracle [13], SOR [44], etc. Ces BDBO permettent de stocker les différents concepts qui composent une ontologie. En particulier, les BDBO stockent les données de niveau ontologie (classes et propriétés) qui contiennent la sémantique des données et les données de niveau instance que nous appelons DBO. Les différentes BDBO se distinguent par :

- les formalismes d'ontologies supportés et qui définissent en particulier, les classes et propriétés (canoniques et non canoniques) que le système peut représenter ;
- le schéma adopté pour la représentation des ontologies ;
- le schéma adopté pour la représentation des DBO ;
- les mécanismes mis en œuvre pour faciliter l'interrogation des DBO .

La problématique de la représentation des DBO dans les bases de données est donc de définir le schéma des tables pour le stockage des données de chacun des niveaux (ontologie et données), et d'assurer la gestion et l'interrogation à la fois des ontologies et des données. Nous présentons dans la section ci-dessous les différentes approches qui ont été suivies pour la représentation des ontologies et des données dans les architectures de BDBO.

3 Architecture des BDBO

Plusieurs chercheurs ont étudié la possibilité d'intégrer à la fois les ontologies et les DBO dans les bases de données relationnelles ou relationnelles/objets. Plus récemment, les grands leaders commerciaux de SGBD tels que Oracle [13] et IBM [44] ont également proposé des supports pour les BDBO. Pour illustrer les différentes approches de stockage proposées par les BDBO, nous utilisons l'ontologie présentée sur la figure 1.2 (cf. section 2.1.1.2 du chapitre 1). Rappelons que cette ontologie définit, entre autres, les classes *Personne*, *Étudiant*, *Document* et *Territoire*.

Le support des DBO et de leur sémantique portée par les classes et propriétés définies dans les ontologies dans un même référentiel pose deux problèmes :

1. celui de la représentation des données à base ontologique et,
2. celui de la représentation des ontologies.

les DBO sont des données contenant une référence à un concept d'une ontologie ; de nombreuses solutions de représentation des données, en particulier les solutions adoptées par les bases de données traditionnelles existent pour la représentation des données. Le problème de la représentation des DBO et des ontologies sont deux problèmes relativement indépendants et, ont été traités séparément par les chercheurs. Nous présentons d'abord dans la section 3.1, les approches qui ont été proposées pour la représentation des ontologies dans les bases de données. Ensuite, nous présentons dans la section 3.2 les approches suivies pour la représentation des DBO.

3.1 Représentation des ontologies dans les BDBO

Trois approches de représentation ont été suivies pour la représentation des ontologies dans les BDBO. Ces trois approches se distinguent à la structure de représentation adoptée.

- La première approche, qui est également la plus simple, utilise une structure qui correspond à la structure d'une déclaration (sujet, prédicat objet) suivant le formalisme RDF. Dans cette représentation, les concepts (classes et propriétés) de l'ontologie ainsi que les DBO étant décrits suivant le formalisme RDF avec la même structure, un seul schéma peut être utilisée pour représenter à la fois les concepts de l'ontologie et les DBO.
- La seconde approche, proposée principalement pour les ontologies décrites suivant les formalismes d'ontologies PLIB, RDFS et OWL, définit des tables spécifiques pour chaque constructeur offert par le formalisme d'ontologie supporté. Ainsi, pour le formalisme d'ontologie RDFS par exemple, des tables spécifiques sont définies pour les constructeurs de classes et de propriétés.
- La troisième approche, basée sur l'approche MDE [12], étend la seconde en introduisant dans la base de données, le méta-modèle du formalisme d'ontologie représenté.

Cette distinction permet de proposer une classification des BDBO en fonction de la structure du schéma qu'il adopte pour la représentation des ontologies. Nous distinguons ainsi :

1. L'architecture de type 1 où un seul schéma est utilisé pour stocker les classes et propriétés. Ce schéma est en particulier utilisé lorsque ontologie et DBO sont décrites suivant le formalisme RDF.
2. L'architecture de type 2, où deux schémas distincts sont utilisés. Un spécifique pour les classes et propriétés de l'ontologie et l'autre spécifique pour les DBO . Cette architecture est adaptée lorsque les ontologies sont décrites suivant les formalisme d'ontologie PLIB, RDFS, OWL, etc. Les constructions disponibles dans ce formalisme vont donc être utilisées pour définir le schéma de représentation des ontologies.
3. L'architecture de type 3. Cette dernière architecture, étend l'architecture de type 2 en définissant un schéma supplémentaire appelé méta-schéma. Ce dernier est en particulier utilisé pour permettre l'évolution du schéma de représentation des ontologies.

3.1.1 BDBO de type 1 : une table pour l'ontologie

Cette première approche de représentation des ontologies dans les BDBO est également la plus simple. Dans les BDBO de type 1, toutes les informations sur l'ontologie sont représentées en utilisant un seul schéma composé d'une unique table de triplets à trois colonnes *(subject, predicate, object)* [28, 68, 13, 55]. Cette table est qualifiée de *table verticale* [3]. L'ensemble des informations de l'ontologie est décomposé en forme de triplets ce qui correspond exactement à la structure de RDF . Cette représentation est donc générique par rapport au formalisme d'ontologie. Cette représentation est également utilisée par les systèmes dits RDF natifs directement construits sur des systèmes de fichiers pour gérer les données RDF indépendamment de toute base de données tels que OWLIM [38], HSTAR [12] ou YARS [29]. Les trois colonnes de cette table représentent respectivement l'identifiant d'un élément d'ontologie, un prédicat et la valeur du prédicat qui peut

[12] Model Driven Engineering

être soit un identifiant d'un élément d'ontologie, soit une valeur littérale. La figure 2.2 présente un extrait de la table verticale correspondant aux classes de l'ontologie de la figure 1.2. Par exemple, le triplet[13] *(Etudiant, subClassOf, Personne)* représente une relation de subsumption entre les classes *Etudiant* et *Personne*. Notons que les données peuvent également être représentées de la même façon ; nous avons donc également représenté sur cette table les DBO en faisant l'hypothèse que la table verticale est aussi choisie pour représenter les données.

TRIPLES		
Subject	**Predicate**	**Object**
Personne	***type***	***Class***
Etudiant	***type***	***Class***
Etudiant	***subClassOf***	Personne
nomPrénom	***type***	***Property***
nomPrénom	***range***	***String***
…	…	…
etu#01	***type***	Etudiant
etu#01	nomPrénom	Philippe
etu#01	aEcrit	doc#01
doc#01	estGéolocaliséPar	ter#01
…	…	…

FIG. 2.2 – Approche (générique) de représentation des ontologies des BDBO de type 1

3.1.2 BDBO de type 2 : un schéma spécifique pour représenter l'ontologie

Lorsque les données deviennent de taille importante, le fait de ranger toutes les informations dans une unique table, impose de réaliser de nombreuses auto-jointures sur une table de grande taille, ce qui se traduit par un coût élevé des interrogations en particulier lorsque les ontologies et les DBO sont représentées dans la même table de triples. Les BDBO de type 2 offrent deux schémas de représentation spécifiques : l'un pour les DBO, l'autre pour les ontologies. Ce dernier étant spécifique du formalisme d'ontologie supporté. Cette représentation est adoptée en particulier par les BDBO RDFSuite [4] et SOR [44], qui définissent une table pour chacun des concepts (classes, propriétés) et axiomes (subsumption de classes) du formalisme d'ontologie cible (par exemple, RDFS, OWL ou PLIB). La figure 2.3 présente un exemple de BDBO de type 2 stockant les données de l'exemple précédent (voir figure 1.2). Dans cet exemple, les données de niveau ontologie sont stockées en utilisant un schéma pour des ontologies RDFS .
Notons que dans l'approche de représentation spécifique, le schéma de représentation du formalisme d'ontologie est figé et ne peut donc s'adapter à des évolutions de ce dernier.

3.1.3 BDBO de type 3 : approche spécifique avec méta-schéma

Les BDBO de type 3 proposent, en plus de la représentation spécifique des ontologies définies suivant le formalisme d'ontologie supporté comme dans les BDBO de type 2, l'ajout d'un autre

[13] RDF utilise des URI comme identifiants. Pour des raisons de lisibilité nous utilisons des noms. Les uri des constructeurs du formalisme d'ontologie sont représentées en italique.

Class	
ID	Name
c1	Personne
c2	Etudiant
c3	Document
c4	Territoire

SubClassOf	
Sub	Sup
c2	c1

Property	
ID	Name
p1	nomPrénom
p2	aEcrit
p3	estGéolocaliséPar
...	...

Domain	
prop	class
p1	c1
p2	c1
p3	c3
...	...

Range	
prop	type
p1	xsd:string
p2	xsd:string
p3	c4
...	...

FIG. 2.3 – Approche (spécifique) de représentation des ontologies des BDBO de type 2

schéma appelé le méta-schéma. Le méta-schéma permet de définir le formalisme d'ontologie supporté. Le *méta-schéma* d'ONTODB [20, 59], par exemple, stocke les constructions du formalisme d'ontologie dans un méta-modèle réflexif. Ce méta-schéma joue pour le schéma des ontologies, le même rôle que celui joué par la méta-base (ou catalogue système) pour les bases de données traditionnelles. En effet, le méta-schéma peut permettre : (1) un accès générique aux ontologies, (2) l'évolution du formalisme d'ontologie utilisé, et (3) le stockage de différents formalisme d'ontologie (OWL, DAML+OIL, PLIB, etc.). Certains formalismes d'ontologies tels que PLIB et OWL subissent des évolutions afin d'intégrer de nouvelles spécifications correspondantes aux nouveaux besoins des utilisateurs. La possibilité d'évolution du formalisme d'ontologie au sein de la BDBO est une fonctionnalité qui serait utile pour les BDBO basées sur ces formalismes d'ontologies. Ainsi ces BDBO intégreraient aisément grâce au méta-schéma les évolutions du formalisme d'ontologie. La figure 2.4 présente le méta-schéma de notre exemple.

Meta-Schema

Entity		
ID	Name	SuperEntity
Entity#1	Resource	
Entity#2	Class	Entity#1
Entity#3	Property	Entity#1

Attribute			
ID	Name	Domain	Range
Att#1	name	Entity#1	Type#1
Att#2	domain	Entity#3	Type#2
...	...	...	...

Type	
ID	Name
Type#1	String
Type#2	Entity#2

FIG. 2.4 – Méta-Schéma des BDBO de type 3

3.1.4 Synthèse sur la représentation des ontologies dans les BDBO

La plupart des BDBO actuelles proposent une représentation spécifique (liée aux constructeurs offerts par le formalisme d'ontologie supporté). Leurs évaluations de performances par rapport à un ensemble de requêtes typiques se sont en effet montrées meilleures que la représentation générique [65, 45, 4]. De plus, la BDBO OntoDB (que nous présentons à la section 6.3) propose une structure pour stocker et faire évoluer le formalisme d'ontologie utilisé. La représentation de cette partie, nommée méta-schéma, permet d'adapter ce système aux évolutions du formalisme d'ontologie PLIB. Nous montrons dans le chapitre 3 que cette approche permet également,

lorsque le besoin s'en fait sentir, d'étendre le formalisme d'ontologie utilisé par des constructeurs provenant d'autres formalismes d'ontologies.

Nous venons de voir que la majorité des BDBO utilise un schéma de base de données traditionnel pour stocker les ontologies. La table 2.1 résume les approches de représentation des ontologies adoptées par les différentes architectures de BDBO.

	Type 1	Type 2	Type 3
Nombre de schémas	1	2	3
Séparation ontologie/données	non	oui	oui
Évolution formalisme d'ontologie	non [1]	non	oui
Approche de représentation des ontologies	générique	spécifique	spécifique

TAB. 2.1 – Approche de représentation des ontologies dans les BDBO

Dans la section suivante, nous étudions les propositions faites pour stocker les DBO.

3.2 Représentation des DBO dans les BDBO

Comme pour la représentation des ontologies, trois principales approches ont été suivies pour la représentation des DBO dans les BDBO. La première approche dite verticale, est adaptée en particulier lorsque les DBO sont décrites suivant le formalisme RDF. Cette approche est souvent utilisée lorsque l'architecture est de type 1. La seconde approche, dite binaire, revient à décomposer l'ensemble des relations sous forme de relation unaire (pour l'appartenance aux classes) et binaire (pour les valeurs des propriétés). Cette approche, permet en particulier, comme l'approche verticale, de supporter la multi-instanciation offerte par les formalismes d'ontologie tels que RDFS et OWL. La troisième approche, dite horizontale, consiste à associer à chaque classe de l'ontologie, une table définie suivant la représentation traditionnelle utilisée dans les bases de données relationnelles. Ces différentes représentation des DBO sont plus ou moins adaptées suivant les besoins des applications à mettre œuvre. Nous décrivons ci-dessous ces différentes approches de représentation des DBO dans les BDBO. L'objectif est de faire ressortir, pour chacune d'elles, les hypothèses induites, et les cas d'utilisations auxquels elles sont adaptées.

3.2.1 Approche verticale

Cette approche de représentation est principalement utilisée par les BDBO de type 1. L'unique table de triples à trois colonnes pouvant également être utilisée pour stocker les DBO. Pour les données de niveau instance, les trois colonnes de cette table représentent respectivement l'identifiant d'une instance, une caractéristique d'une instance (par exemple, une propriété ou l'appartenance à une classe) et la valeur de cette caractéristique. Par exemple (cf. figure 2.5-a), les triplets *(etu#o1, rdf :type, Etudiant)* et *(etu#01, aEcrit, doc#01)* représentent le fait que *etu#01* est un étudiant et qu'il a écrit le document *doc#01*. L'insertion de triples dans cette approche est une opération simple mais, associée à un formalisme plus complexe (OWL par exemple), la mise à

[1] on peut modifier le formalisme d'ontologie, mais il n'est pas possible d'anticiper les modifications de façon générique. Toute modification pour être prise en compte nécessite une modification des programmes d'accès.

jour d'une information ne correspond pas toujours à la modification d'un seul triplet, mais peut nécessiter la modification d'un ensemble de triplets. Cette approche nécessite également, pour accéder aux instances d'une classe et aux valeurs des propriétés qui les caractérisent, de réaliser de nombreuses auto-jointures sur la table de triples. Cette approche de représentation est par exemple utilisée par RStar[45] et Sesame [11].

Une variante de cette approche, que nous appelons *approche verticale avec* ID, est utilisée par les système Oracle et SOR. Cette variante consiste à stocker respectivement l'identifiant (URI) des ressources et toutes les valeurs littérales dans des tables spécifiques : *ressources* et *littéral* (cf. figure 2.5-b). La table *ressource* est constituée de deux colonnes : la colonne *id* de type entier, et la colonne URI qui permet de représenter l'URI des ressources. La table *littéral* est également composée de deux colonnes : la colonne *id* de type *entier* et la colonne *value* pour les valeurs littérales. Les identifiants des ressources et des littéraux dans la colonne *id* de ces deux tables sont référencés dans la table *triples*.

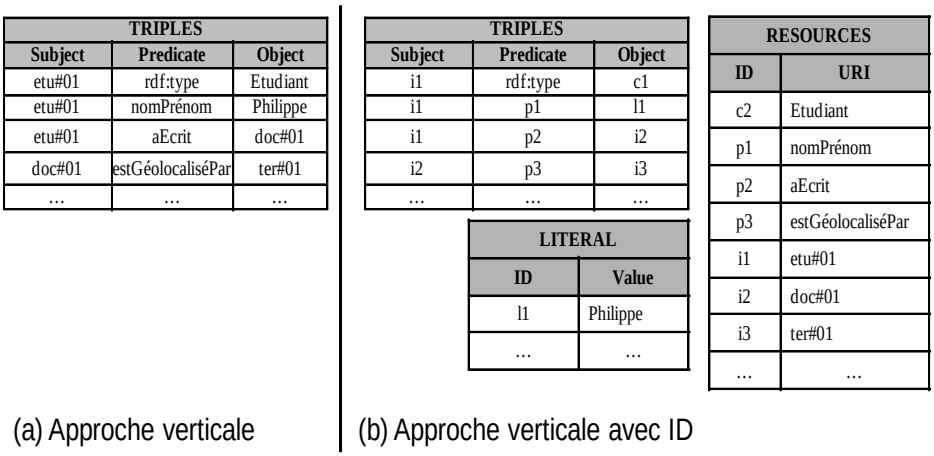


FIG. 2.5 – Schéma des instances suivant l'approche verticale

3.2.2 Approche binaire

Dans cette approche, les classes auxquelles appartient une DBO et leurs propriétés sont stockées dans des tables de structures différentes. L'approche binaire donne lieu à trois variantes pour la partie classification en fonction de la stratégie utilisée pour la gestion de l'héritage et du polymorphisme :

- une unique table pour toutes les classes de l'ontologie,
- une table par classe avec héritage de table (ISA),
- une table par classe sans héritage de table (NOISA)

1. La première variante [45], consiste à créer une seule table pour stocker toutes les instances des classes (au lieu d'une par classe). La table est constituée de deux colonnes (*uri*, *class*ID) : la colonne *uri* permet de représenter l'identifiant des instances et la colonne *class*ID, l'identifiant de la classe stockée dans l'ontologie (cf. figure 2.6-a).

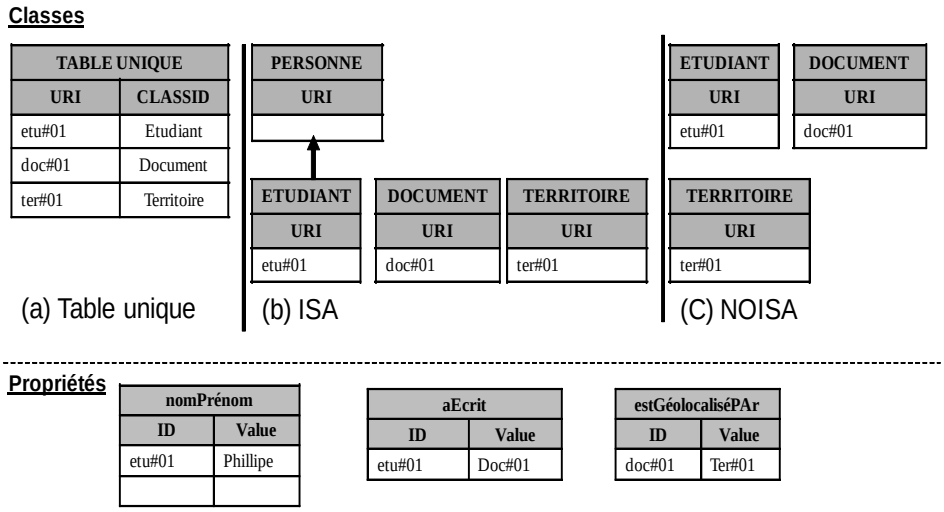


FIG. 2.6 – Schéma des instances suivant l'approche binaire

2. La seconde variante dite ISA dans [65] (cf. figure 2.6-b), consiste à associer à chaque classe et propriété stockées dans la partie ontologie une table spécifique permettant de stocker leurs instances respectives. Une table d'une propriété P_i (respectivement d'une classe C_i) héritera éventuellement de la classe de sa super-propriété (respectivement de sa super-classe) si le SGBD utilisé est relationnel objet (SGBDRO) comme PostgreSQL, Oracle, etc. Chaque table d'une classe C_i est unaire. Elle est constituée d'une colonne *id* qui servira à stocker l'identifiant des instances de la classe. Les tables de propriétés sont binaires. Elles sont constituées de deux colonnes : *id* pour l'identifiant de la propriété et *value* pour la valeur des propriétés. Le domaine de la colonne *value* correspond au co-domaine de la propriété décrite dans la partie ontologie. Cette approche a l'avantage de profiter des potentialités de SQL99 lorsqu'il s'agit de faire des requêtes hiérarchiques (polymorphes). Mais malheureusement, cette solution souffre de la complexité de la mise à jour de la structure des tables lorsque des classes ou propriétés de l'ontologie sont modifiées. Par exemple l'insertion d'une classe entre deux classes demande de supprimer des tables puis de les remettre en relation. Il est également nécessaire pour accéder aux instances d'une classe et à leurs valeurs propriétés, de réaliser des jointures entre la table unaire de la classe et l'ensemble des tables binaires des propriétés qui la caractérisent.

3. La troisième variante, appelé NOISA [65], consiste à ne pas utiliser l'héritage de tables offert par les SGBDRO. Les tables des propriétés et des classes sont définies séparément sans être mises en relation (cf. figure 2.6-c). La réalisation d'une requête polymorphe, nécessite d'accéder à l'ontologie pour récupérer toutes les classes et/ou propriétés impliquées dans la requête. Cette approche présente de mauvaises performances si les requêtes polymorphes sont exécutées sur des arbres très profonds. Les opérations de calcul de sous-classes ou sous-propriétés sont des opérations récursives et coûteuses en temps.

3.2.3 Approche horizontale

Récemment, une approche appelée *table par classe* a été proposée [20, 53]. Elle consiste à associer à chaque classe une table d'instances ayant une colonne pour chaque propriété associée à une valeur pour au moins une instance de la classe. Si le SGBD support le permet, l'héritage de table peut être mis en œuvre. La figure 2.7 présente un exemple de représentation des données de niveau instance suivant cette approche.

Contrairement à l'approche verticale et à l'approche binaire, l'approche horizontale permet

Etudiant		
ID	nomPrénom	aEcrit
etu#01	Phillipe	doc#01
...	...	...

Document	
ID	titre
doc#01	Rapport des zones sismologiques
...	...

Territoire	
ID	nom
ter#01	castellane
...	...

FIG. 2.7 – Schéma des instances suivant l'approche horizontale

d'avoir un schéma pour les DBO similaire aux schémas utilisés par les bases de données traditionnelles. Ce schéma permet d'éviter toutes les jointures nécessaires dans les approches verticales et binaires pour accéder à la description complète d'une instance. Toutefois, la représentation horizontale suppose de respecter certaines hypothèses fondamentales de la théorie des bases de données ; en particulier, le typage fort des instances et des propriétés. Le typage fort des propriétés suppose que chaque propriété possède un unique domaine et un unique co-domaine. Cette hypothèse n'est par exemple pas respectée dans les formalismes RDFS, OWL DL et OWL Full. Par contre elle est respectée par les formalismes PLIB et OWL Lite. Le typage fort des instances suppose que chaque instance appartient à une (hiérarchie de) classe(s). Ceci n'est par imposé par OWL qui permet à la fois qu'une instance n'appartienne à aucune classe explicite (elle appartient alors à la classe universelle owl :Thing), ou qu'une instance appartienne à l'extension de plusieurs classes qui n'ont pas été explicitement définies comme disjointes. Ce n'est pas le cas dans l'approche horizontale (identique à l'approche traditionnelle de représentation en bases de données), car une classe est considérée comme un prototype pour la définition ses instances. Une instance n'appartenant qu'à une seule classe donnée, toute intersection de classes sera vide.

3.2.4 Synthèse sur la représentation des DBO

Dans les BDBO, les individus sont :

- soit porteurs de leur propre structure dans le cas de représentation sous forme de triples ou sous forme binaire,
- soit structurées avec un schéma de données au sens usuel des bases de données, ce qui permet de traiter de façon plus rapide et sans jointure ou auto-jointure de très gros volumes de données.

La représentation binaire et la représentation horizontale présentent dans la plupart des cas des performances meilleures que la représentation verticale. Cependant, des tests réalisés sur un ensemble de requêtes montrent que les performances des BDBO sont liées aux types de requêtes

que visent à répondre les applications [20]. Ainsi, les différentes architectures de BDBO se comportent différemment selon le type d'information qui doit être géré.

- La représentation horizontale est plus appropriée lorsque les DBO sont décrites par un grand nombre de propriétés communes et, les expérimentations [20] ont montrés que dans ce cas les requêtes portant sur un nombre assez grand de propriétés (à partir de six) présentent de meilleures performances que l'approche binaire.
- L'approche binaire est adaptée lorsque les différentes instances d'une classe sont décrites par des propriétés différentes. Elle permet d'obtenir un coût de stockage réduit par rapport à l'approche horizontale qui présenterait dans ce cas de nombreux champs nuls. Le coût des mises à jour est plus élevé que dans l'approche horizontale car les instances étant représentées de manière éclatée, leur mise à jour nécessite souvent des traitements sur plusieurs tables. Notons cependant que le coût des mises à jour reste toutefois plus élevé dans la représentation verticale [65].

Dans les BDBO, la liaison entre les ontologies et les DBO permet de réaliser différents raisonnement sur les données. Ces raisonnements sont toutefois limités par le formalisme d'ontologie sur lequel se fonde la BDBO. Dans la prochaine section, nous discutons de capacités des BDBO pour résoudre les problèmes de montée en charge et de raisonnement.

4 Montée en charge et raisonnement dans les BDBO

L'idée de représenter au sein des bases de données les ontologies et les données associées, avait pour objectif premier d'exploiter les capacités des bases de données pour offrir une solution au problème de montée en charge des DBO. Par ailleurs, divers raisonnements peuvent être réalisés sur les ontologies. Avant l'émergence des BDBO, ces raisonnement étaient réalisés en mémoire centrale. Un second objectif des BDBO a donc été de pouvoir offrir des solutions pour assurer la réalisation de certains de ces mêmes raisonnements. Nous présentons dans cette section, les solutions proposées pour atteindre ces deux objectifs.

4.1 Montée en charge des BDBO

Comme nous l'avons vu, différents schémas de représentation ont été adoptés par les différentes architectures de BDBO. Nous étudions dans cette section les capacités de chaque architecture de BDBO pour assurer la montée en charge des données.

4.1.1 BDBO de type 1.

Dans les BDBO de type1, l'approche de représentation utilisant une table verticale ou table de triples présente des problèmes de performance lorsque des requêtes requièrent de nombreuses auto-jointures sur cette table [4]. Pour obtenir de bonnes performances dans le traitement des requêtes, chaque colonne de la table verticale doit être indexée [45]. De plus, la colonne qui correspond au prédicat doit être « clustérisée », c'est à dire triée [3] ou des vues matérialisées doivent être créées [51]. Dans les deux cas, cette approche entraîne un coût de stockage supplémentaire (dû aux structures redondantes d'optimisation) et un surcoût très important de traitement des

opérations de mises à jour (dû à la nécessité de « re-clusteriser ») la table. Même avec ces optimisations, plusieurs travaux ont montré que dans de multiple cas, les BDBO de type 2 surpassent les BDBO de type 1 [65, 45, 4].

4.1.2 BDBO de type 2.

Les performances de ces BDBO dépendent de la représentation utilisée pour les données de niveau instance.

Les évaluations de performance conduites dans [3, 65, 51, 1] ont montré que l'approche utilisant une table verticale pour les instances souffre des mêmes faiblesses que celles évoquées précédemment pour les BDBO de type 1. Ainsi, la structure de type 2 associée à une représentation binaire des instances a été considérée pendant longtemps comme la meilleure représentation pour les instances.

Les résultats expérimentaux obtenus sur la représentation horizontale (table par classe) ont remis en cause cette idée [20, 53]. Pour les requêtes où les classes à interroger sont spécifiées, la représentation table par classe (horizontale) surpasse l'approche binaire par un ratio souvent supérieur à 10, en particulier lorsque les instances sont associées à plusieurs propriétés [20]. De plus, les insertions et mises à jour sont plus rapides. Le seul cas où la représentation binaire est meilleure que la représentation table par classe correspond aux requêtes où la classe à interroger n'est pas spécifiée et qui ne concernent qu'un nombre faible de propriétés.

4.1.3 BDBO de type 3.

L'ajout du méta-schéma dans les BDBO de type 3 n'améliore pas les performances des requêtes mais offre de nouvelles fonctionnalités comme nous l'avons précisé précédemment. La disponibilité d'un méta-schéma d'ontologies dans la base de données facilitera en particulier l'extension du formalisme d'ontologie supporté.

4.1.4 Synthèse sur la montée en charge des BDBO

Les BDBO exploitent la puissance et la maturité des bases de données pour assurer la montée en charge des DBO à gérer par les applications. Cette étude sur la scalabilité des BDBO montre que, pour un nombre de cas d'utilisation correspondant aux différents benchmarks réalisés dans la littérature, les BDBO de type 2 ou 3 utilisant soit une représentation binaire, soit une représentation table par classe passent relativement bien à l'échelle et permettent la gestion de données de taille réelle. Certaines BDBO de type 1 [13] offrent des mécanismes nécessaires pour matérialiser des vues pouvant correspondre, en particulier, à une structure de type 2, ce qui leur permet, sous réserve d'optimisation utilisant des vues et des index par l'administrateur, d'avoir l'efficacité des BDBO de type 2. L'autre défi des BDBO est de fournir en même temps des capacités de raisonnement.

4.2 Capacités de Déduction des BDBO

En dehors des bases de données déductives, peu utilisées dans les applications pratiques, les capacités de déduction ne sont pas le principal atout des bases de données. Indépendamment des

différents schémas de représentation des données adoptés, diverses approches ont été proposées pour faciliter l'interrogation des ontologies et des DBO .

Comme nous l'avons vu à dans le chapitre précédent, les ontologies sont constituées de trois couches :

1. une couche canonique qui permet une description unique et non ambigüe des concepts,
2. une couche non canonique qui, au travers de constructeurs d'équivalence conceptuelle, offre la possibilité d'utiliser différentes manières de représenter et de structurer la même information, permettant ainsi de raisonner sur les données, et
3. une couche linguistique qui permet d'offrir une interface d'accès langagière multilingue aux données.

Ainsi, les DBO peuvent soit être décrites en termes de concepts canoniques (ou primitifs), soit en termes de concepts non canoniques (ou définis). Les BDBO doivent donc offrir des solutions pour interroger ces différentes catégories de DBO.

Afin de répondre aux interrogations impliquant des données non canoniques, trois solutions ont été utilisées par les systèmes existants :

- le raisonnement pendant la requête : pour cela, les BDBO sont couplées à des raisonneurs internes ou externes pour réaliser les inférences nécessaires lors du traitement des requêtes ;
- le raisonnement par saturation : dans ce cas, tous les nouveaux faits qui peuvent être inférés sont calculés et matérialisés dans la BDBO lors du chargement des données ;
- l'absence de raisonnement : dans ce cas, les DBO non canoniques ne sont pas gérées dans les BDBO.

4.2.1 Raisonnement pendant la requête

Cette approche a été adoptée par un grand nombre de BDBO (SESAME, RDFSuite, Jena2). Elle consiste à réaliser le raisonnement pendant le traitement des requêtes en produisant des faits déduits utilisés pour fournir le résultat des requêtes. Cette approche entraîne un surcoût pour le traitement des requêtes liées aux inférences réalisées par les raisonneurs auquel s'ajoute le coût de transfert des données lorsque le raisonneur est externe à la BDBO. Cette approche n'impose toutefois pas de coût de stockage et de mises à jour supplémentaire. Les raisonneurs permettent de réaliser en particulier des inférences prédéfinies telles que la subsumption et le test l'instanciation. Ils supportent également, tout ou partie des inférences possibles sur les axiomes définies par le formalisme d'ontologie supporté (RDFS, OWL).

4.2.2 Raisonnement par saturation

Cette approche consiste à réaliser le raisonnement *avant* le traitement des requêtes et à matérialiser tous les faits déduits et, en particulier, la fermeture transitive de toutes les relations transitives. Les BDBO adoptant cette approche (SOR et Oracle par exemple) permettent le traitement efficace de requêtes puisque aucun raisonnement n'est réalisé à l'exécution des requêtes. Son inconvénient est (1) de coûter cher pour le chargement de données volumineuses et (2) d'entraîner la redondance des données dans la BDBO. De plus, (3) certaines opérations comme la suppression des données sont difficiles, voire impossible à réaliser. Les systèmes actuels, bien que permettant

de distinguer les données inférées des données initiales, ne permettent pas de connaître l'origine d'une relation après fermeture transitive. De ce fait, lorsqu'une instance de relation transitive est supprimée, il n'est pas possible de connaître les faits inférés à partir d'elle. La saturation doit donc être entièrement ré-effectuée pour toute modification nécessitant de supprimer des relations.

4.2.3 Absence de raisonnement

Dans cette dernière approche, seules les DBO canoniques sont manipulées par les BDBO. Aucun support n'est alors fourni pour les constructeurs d'équivalences conceptuelles de classes et de propriétés. OntoDB par exemple, fondé sur le formalisme d'ontologie PLIB, adopte cette approche. Cette dernière approche est efficace car aucun traitement (inférence) n'est nécessaire et, il n'y a pas de coût supplémentaire de stockage car les DBO sont uniquement canonique. Cependant, elle limite l'expressivité des ontologies et les interrogations qui peuvent être réalisées sur les DBO.

4.2.4 Discussion sur le raisonnement dans les BDBO

Actuellement, les BDBO supportent principalement les raisonnements sur la subsumption (c'est à dire, exploitant les relations *subClassOf* et *typeOf*). La plupart des BDBO réalisent les raisonnements avant le traitement des requêtes en utilisant différents mécanismes tels que les vues [51], les techniques d'étiquetage [53] ou l'héritage de tables des bases de données relationnelles-objets [4, 11].

Certaines BDBO permettent des raisonnements plus complexes. Par exemple, ONTOMS supportent des raisonnements sur les instances utilisant les propriétés inverses, symétriques et transitives [53]. Mais, ces raisonnements sont en général réalisés sur des fermetures transitives ce qui provoque un surcoût de stockage et de mise à jour important dans des applications de taille réelle. Le système SOR permet de répondre à tous les raisonnements sur les ontologies définies suivant le formalisme d'ontologie OWL DL en saturant complètement toutes les tables au chargement des ontologies, ce qui engendre à la fois un coût de chargement, et un coût supplémentaire de stockage, et surtout un coût important de mise à jour.

D'autres approches proposent d'utiliser des moteurs de règles des bases de données déductives (par exemple, un moteur de règles pour DATALOG) ou des raisonneurs OWL pour réaliser des tâches de raisonnement plus complexes [47, 67, 8, 51]. Cependant, les bases de données déductives n'ont pas été largement adoptées en dehors du domaine académique et le temps de réponse de ce type d'architecture est souvent incompatible avec le besoin d'interaction homme-machine. Enfin, Oracle offre [13, 69] parallèlement à son implémentation de triplets RDF/RDFS, la possibilité de définir en plus des bases de règles prédéfinies pour RDFS et OWL, des bases de règles utilisateurs définissant implicitement de nouveaux triplets. Ces bases de règles peuvent être traitées, au choix de l'administrateur, soit a priori, en représentant dans une vue matérialisée le résultat de l'exploitation des règles d'un certain modèle (ici, un ensemble de triplets) en chaînage avant, soit a posteriori, lors de la requête.

4.2.5 Ontologies gérées

Comme l'a montrée l'étude des architectures de BDBO, les systèmes actuels ne permettent que la gestion d'ontologies définies à partir d'un formalisme d'ontologie spécifique. Pour les données, elles sont gérées soit par saturation (coûteuse du fait des mises à jour), soit en réalisant des raisonnements (également coûteux) lors des interrogations des données canoniques ou non. Ces différentes approches sont actuellement implantées dans différents systèmes de BDBO (Sesame, Jena, RDFSuite, OntoDB, Oracle, SOR, DLDB, ...). Cependant, peu de ces systèmes se sont intéressés à offrir la possibilité de supporter simultanément les ontologies issues d'autres formalismes d'ontologie. De même ces systèmes ne se sont pas intéressés à la gestion des données canoniques sans inférence et sans saturation comme c'est le cas dans les bases de données traditionnelles. Enfin, les systèmes actuels exploitent très peu (voir pas du tout) les capacités des bases de données pour réaliser des raisonnements sur les ontologies et les DBO associées. Nous montrons ceci dans la section 5 en identifiant les besoins actuels des applications et dans la section 6 où nous étudions comment sont gérées et interrogées les ontologies et les DBO des systèmes existants de BDBO.

5 Besoins des applications pour les BDBO

Comme nous l'avons déjà précisé, les utilisations des ontologies dépassent largement aujourd'hui le cadre du Web Sémantique. Les organisations sont donc amenées à manipuler des données qui proviennent de différentes sources et qui recouvrent des domaines différents. Ces données sont souvent elles-mêmes associées à des ontologies basées sur des formalismes différents. D'où le besoin de pouvoir gérer de façon uniforme plusieurs formalismes d'ontologie.

Nous avons vu que les BDBO de type 2 et 3 utilisent pour la représentation des ontologies, un schéma spécifique basé sur un formalisme d'ontologie particulier ; ce schéma est donc figé. Les BDBO de type 1 utilisent une approche de représentation générique, cependant elles n'offrent pas de mécanismes automatiques permettant d'exprimer la sémantique de nouvelles constructions. Donc en général, un seul formalisme est supporté. Ainsi, aucune des solutions existantes de représentation des ontologies dans les BDBO ne satisfait, en l'état actuel, les besoins des applications. En effet, les BDBO existantes ne disposent de la flexibilité requise ni (1) pour s'adapter aux évolutions des différentes applications d'entreprise qui couvrent des domaines hétérogènes et utilisent des formalismes d'ontologies différents, (2) ni pour permettre la représentation de toutes les données manipulées par les applications dont les types ne sont pas prévus par le formalisme d'ontologie.

Nous avons également vu les solutions suivies pour assurer le raisonnement dans les BDBO. La solution par raisonnement et la solution par saturation permettent de répondre aux interrogations en particulier lorsque des DBO non canoniques sont manipulées. Cependant un grand nombre d'applications n'utilisent qu'un sous-ensemble des primitives non canoniques. Il est donc nécessaire de (1) pouvoir accéder au maximum à toutes les informations canoniques contenues dans les BDBO sans raisonner, et (2) développer des mécanismes permettant d'accéder de manière efficace aux informations non canoniques ou à certaines d'entre elles sans saturation.

5.1 Flexibilité et efficacité de modélisation

Les BDBO doivent disposer de mécanismes permettant la flexibilité de modélisation requise par les applications qui doivent être capables :

1. d'intégrer des ontologies exprimées suivant différents formalismes, puis de les exploiter de façon efficace ;
2. de supporter la représentation des types de données non standards, non prévus par les formalismes d'ontologie.

5.1.1 Intégration d'ontologies exprimées suivant différents formalismes

Il est illusoire de penser que les organisations peuvent construire une ontologie pour toutes les données qu'elles doivent manipuler. Les données manipulées par les applications recouvrent des domaines qui sont en général, très variés. Les organisations ont donc besoin de combiner les données de différentes sources et qui utilisent des formalismes d'ontologie différents.

Les BDBO utilisées doivent donc être capables de représenter des ontologies modulaires définies à partir des ontologies existantes au sein des domaines couverts. La réutilisation d'ontologies existantes facilitera en particulier les échanges entre les différentes applications. Les BDBO doivent donc offrir la capacité d'intégrer des ontologies exprimées selon différents formalismes et de représenter non seulement ce qui est commun aux différents formalismes d'ontologies en jeu, mais aussi et idéalement, permettre de choisir les mécanismes mis en jeu dans ces différents formalismes et qui entrent dans la portée des données à modéliser.

De nombreuses études ont établi la nécessité pour les applications basées sur les ontologies de pouvoir intégrer des constructions de différents formalismes d'ontologies [49, 22]. C'est par exemple le cas pour les applications de commerce électronique qui ont besoin à la fois de la capacité de définir avec une grande précision (contexte, unités) les données et de vérifier les contraintes sur ces dernières comme dans PLIB mais aussi, d'une grande expressivité dans la description des informations comme dans formalisme OWL. Les informations manipulées par ces applications ne peuvent donc pas être complètement modélisées et interprétées en utilisant uniquement soit une BDBO orientée vers le formalisme d'ontologie PLIB, soit une BDBO orientée vers les formalismes d'ontologie RDFS/OWL. L'utilisation d'une BDBO orientée PLIB impliquerait de privilégier les objectifs de précision et de vérification au détriment des objectifs d'expressivité, alors qu'à l'inverse, l'utilisation d'une BDBO orientée RDFS/OWL impliquerait de privilégier les objectifs d'expressivité au détriment des objectifs de précision et de vérification.

Plusieurs discussions ont montré que la plupart des constructions spécifiques, proposées par les différents formalismes d'ontologies sont complémentaires [48, 18, 54, 22, 58]. De nombreuses études [25, 49, 7] ont également permis de montrer (1) la faisabilité de combiner la sémantique des différents formalismes d'ontologie avec les bases de données, et, (2) la plus-value que cette combinaison apporterait aux applications.

Par exemple, dans le contexte du projet e-Wok Hub, visant à gérer la mémoire de plusieurs projets d'ingénierie sur la capture et le stockage de CO2, les experts doivent développer des ontologies pour couvrir ce domaine. Ils doivent choisir un formalisme d'ontologie 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'éva-

luation. Ce besoin suggère l'utilisation d'un modèle d'ontologies tel que PLIB. D'autre part, l'utilisation de certains constructeurs introduits par les formalismes d'ontologies issus de la logique de description, tels que OWL, apparaissent également nécessaires pour permettre d'améliorer la qualité des recherches documentaires qui nécessitent l'exploitation des synonymes terminologiques, également essentielles pour le problème visé. Cet exemple montre que les applications ont besoin de manipuler différentes constructions qui ne sont malheureusement pas disponibles dans un seul formalisme d'ontologie. En conséquence, les BDBO doivent soit permettre d'intégrer efficacement différents formalismes d'ontologies, soit permettre l'évolution du formalisme d'ontologie supporté en fonction des besoins de l'application finale. De plus, en ce qui concerne nos besoins propres, la plupart de nos applications étant basées sur des ontologies PLIB, il est important que la gestion des ontologies PLIB soit effectuée de façon efficace.

5.1.2 Représentation des types de données non standards

Les formalismes d'ontologie (PLIB, RDFS, OWL) intègrent dans leur système de type principalement les types de données simples (caractères, entier, réel, booléen), leurs dérivés et également certains types de données plus élaborés tels que les unités de mesures et les unités monétaires (PLIB). Cependant, ils n'offrent pas de support pour des types de données spécifiques ou plus complexes non prévus dans les formalismes d'ontologies ; nous appelons ces données des données non standards. Les organisations sont souvent amenées à caractériser les données avec des valeurs de types de données plus ou moins spécifiques suivant les domaines couverts ou encore suivant les traitements à effectuer. Par exemple dans le projet e-Wok Hub, la définition du découpage de la France en zone géographique requiert des caractéristiques précises comme la géométrie représentant le contour (coordonnées géométriques) de chaque zone géographique. Les propriétés de type géométrique peuvent s'avérer très utiles pour réaliser de nombreux traitements (inclusion, contiguïté, représentation graphique, ...). Une caractéristique importante du formalisme d'ontologie de la BDBO est donc d'offrir le support pour représenter de nouveaux types de données.

5.2 Gestion efficace de gros volumes de données canoniques et non canoniques

Comme nous l'avons vu dans le modèle en oignon (cf. section 3.3 du chapitre 2), les OCNC et les OL introduisent des extensions aux OCC. En particulier, elles offrent la possibilité de définir des concepts dits définis en introduisant des équivalences conceptuelles. Les systèmes de gestion des ontologies s'appuyant sur de telles ontologies doivent donc être capables de gérer efficacement de gros volumes de données comportant à la fois des données définies et des données primitives. Le problème fondamental qui résulte de la représentation de l'information sous forme de données canonique et/ou non canonique est leur représentation et leur interrogation au sein des BDBO.

Si nous considérons par exemple les classes *Personne* et *Femme* ; avec la classe *Femme* définie comme égale à la restriction de la classe *Personne* aux individus dont la valeur pour la propriété *genre* est égale à *'féminin'* (*Personne [genre = 'féminin']*). La représentation par exemple des informations *(a = Femme[nom = 'Anne'])* et *(b = Personne[nom = 'Paule', genre = 'féminin'])*, au sein des BDBO actuelles implique :

– soit de saturer à la fois les classes *Personne* et *Femme*, ce qui va ainsi permettre de

disposer au sein de la BDBO de toute l'information nécessaire pour répondre aux différentes interrogations. Cette solution est cependant coûteuse car la saturation doit être réalisée à chaque mise à jour ;
- soit de ne pas saturer l'information, ce qui nécessite de réaliser des raisonnements spécifiques (souvent lents) lors de l'interrogation des données que ce soit pour l'accès aux instances de la classe primitive *Personne* ou de la classe définie *Femme*.

Aucune des approches de représentations des DBO ne semble donc totalement satisfaisante pour la gestion des DBO canoniques et non canoniques. Pour les applications manipulant peu ou pas du tout des primitives non canoniques, l'approche par raisonnement impose tout de même de raisonner lors des interrogations sur les classes canoniques. A contrario, l'approche par saturation efficace pour les interrogations portant sur les primitives non canoniques nécessite une re-saturation (coûteuse) des données à chaque mise à jour. Également, les bases de données traditionnelles offrent la possibilité de gérer la redondance par le langage d'interrogation SQL. Ainsi par exemple, les instances de la classe *Femme* pourraient être déterminées à partir de celles de la classe *Personne* en utilisant une vue SQL. Il n'est donc pas toujours nécessaire de saturer les données associées aux primitives non canoniques.

les BDBO doivent pour satisfaire les besoins des applications en terme des gestion des données :
- permettre l'accès au maximum de données canoniques sans raisonner ;
- définir des mécanismes efficaces d'accès à certaines informations non canoniques sans saturation.

Afin de proposer une BDBO adaptée aux besoins actuels des organisations, il est nécessaire d'analyser les différentes BDBO existantes (OntoMS [53], Sesame [11], OntoDB [20], RDFSuite [4], Jena [68], Oracle [13], SOR [44], ...) afin de déterminer si elles répondent aux besoins actuels des applications discutés dans cette section. Compte tenu de la multitude d'architectures de BDBO proposées, nous avons choisi de présenter un ensemble de propositions de BDBO que nous avons considéré comme représentatifs des différentes solutions proposées.

6 Quelques implémentations existantes de BDBO

Nous présentons dans cette section, les BDBO RDFSuite, Jena, OntoDB, Oracle et SOR. L'objectif étant d'identifier les formalismes d'ontologies supportés par chacune de ces BDBO, la représentation utilisée pour les ontologies et les DBO, les mécanismes d'interrogation des DBO fournis et enfin le positionnement de ces BDBO par rapport aux besoins que nous avons identifiés.

6.1 RDFSuite

RDFS*uite* [4] développé par ICS-FORTH[14]a été partiellement supporté par les projets Européen C-Web[15], MesMuses[16] et QUESTION-HOW[17]. RDFSuite est constitué d'un ensemble de modules pour la gestion (stockage, interrogation, validation, échange) de grand volume de données RDF dans des SGBD Relationnels ou Relationnels Objets. Ses principaux modules sont :

[14]http ://www.ics.forth.gr/
[15]http ://cweb.inria.fr/
[16]http ://cweb.inria.fr/Projects/Mesmuses
[17]http ://www.w3.org/2001/qh/

- RSSDB (RDF Schema Specific DataBases) : pour le stockage de grand volume de données de méta-données RDF et RDFS dans une base de données ;
- RQL (RDF Query Language) : un langage de requêtes déclaratif pour l'interrogation des données RDF et RDFS stockées dans la base de données ;
- VRP (Validating RDF Parser) : pour la validation des données (documents RDF).

RDFSuite est une BDBO de type 2 avec un schéma pour l'ontologie fondée sur le formalisme d'ontologie RDFS. Elle ne permet donc pas le support de primitives non canoniques définies dans OWL. Sur le SGBD relationnel/objet PostgreSQL, les données sont représentés dans une approche binaire avec héritage des tables. Les types de données supportés par RDFSuite incluent les entiers, les réels, les chaînes de caractères, les URI, les dates, les types énumérées et les collections dont les éléments ont pour type, un des types précédents.

Il faut préciser que RDFS*uite* exige trois contraintes [4] que doivent respecter les ontologies et leurs données avant d'être stockées dans une base de données à base ontologique.

1. Toutes les propriétés des classes doivent avoir obligatoirement un domaine et co-domaine. Sans cette contrainte de typage fort, une propriété pourrait par exemple déclarer comme co-domaine, une classe et le type de donnée réel, alors qu'en RDFS, l'union des classes et des types littéraux est dépourvue de sens [4].
2. Le domaine et le co-domaine d'une propriété doivent être uniques. Cette exigence vient toujours du fait qu'en OWL et RDFS, une propriété peut déclarer plusieurs domaines et co-domaine. Sans cette exigence, d'une part, il y aurait une ambiguïté dans le choix du type des colonnes des tables des propriétés. C'est une contrainte que nous imposerons également.
3. Enfin, la définition des classes et propriétés doivent être complètes. Cette dernière contrainte de nature syntaxique, impose par exemple que les superclasses d'une classe soient déclarées dans l'expression de la classe et que le domaine et le co-domaine d'une propriété soient également déclarés dans l'expression de la propriété.

6.2 Jena

Jena[18] est un système complet qui permet la manipulation des ontologies RDF, RDFS, DAML+OIL et OWL. Jena adopte une architecture de BDBO de type 1 dans laquelle une table principale de triples est utilisée pour le stockage persistant des ontologies et des données. Les types de données supportés par Jena sont les entiers, les réels, les dates, les chaînes de caractères et les URI.

En plus de son API de programmation Java, Jena offre des supports de persistance pour différents SGBD relationnels (PostgreSQL, MySQL, Interbase, Oracle) via JDBC. Afin de réaliser des inférences sur les données, Jena dispose d'un moteur d'inférence interne. Elle peut également être couplée à des raisonneurs tels que Java Theorem Proover, Racer [66] ou encore FACT [30]. L'ensemble des données est alors transféré vers le raisonneur qui réalise les inférences en mémoire centrale et retourne le résultat l'API Jena. Jena offre également un parseur pour les ontologies et données définies suivant les formats RDF/XML.

La version courante de Jena est *Jena2* [68]. Jena est intégrée dans l'outil *protégé4.0* qui est un éditeur d'ontologie et de DBO. Dans chacune de ces versions des schémas différents de base

[18] http ://jena.sourceforge.net/

de données ont été proposés. Dans *Jena1* [6], tant les ontologies et les DBO sont stockées dans le schéma de l'approche de représentation *verticale*. Dans *Jena2*, l'approche utilisée est la deuxième variante de l'approche *verticale* avec ID où les ressources et les valeurs des propriétés de type littérale (types simples) sont stockées séparément dans les tables *ressources* et *litteral*.

6.3 OntoDB

OntoDB est une BDBO conçue pour supporter l'évolution du schéma des ontologies et pour offrir un accès aux données au niveau ontologique. Actuellement, OntoDB est implémentée sur le SGBD PostgreSQL. Elle consiste en 4 parties (figure 2.8) : (1) la partie données contenant les données d'instances et (2) la partie méta-base qui contient le catalogue système. A ces deux parties habituelles disponibles dans tous les SGBD, OntoDB ajoute (3) la partie ontologie qui permet de représenter complètement les différentes ontologies et (4) la partie méta-schéma qui permet de représenter, au sein d'un modèle réflexif, à la fois le modèle d'ontologie utilisé et le méta-schéma lui-même (cette partie permet de rendre générique beaucoup de traitements sur les ontologies). Les DBO sont représentées dans OntoDB en utilisant une approche horizontale : une table est créée pour chaque classe de l'ontologie, ses colonnes correspondent au sous-ensemble des propriétés applicables de la classe, c'est à dire celles qui sont utilisées par au moins une instance de la classe.

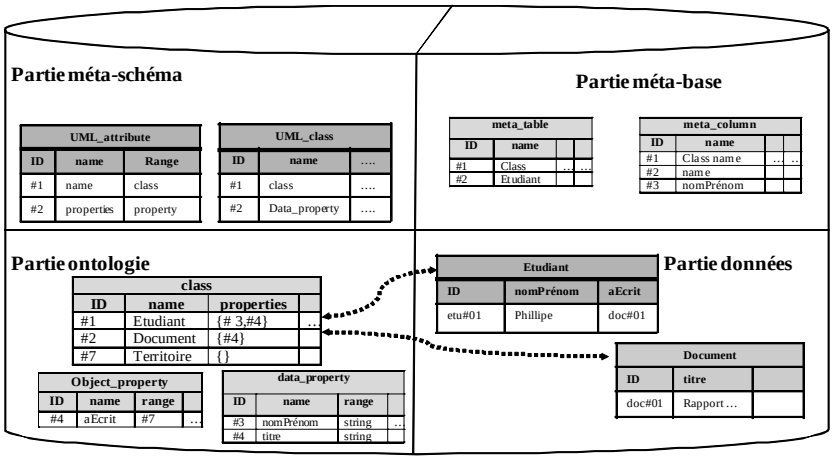


FIG. 2.8 – Architecture d'OntoDB

Comme RDFSuite, OntoDB se fonde sur plusieurs hypothèses fondamentales :

1. **Typage fort des propriétés.** Comme RDFSuite, OntoDB impose le typage fort des propriétés.
   - toutes les propriétés des classes doivent avoir obligatoirement un domaine et un co-domaine,
   - le domaine et le co-domaine d'une propriété doivent être uniques.
2. **Complétude de définition.** Comme RDFSuite, OntoDB exige que les descriptions complètes de tous les concepts qui contribuent à la définition d'un concept soient fournis au

système lorsque ce concept est introduit. Ceci suppose en particulier que les super-classes d'une classe, si elles existent, soient fournies, de même que les domaines et le co-domaine de ses propriétés.

3. **Mono-instanciation.** Toute instance représentée est décrite par son appartenance à une et une seule classe, dite classe de base qui est la borne inférieure unique pour la relation de subsumption de l'ensemble des classes auxquelles elle appartient (elle appartient bien sûr également à ses super-classes).
4. **Typage fort des DBO** Toute instance du domaine ne peut être décrite que par les propriétés applicables à sa classe de base, c'est-à-dire celles dont le domaine subsume cette classe de base.

La représentation horizontale adoptée par OntoDB passe bien à l'échelle lorsque de nombreuses propriétés sont utilisées par instance[20]. Les DBO et les ontologies contenues dans OntoDB sont manipulées grâce à un langage de requête qui est une extension ontologique du langage SQL appelé OntoQL [36].
OntoDB étant fondée sur le formalisme d'ontologie PLIB, il n'offre pas de support pour la gestion des données non canoniques.

6.4 Le système d'Oracle

Oracle propose le premier système de gestion de données RDF au dessus d'une base de données commercialisée. Dans ce système, Oracle propose de stocker des ontologies RDFS, OWL et données RDF sous forme de triplets dans la table RDF_L*ink$*. RDF_L*ink$* stocke les triplets sous une forme normalisée en utilisant des identifiants. La correspondance entre chaque identifiant de la table RDF_L*ink$* et l'URI correspondante est réalisée dans la table RDF_V*alues$*. Le stockage de littéraux est également réalisé dans la table RDF_V*alues$*. La première forme rencontrée d'un littéral est considérée comme sa forme canonique. Les formes équivalentes rencontrées ensuite référencent la forme canonique pour faciliter les comparaisons.

Le système d'Oracle propose également un langage de règles logiques dont les résultats sont susceptibles d'être matérialisés. Afin de réduire le coût de jointure lié à l'utilisation de la table principale de triplets RDF_L*ink$*, des bases de règles prédéfinies représentent les principaux axiomes de RDFS et OWL. Les résultats de ces bases de règles peuvent être sauvegardés dans une structure redondante (vue matérialisée). De plus, le système offre la possibilité de définir implicitement de nouveaux triples par des règles. Ces nouveaux triples pouvant eux-mêmes être matérialisés par des vues. Cette extension d'Oracle constitue donc un système autonome, permettant de gérer efficacement les données RDF/RDFS du Web sémantique.

Par défaut, Oracle ne réalise pas d'inférence lors du traitement des requêtes. Afin de réaliser des inférences sur les données, l'utilisateur doit explicitement spécifier lors de chaque interrogation les bases de règles à utiliser. Une caractéristique importante de l'approche proposée ici est de permettre l'interrogation de ces données dans une syntaxe SQL par l'introduction d'une fonction RDF_MATCH qui accède aux données RDF/RDFS/OWL. La même requête SQL permet donc d'accéder à la fois aux données RDF/RDFS, et aux autres données représentées dans la base de données. Notons que cette caractéristique permet d'offrir simultanément un accès traditionnel aux données, à partir des tables SQL, et un accès à partir d'ontologies.

Le système proposé par Oracle représente donc une infrastructure qui peut être utilisée soit (1) comme une BDBO de type 1 où les informations sur les ontologies et les données sont stockées dans les mêmes tables dans une approche verticale avec ID, soit (2) comme une BDBO de type 2 où les ontologies sont stockées suivant l'approche verticale avec ID, et les données sont stockées dans les tables relationnelles usuelles de la base de données. Notons également que la sémantique des ontologies et des données représentées peut être enrichie en définissant des bases de règles utilisateurs pour exprimer des règles métiers par exemple.

6.5 SOR

Le système SOR (Scalable Ontology Repository)[44] proposé par IBM sur le SGBD relationnel DB2 et le SGBD relationnel embarqué Java Derby, permet de gérer des ontologies OWL. SOR est une BDBO de type 2 basée sur le formalisme d'ontologie OWL. Ainsi une table dans la base de données est associée à chaque prédicat du formalisme d'ontologie OWL (ex. *typeOf*, *someValuesFrom*, *intersectionOf*, *domain*, etc.). Dans SOR, les DBO sont représentées suivant une approche verticale avec ID. Afin d'optimiser les traitements sur les données, SOR comporte à la fois un raisonneur sur l'ontologie et un moteur d'inférence sur les données afin de pré-calculer et de matérialiser tous les axiomes (ex. *subClassOf*, *subPropertyOf*, *inverse*, *typeOf*, etc.) et tous les nouveaux faits ou triplets inférés dans la base de données. Enfin, SOR propose un traducteur SPARQL-*to*-SQL doté d'une gestion efficace des patrons utilisées par SPARQL pour l'interrogation de données.

6.6 Synthèse sur les BDBO existantes

Les systèmes de BDBO que nous avons étudiés offrent aux applications, une solution de persistance d'ontologies et de DBO au sein du même référentiel. Cependant, aucune de ces BDBO ne permet de satisfaire complètement les besoins des applications que nous avons identifiés.

- L'étude que nous avons réalisée sur les BDBO Jena, RDFSuite, OntoDB, Oracle et SOR montre que les bdbo de type 2 RDFSuite et SOR ont un schéma d'ontologie figé par rapport aux formalismes d'ontologie RDFS et OWL respectivement. La BDBO de type 1 Jena, qui utilise une représentation par triple ne fournit aucun mécanisme pour enrichir les API de gestion de manière à permettre la définition de nouvelles constructions. OntoDB possède également un schéma d'ontologie figé pour le formalisme d'ontologie PLIB. Enfin, l'infrastructure proposée par Oracle fournit des bases de règles prédéfinies pour les formalismes RDFS et OWL, et la possibilité de définir de nouvelles règles. Oracle permet donc en cas de besoin, et ce moyennant des efforts de programmation, de définir de nouvelles bases de règles pour d'enrichir la sémantique des ontologies et des données.
- Également, ces différentes BDBO permettent uniquement la représentation des données des types prévus par le formalisme d'ontologie support. L'infrastructure proposée par Oracle qui permet avec la même requête SQL d'accéder à la fois aux données RDF/RDFS et aux données usuelles représentées dans la base de données, fournit ainsi la possibilité de supporter en plus, les types de données définies dans la base de données relationnelles. Il ne permet toutefois pas d'introduire dynamiquement de nouveaux types de données. Enfin le méta-schéma de l'architecture de type 3 d'OntoDB pourrait servir à doter ce dernier des mécanismes nécessaires pour enrichir des types de données supportés.

– Concernant la gestion et l'interrogation des données, les différentes BDBO disposent d'un langage d'interrogation des données. Cependant, RDFSuite et OntoDB n'offrent pas de support pour la représentation des données non canoniques. La BDBO Jena représente à la fois les données canoniques et les données non canoniques sur le support de persistance. Elle nécessite donc d'accéder à un raisonneur pour réaliser les inférences ce qui induit un coût supplémentaire lors de l'interrogation des données. Enfin, les BDBO Oracle et SOR adopte une approche par saturation pour la gestion des données. Bien que cette approche permette de répondre aux interrogations sans inférence, elle entraîne un coût important de mise à jour des données. En effet, SOR réalise des inférences à chaque nouvelle insertion et rajoute les nouveaux faits inférés afin de conserver la base de données dans un état toujours saturé. Dans Oracle, il est nécessaire de mettre à jour les vues associées aux différentes bases de règles après chaque modification des données.

La table 2.2 résume la comparaison de ces différents systèmes par rapport aux besoins que nous avons identifiés à la section 5.

	RDFSuite	Jena	OntoDB	Oracle	SOR
Ontologie					
formalisme d'ontologie	RDFS	RDFS/OWL	PLIB	RDFS/OWL	OWL
flexibilité du formalisme d'ontologie	non	non	oui	non	non
DBO					
support de types de données non standards	non	non	oui	non	non
DBO non canoniques	non	oui	non	oui	oui
Raisonneur	externe	externe		interne	externe
Saturation	non	oui		oui(vue)	oui
Langage de requête	RQL	AQL	OntoQL	SQL	SPARQL

TAB. 2.2 – Comparaison des architectures de BDBO

Conclusion

Nous avons présenté dans ce chapitre les notions de Données à Base Ontologique (DBO) et de Bases de Données à Base Ontologique (BDBO) qui permettent la représentation et la gestion à la fois des DBO et des ontologies qu'elles référencent au sein des bases de données. Ceci permet de profiter de leur maturité et de leurs performances pour la gestion de volumes croissant de données. Nous avons détaillé ensuite les principales architectures proposées dans la littérature pour représenter les ontologies et les données. Suivant le schéma de représentation choisi pour les ontologies, nous avons classifié les architectures de BDBO comme étant :

– soit des BDBO de type 1 susceptible de représenter également les ontologies et les DBO associées dans une seule table ;
– soit des BDBO de type 2 avec deux schémas distincts, respectivement pour les DBO et pour ontologies, ce dernier étant figé et constitué de tables spécifiques du formalisme d'ontologie supporté et enfin ;
– soit des BDBO de type 3 qui, par rapport au type 2, offrent la possibilité de faire évoluer le

formalisme d'ontologie utilisé grâce à l'ajout et la représentation dans la BDBO du méta-modèle du formalisme d'ontologie.

Nous avons également présenté les trois approches suivies pour la représentation des DBO.

- une approche verticale avec une unique table de triplets,
- une approche binaire dans des tables spécifiques définies pour chaque classe et propriété utilisées de l'ontologie et,
- une approche horizontale qui utilise un schéma similaire au modèle de données des bases de données usuelles : chaque classe de l'ontologie est associée une table avec une colonne pour chaque propriété applicable aux instances.

A l'issue de cette discussion, nous avons analysé les besoins auxquels devraient désormais faire face les systèmes de gestion de BDBO pour satisfaire les besoins des diverses applications qui apparaissent. Nous avons ainsi identifié quatre besoins :

1. la flexibilité et l'efficience du formalisme d'ontologie afin de s'adapter à la diversité des besoins des applications ;
2. la flexibilité du système de types de données pour permettre la représentation de données non standards lorsque le besoin ce fait sentir ;
3. la gestion efficace des DBO canoniques sans inférence ;
4. la gestion des DBO non canoniques, sans saturation des données.

Analysant ensuite un ensemble de systèmes existants et représentatif de l'offre actuelle, nous avons identifié que ces quatre besoins n'étaient pas bien couverts.

Afin de répondre aux besoins identifiés, le but de cette thèse est de proposer une nouvelle architecture de BDBO fondée sur un formalisme d'ontologie capable d'être adapter de manière à choisir et à combiner les constructions mises en jeu suivant le problème à résoudre. Pour cela, cette architecture de BDBO devra :

1. se fonder sur un formalisme d'ontologie flexible qui permet d'exprimer des ontologies de différentes sources et dans lequel on peut combiner les constructions suivant les besoins de l'application à mettre en œuvre. Ce formalisme devra également supporter de façon efficace le formalisme PLIB ;
2. disposer d'un système de types flexible et extensible afin de permettre la représentation des données non standards ;
3. offrir des outils efficaces de gestion et d'interrogation à la fois des DBO canoniques et des DBO non canoniques pour pouvoir supporter de gros volumes de données.

Dans le chapitre suivant, nous présentons l'architecture de BDBO que nous proposons dans le cadre de notre thèse pour obéir à ces exigences.

Deuxième partie

Notre proposition d'architecture

Chapitre 3

Description du modèle OntoDB2

Sommaire

Introduction

En réponse aux exigences identifiées au chapitre 2, nous présentons dans ce chapitre les solutions que nous proposons pour y répondre. L'ensemble de ces propositions définit une nouvelle architecture de bases de données appelée OntoDB2.

OntoDB2 permet de supporter des constructions issues de différents formalismes d'ontologie (PLIB, et OWL Lite). Pour cela, OntoDB2 est fondée sur une architecture de type 3 qui lui confère la possibilité de faire évoluer le formalisme d'ontologie supporté. Le formalisme d'ontologie supporté dans OntoDB2 est constitué :

1. d'un noyau défini à partir des constructions canoniques communes aux formalismes d'ontologie PLIB et OWL Lite, associé aux constructeurs canoniques existants dans PLIB ;
2. d'extensions du noyau définies suivant les besoins de chaque application particulière, le formalisme d'ontologie global étant dans tous les cas généré en utilisant des générateurs de code à partir d'une spécification formelle des besoins ;
3. des extensions particulières réalisées par nous et permettant de traiter les constructions non canoniques de OWL Lite.

Concernant la gestion des informations non canoniques, OntoDB2 propose une solution originale qui permet d'éviter aussi bien saturation que raisonnement pendant les requêtes. Cette solution appelée *migration d'instances*, consiste à convertir lors du chargement de la base de données, toutes les informations non canoniques pour lesquels c'est possible, en leur représentation canonique. OntoDB2 permet alors

- d'accéder, sans aucun raisonnement, à l'ensemble de l'information canonique existant, y compris celle qui représente, comme instances de classes canoniques, de l'information initialement fournie par des instances de classes non canoniques et,
- de calculer, grâce à des vues, l'extension des classes définies (non canoniques) à partir de celle des classes primitives (canoniques), éliminant ainsi la redondance des données au sein de la BDBO.

Ce chapitre est composé de quatre sections. Dans la première section, nous présentons le type d'architecture adopté pour la BDBO OntoDB2. La deuxième section présente le formalisme d'ontologie noyau d'OntoDB2 ainsi que les extensions qui permettent d'adapter ce formalisme d'ontologie afin d'intégrer des constructions issus du formalisme d'ontologie OWL Lite. Nous y présentons également l'extension du système de types de données. Dans la troisième section, nous précisons d'abord le cadre d'hypothèses d'OntoDB2. Ensuite, nous présentons une synthèse du niveau ontologique d'OntoDB2 en précisant pour le formalisme OWL Lite les constructions que nous souhaitons supporter. La quatrième section est consacrée à la gestion et l'interrogation des DBO. Tout d'abord nous présentons comment sont représentées les DBO canoniques et non canoniques. Nous définissons ensuite les mécanismes permettant de raisonner efficacement sur les DBO non canoniques au sein de la BDBO OntoDB2 et, nous discutons des autres mécanismes de raisonnement implantés pour traiter les caractéristiques de propriété et l'indexation de données spatiales. Enfin, nous discutons du positionnement d'OntoDB2 par rapport aux architectures existantes présentées au chapitre précédent.

1 Choix du type d'architecture pour la BDBO OntoDB2

Les deux premiers besoins que nous avons identifiés sont :

1. la possibilité de pouvoir modifier aisément le formalisme d'ontologie afin de pouvoir intégrer des ontologies basées sur différents formalismes ;

2. la possibilité de pouvoir étendre le système de types de données utilisable afin de représenter de nouvelles données.

L'étude des architectures de BDBO faite au chapitre précédent a montré que seule l'architecture de types 3 offre la possibilité de représenter au sein de la base de données le méta-modèle du formalisme d'ontologie et le formalisme d'ontologie en tant qu'instance de ce méta-modèle. Cette caractéristique permet d'assurer la flexible du formalisme d'ontologie en ce sens que dans une telle architecture, la partie ontologie n'est pas figée. En effet, la disponibilité de la partie méta-schéma et la représentation du formalisme d'ontologie utilisé dans la BDBO (au sein de la partie méta-schéma) permet d'assurer l'évolution de la partie ontologie. Elle rend générique l'accès aux ontologies représentées. De plus, ce méta-modèle va permettre de représenter les évolutions et les modifications éventuelles du formalisme d'ontologie.

Cette dernière caractéristique est essentielle pour OntoDB2. En effet, elle va permettre d'assurer la flexibilité du système de types supportés et l'extensibilité du formalisme d'ontologie noyau que nous définissons à la section 2.

2 Flexibilité et efficience du formalisme d'ontologie

L'étude que nous avons faite des formalismes d'ontologies PLIB, RDFS et OWL nous a permis d'identifier deux caractéristiques fondamentales partagées par ces différents formalismes d'ontologies.

1. Pour modéliser un domaine donné, ces formalismes permettent de définir des catégories d'objets (classes) et les différentes caractéristiques qui s'appliquent à ces objets (propriétés) ainsi que, les domaines de valeurs (types de données) que peuvent prendre les caractéristiques des objets.
2. Les ontologies conçues suivant ces formalismes d'ontologies peuvent se décomposer suivant un modèle en oignon au sein duquel on retrouve (1) une couche canonique qui contient les primitives de base permettant de décrire de façon unique les informations à modéliser. Ces primitives sont voisines à tous les formalismes d'ontologies. Au dessus de la couche canonique se trouve (2) une couche non canonique constituée principalement de constructeurs d'équivalences conceptuelles. Ces constructeurs d'équivalences conceptuelles permettent de définir différentes vues sur les classes de la couche canonique et de leur associer différents noms. Comme nous l'avons vu, les constructeurs d'équivalence conceptuelle sont pour la plupart orthogonaux et peuvent coexister sans conflit au sein de la même architecture (cf. section 3 du chapitre 1). Enfin, (3) une couche linguistique qui au travers de différentes descriptions textuelles (nom préféré, définition et commentaire) permet d'enrichir la description informelle des classes et propriétés de l'ontologie et, en particulier, offre la possibilité d'associer à chaque concept des descriptions dans différentes langues.

Partant de ce constat, notre proposition est de nous appuyer sur le modèle en oignon et sur le fait que les constructions canoniques sont très proches dans les différents formalismes d'ontologie pour définir un formalisme constitué de deux parties :

1. une partie noyau, représentée explicitement au sein de la partie méta-modèle, constituée des constructions canoniques essentielles présentent dans les différents formalismes d'ontologie ;

2. une couche périphérique, définie lors de l'initialisation du système, permettant de supporter les extensions de modélisation nécessaires pour les applications visées. Certaines de ces extensions, fréquemment utilisées dans les applications, seront également modélisées explicitement représentées dans ce méta-modèle d'ontologie pour rendre leur utilisation immédiatement possible sans aucun paramétrage particulier du système.

La spécification d'un tel formalisme d'ontologie nécessite :

1. d'en définir précisément le noyau de base (constructions de base) de telle sorte que ce dernier puisse être par la suite enrichi par de nouvelles constructions,
2. de déterminer comment ce noyau peut être enrichi. Le choix des nouvelles constructions pour étendre le noyau dépendra par exemple des besoins de l'application à mettre en œuvre. Ces constructions doivent cependant pouvoir coexister simultanément au dessus du formalisme d'ontologie noyau. Dans notre cas nous nous intéressons à deux extensions : (1) voir quels sont les constructions d'OWL Lite que nous pouvons supporter et (2) voir comment supporter les types de données géométriques.

Pour définir le formalisme d'ontologie d'OntoDB2, nous nous sommes limité aux formalismes d'ontologies PLIB et OWL Lite qui sont les principaux formalismes utilisés dans les domaines qui nous intéressent.

2.1 Description du formalisme noyau d'OntoDB2

La définition du noyau de base du formalisme d'ontologie d'OntoDB2 nécessite d'identifier les constructions qu'il doit comporter. Ces constructions doivent non seulement (1) permettre comme pour tout formalisme d'ontologie de structurer l'information (en terme de classes et de propriétés caractéristiques), mais aussi, il doit être (2) suffisamment expressif pour permettre de représenter les constructions communes aux formalismes d'ontologies PLIB et OWL en respectant les caractéristiques spécifiques de chacun et (3) suffisamment puissant pour modéliser le reste du modèle PLIB de façon efficace. Enfin, le formalisme noyau doit (4) posséder une structure assez simple afin d'être facilement compréhensible et extensible.

Comme nous l'avons déjà précisé, l'ensemble des constructeurs canoniques peut servir de base à la définition de notre noyau. Nous avons également étudié plus précisément dans la section 3.3 du chapitre 1, les constructeurs des formalismes PLIB RDFS et OWL qui composent la couche canonique. Cette étude a montré que dans certains cas, un choix devrait être réalisé car, ces différents constructeurs ne peuvent pas toujours co-exister simultanément dans le même environnement : définir notre formalisme noyau à partir de ces constructeurs nécessite donc de réaliser certains choix.

Toutefois, un de nos objectifs étant d'avoir un système PLIB complet, ce dernier doit donc être choisi en priorité en cas de conflits car nous souhaitons le supporter complètement. Compte tenu de cet impératif, nous avons choisi de nous baser sur PLIB pour définir le formalisme noyau. PLIB dispose d'un méta-schéma (EXPRESS) associé à un environnement complet (ECCO) d'ingénierie dirigé par les modèles. Cet environnement va nous être très utile, comme nous le verrons au chapitre suivant, dans la mise en œuvre de notre solution. Cependant, PLIB est très riche et, en particulier, il possède une hiérarchie profonde de spécialisation des constructions qu'il offre. Afin d'assurer l'efficience du formalisme noyau, de pouvoir facilement l'étendre et le spécialiser suivant

les besoins, ce noyau doit être simple. Pour cette raison, une transformation très importante de la structure du formalisme PLIB doit être réalisée. Cette simplification est présentée à la section 2.2 du chapitre suivant.

Nous présentons dans la suite, les constructions canoniques qui composent le noyau de base du formalisme d'ontologie que nous proposons et, les raisons qui ont guidé le choix de ces constructions. Nous ferons toujours le choix de l'approche retenue par PLIB concernant les éléments incompatibles.

2.1.1 Notre proposition : un formalisme d'ontologie noyau basé sur PLIB

Toute ontologie possède au minimum une couche canonique qui permet de décrire sans ambiguïté les concepts du domaine à modéliser. Ce constat permet de définir une contrainte importante du formalisme noyau d'OntoDB2 :

le formalisme noyau doit comporter les primitives nécessaires pour définir des classes et des propriétés primitives (canoniques).

Le formalisme noyau doit donc être défini à partir des constructions canoniques des formalismes d'ontologies PLIB et OWL. Ces différents formalismes d'ontologies décrivent les domaines en terme de classes et de propriétés qui les caractérisent. Le co-domaine de chaque propriété est défini par un domaine de valeurs qui peut être soit une classe, soit un type de donnée primitif. Nous décrivons ces différents éléments dans les sections suivantes.

2.1.1.1 Classe

Unicité de l'identifiant

Comme nous l'avons vu lors de l'étude des formalismes d'ontologies au chapitre 1, tous les formalismes d'ontologies permettent d'associer aux classes et aux propriétés des identifiants. En particulier ces identifiant sont exploités par les individus des ontologies pour référencer la ou les classes auxquelles ils appartiennent, ainsi que les propriétés utilisées pour les caractériser. Notre choix a donc été, comme c'est le cas dans les formalismes d'ontologies PLIB et OWL, d'identifier les classes du formalisme noyau par un identifiant globalement unique qui permet de les référencer de manière universelle.

Toute classe, définie suivant le formalisme noyau possède un identifiant universel.

Hiérarchie de subsumption / Importation de concepts

Tous les formalismes d'ontologie permettent de structurer les classes suivant des hiérarchies de subsumption. Dans le formalisme d'ontologie PLIB, l'héritage associé à ces hiérarchies est simple alors que dans les formalismes d'ontologies RDFS et OWL, cet héritage peut être multiple.

Par ailleurs, concernant la référence d'une ontologie à une autre, le formalisme d'ontologie OWL intègre la possibilité d'étendre des ontologies existantes, en important des ontologies externes pour compléter la définition d'une nouvelle ontologie. A cette fin, OWL, utilise les identifiants uniques (des URI) pour définir des espaces de nommage indiquant à quelle ontologie se rapporter lorsque l'on parle de classes ou de propriétés externes à l'ontologie courante. L'utilisation de classes ou de propriétés définies dans un espace de nommage donnée externe, font que les outils, accèdent et importent globalement à l'ensemble des classes et propriétés de l'ontologie associé à cet espace de nommage.

A l'inverse, PLIB possède un constructeur *is_ case_ of* (cf. section 2.1.2.1 du chapitre 1) qui permet d'importer individuellement les classes de différentes ontologies externes. Ce mécanisme permet en particulier, de n'importer d'une ontologie que la ou les classes dont on a besoin et, pour chacune de ces classes d'importer seulement le sous-ensemble des propriétés pertinentes pour l'application à mettre en œuvre. L'approche modulaire proposée par PLIB est à notre sens particulièrement intéressante car, elle peut être utilisée (1) pour simuler l'héritage multiple au sens d'OWL (en important explicitement toutes les propriétés des classes référencées par le constructeur *is_ case_ of*). Elle permet en plus, (2) de créer des ontologies indépendantes qui référencent et utilisent uniquement des concepts spécifiques d'autres ontologies, créant ainsi une sorte d'ontologie globale. Cette approche permet de faciliter les échanges futurs entre différentes applications qui partageraient la même ontologie globale.

A partir du constat ci-dessus, nous avons défini pour le formalisme noyau la contrainte suivante :

Héritage simple

Les classes peuvent être organisées en une taxonomie d'héritage simple par une relation de subsumption avec héritage. Toutefois, une classe peut également définir des liens de subsumption (sans héritage) avec d'autres classes de la même ontologie ou d'autres ontologies avec importation (sélective) des propriétés (en terminologie PLIB *is_ case_ of).*

Multi-instanciation et point de vue

Le formalisme d'ontologie PLIB propose une approche de représentation par point de vues qui offre la possibilité de représenter un objet modélisé par plusieurs instances, chaque instance étant caractérisée par une instance qui définit le point de vue qu'elle représente. Les classes PLIB sont donc découpées en trois catégories : (1) les classes qui définissent des objets, par exemple une vis (*general model class* = classe de définition), (2) les classes qui définissent un point de vue, par exemple *« les conditions commerciales pour différents types de client »* (exemple : Particulier, Entreprise) (*functional view class* = point de vue) et (3) les classes définissant la représentation dans un certain point de vue pour une certaine classe d'objets, par exemple les

« conditions_ commerciale_ de_ vis » (*functional model class* = modèle fonctionnel) qui décrit par exemple le prix de différents vis en fonction du type de client. Les différentes classes susceptibles d'être associées à une classe de vis sont spécifiées. On ne peut donc pas associer à une vis une instance de la classe *« conditions_ commerciales_ de_ batteries »* qui définit le coût des batteries. Les liens possibles entre une classe de définition et ses modèles fonctionnels possibles sont définis au niveau ontologie et prédéfinissent donc les agrégats d'instances possibles.

A contrario, l'approche par multi-instanciation suivie par OWL, ne permet pas d'associer à chaque classe d'une multi-instanciation un sens particulier, par exemple un point de vue, ou de limiter la multi-instanciation à des classes prédéfinies. La multi-instanciation de OWL correspond à un typage faible. On ne sait pas a priori à quelles classes une instance appartient. Nous choisissons donc d'implanter le mécanisme PLIB. En conséquence :

Mono-instanciation et agrégat d'instances

Une instance ne peut appartenir qu'à un ensemble de classes ayant un minorant pour la relation de subsumption avec héritage, cette classe est appelée sa classe de base. Par contre le formalisme noyau permet de représenter chaque objet par un agrégat d'instances, chaque instance représentant soit la définition de l'objet, soit la vue de l'objet suivant un point de vue particulier (en terminologie PLIB *is_ view_ of), lui-même défini par une instance de classe.*

2.1.1.2 Propriété

Le formalisme noyau doit permettre de définir des propriétés. Une propriété doit posséder un identifiant globalement unique.

Typage fort des propriétés

Le formalisme d'ontologie PLIB impose le typage fort des propriétés. De ce fait, le domaine d'une propriété est toujours connu, c'est la classe sur laquelle elle est définie. De même, toute propriété doit préciser son co-domaine. La notion de typage fort est également présente dans le formalisme OWL Lite car ce dernier exige que l'objet de tout triplet de prédicat *rdfs :domain* soit une classe de l'ontologie et que l'objet de tout triplet de prédicat *rdfs :range* soit, une classe de l'ontologie ou un type de données. Enfin, RDFS et OWL DL permettent de spécifier plusieurs classes comme domaine d'une propriété. Dans ce cas, c'est l'intersection de ces classes qui constitue le domaine de la propriété. A contrario, la BDBO RDFSuite fondée sur le formalisme d'ontologie RDFS, impose pour assurer la consistance de l'ontologie et des instances associées, qu'une propriété définisse un unique domaine et un unique co-domaine.

Pour assurer une implémentation efficace du formalisme d'ontologie d'OntoDB2, nous avons choisi comme, c'est le cas pour les architectures OntoDB et RDFSuite, d'adopter le typage fort des propriétés.

Typage fort des propriétés

Une propriété doit obligatoirement définir une unique classe à laquelle elle s'applique (son domaine) et un unique domaine de valeurs (co-domaine) dans lequel elle prend ses valeurs. La propriété est dite applicable à son domaine et à toutes des sous-classes.

Cardinalités des propriétés

Dans les formalismes d'ontologies PLIB et OWL, la distinction propriété monovaluée (en terminologie OWL : fonctionnelle) / propriété multivaluée est présente. Dans le premier cas elle a au maximum une valeur par instance du domaine. Pour une propriété multivaluée, il est possible de déclarer une cardinalité minimale et une cardinalité maximale qui correspondent respectivement au nombre minimum et maximum de valeurs qu'elle peut avoir pour toute instance de son domaine. Les formalismes d'ontologies PLIB, RDFS et OWL distinguent quatre catégories de collections :

- les listes, définissant une collection ordonnée où la répétition de valeurs est permise ;
- les sacs, définissant une collection non ordonnée où la répétition de valeurs est permise ;
- les séquences, définissant une collection ordonnée où la répétition de valeurs n'est pas permise ;
- les ensembles, définissant une collection non ordonnée où la répétition de valeurs n'est pas permise.

Nous adoptons la même caractérisation dans le formalisme noyau :

Une propriété peut être monovaluée ; dans ce cas, elle admet au plus une valeur pour toute instance de son domaine. Elle peut être multivaluée (liste, sac, séquence, ensemble) ; dans ce cas, elle a une collection de valeurs éventuellement vide pour une instance donnée de son domaine.

Propriété de type objet / Propriété de type simple

Également, les formalismes d'ontologies (à l'exception d'OWL Full) distinguent l'ensemble des propriétés dont le domaine de valeurs est composé des instances de classes de l'ontologies (associations, appelées propriétés de type objet), de celles dont le domaine de valeurs est un type de données dit simple défini par le formalisme d'ontologie (propriétés de type simple). Nous adoptons pour le formalisme noyau ces distinctions :

Une propriété peut être soit une propriété de type objet auquel cas son co-domaine est une classe de l'ontologie, soit une propriété de type simple si son co-domaine est un type de données simple du formalisme d'ontologie.

Propriété dépendante

Au contraire d'OWL qui ne permet pas d'établir des liens entre propriétés, le formalisme d'ontologie PLIB permet de préciser le contexte d'évaluation dans lequel la valeur d'une propriété est définie. Comme nous l'avons vu à la section 3.3.1 du chapitre 1, cette construction canonique est complémentaire aux constructions canoniques des autres formalismes d'ontologies. De plus, elle est nécessaire dans les applications d'ingénierie et les applications de e-commerce où l'objectif de précision dans la représentation de l'information est important. Nous l'incluons donc dans le formalisme d'OntoDB2.

Une propriété peut préciser le contexte dans lequel est évaluée sa valeur ; celui-ci est lui-même défini par un ensemble de valeurs de propriétés.

2.1.1.3 Types de données

Le formalisme d'ontologie PLIB intègre dans sa spécification les types de données simples incluant : les chaînes de caractères, les entiers, les réels et les booléens. Ces types de données sont également disponibles dans le système de types de données simples issues de XML Schéma qui est intégré dans les spécifications des formalismes d'ontologies et OWL .

Les types de données définis par le formalisme noyau, sont les types de données simples (caractères, entier, réel, booléen, etc.) supportés par les formalismes d'ontologies PLIB, RDFS et OWL.

2.1.1.4 Méta-descripteurs

Tous les formalismes d'ontologies, offrent la possibilité de rattacher des descriptions textuelles aux différents concepts. Ces descriptions permettent de définir informellement quel objet du monde réel est dénoté par un concept. Comme nous l'avons vu à la section 3.3.3 du chapitre 1, le formalisme d'ontologie PLIB offre un grand nombre de descriptions. Ces dernières permettent de décrire de façon très précise tout concept auquel elles sont rattachées. Les descriptions sont d'autant plus importantes, que dans les documents par exemple, de nombreux termes synonymes peuvent être associés au même concept. De plus, la possibilité de définir les descriptions dans différentes langues permet d'offrir un accès multilingues aux concepts. Notons que, la plupart des systèmes existants (Protégé [19], Jena [20], ...) n'interprètent pas les méta-descripteurs rattachées au concepts. Dans OntoDB, ces dernières sont principalement utilisées par le langage OntoQL pour accéder dans différentes langues à un concept via son nom préféré spécifié dans ces différentes langues. Ce constat nous a amené à ne considérer que les méta-descripteurs textuels, et à ne pas intégrer au formalisme noyau les descripteurs RDFS et OWL telles que *seeAlso* et *definedBy*, qui

[19] http ://protege.stanford.edu/
[20] http ://jena.sourceforge.net/

peuvent référencer des instances de concepts de l'ontologie.

Méta-descripteurs textuels

Seuls des méta-descriptions textuelles peuvent être rattachées aux classes et aux propriétés de l'ontologie.

2.1.1.5 Individus

Rappelons que nous avons décidé de ne pas implanter la multi-instanciation (voir 2.1.1.1).

Typage fort des instances

Tout individu doit contenir une référence à sa classe de base et, ne peut être décrit que par les propriétés applicables à sa classe de base.

L'ensemble des classes auxquelles peut appartenir un individu, possède un minimum pour la relation de subsumption. Nous appelons ce minimum, la classe de base de l'individu. C'est à cette dernière que sera rattaché cet individu et, seules les propriétés applicables de sa classe de base pourront être utilisées pour le décrire.

2.1.2 Représentation simplifiée du noyau

La figure 3.1 illustre une représentation simplifiée du formalisme noyau d'OntoDB2.
Afin d'éviter toute confusion avec les termes *classe* et *propriété* utilisés pour les ontologies, nous emploierons dans la suite le terme *entité* pour identifier les constructeurs de classes et de propriétés, et le terme *attribut* pour identifier les caractéristiques des entités de niveau formalisme d'ontologie. Les termes *classe* et *propriété* seront réservés effectivement au niveau ontologie qui est une instanciation du formalisme d'ontologie.

Ce schéma simplifié comporte les entités *Ontology*, *Class*, *Property*, *Data_type* et *Individual* qui permettent respectivement de définir des ontologies, des classes, des propriétés, des types de données et des instances.

L'entité *Ontology* est caractérisée par un identifiant (BSU) et contient un ensemble de classes (*contained_classes*).

L'entité *Class* est caractérisée par un identifiant (BSU), une super-classe éventuelle (*subclassOf*) et, zéro ou plusieurs (0..n) relations de subsumption avec d'autres classes (*is_case_of*) ; chacune de ces relations permet d'importer des propriétés (*imported_properties*). Une clase déclare d'autre part, les nouvelles propriétés applicables pour décrire ses instances (described_by), en plus des propriétés importées et héritées. Une classe peut définir un objet ou fournir une représentation correspondant à différents points de vue sur l'objet(*is_view_of*) qui sont eux mêmes des instances de classes.

L'entité *Property* est caractérisée par un identifiant (BSU), un domaine (*domain*) et un co-domaine (*range*) (ce dernier référence l'entité *Class_instance* pour les propriétés de types objet

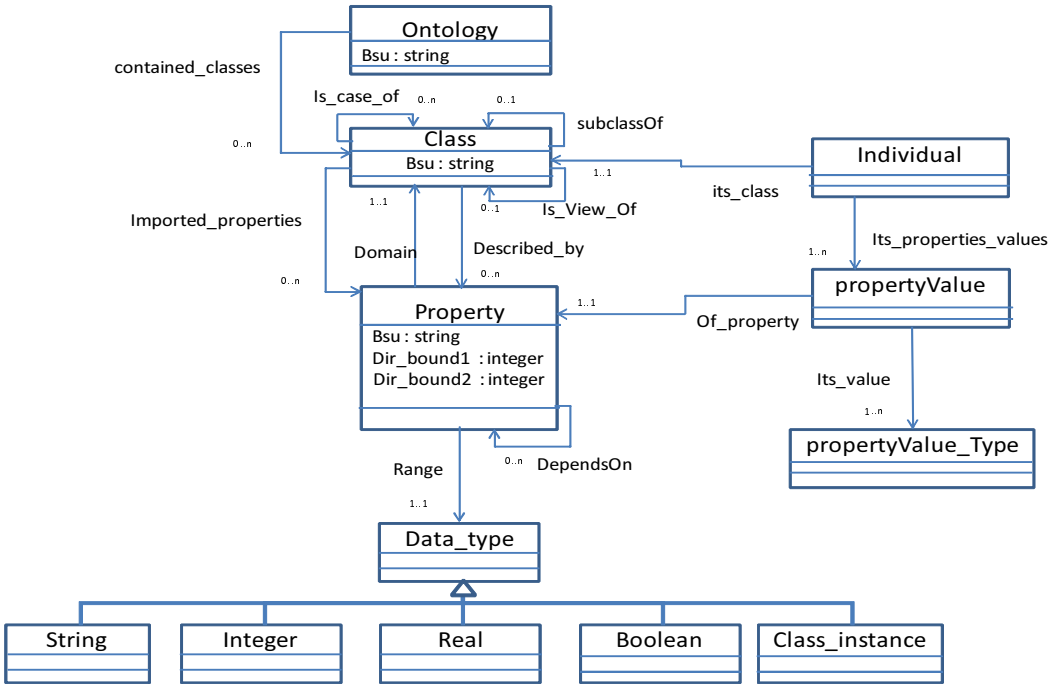


FIG. 3.1 – Représentation UML simplifiée du formalisme noyau d'OntoDB2

et, un des autres sous-types de l'entité *Data_ type* pour les propriétés simples ; cette contrainte n'est pas représentée dans le modèle. Une propriété peut dépendre (*depends_ on*) d'un ensemble d'autres propriétés. Une propriété (de type collection) peut également préciser, sa cardinalité minimale *dir_ bound1* et sa cardinalité maximale *dir_ bound2*.

L'entité *Indivudual* est reliée à l'entité *Class* par l'attribut *its_ class* permettant de définir l'appartenance d'une instance à une unique classe et l'attribut *its_ properties_ values* permet de spécifier pour toute chaque propriété utilisée (*of_ Property*) pour décrire l'instance, la valeur associée (*its_ value*). Pour une propriété de type objet, cette valeur sera un individu et pour une propriété de type simple, cette valeur sera une valeur simple. Cette contrainte, de même que les méta-descripteurs ne sont pas représentés sur ce schéma.

Nous allons à présent, nous intéresser à définir comment le formalisme noyau peut être enrichi par de nouvelles constructions. Nous allons nous intéresser tout particulièrement dans ce travail :

- au support des constructions non canoniques OWL Lite,
- au support de la représentation géométrique.

2.2 Enrichissement du formalisme noyau

Les ontologies conceptualisent les informations des domaines en termes de classes et de propriétés associées à des types de données. Ces différents concepts sont définis à partir des entités (constructeurs de classes et de propriétés), d'attributs et de types de données disponibles dans le formalisme d'ontologie. Ainsi, enrichir le formalisme d'ontologie noyau, suppose soit :

1. de définir de nouvelles entités ;

2. de modifier une entité existante ; c'est-à-dire d'ajouter un ou plusieurs attributs à une entité existante ;

Ainsi, ces nouvelles constructions (entité, attribut et type de données) pourront ensuite être utilisées pour représenter des nouvelles informations dans les ontologies.

La mise en œuvre de telles extensions est réalisée en étendant, au niveau du méta-schéma, la méta-représentation du formalisme d'ontologie décrit comme instance du méta-schéma. Ces extensions sont simplement décrites dans le langage de spécification EXPRESS. Des compilateurs et générateurs de code prennent alors en charge à la fois (1) la représentation des extensions au niveau du méta-schéma sous forme d'instances SQL pour permettre les traitements génériques, et (2) leur représentation au niveau des tables du formalisme d'ontologie pour permettre leur mise en œuvre.

Nous présentons au chapitre 4, la mise en œuvre de ces mécanismes. Nous présentons ici les premières extensions que nous avons réalisées de façon native.

Notons que la définition de nouvelles entités et l'extension du système de types sont principalement réalisées par sous-typage des entités existantes (*Class*, *Property*, *Data_ type*). Le choix du type d'extension à adopter pour une construction donnée se fait suivant le principe suivant :

- une nouvelle entité n'est introduite que si elle représente le domaine d'un nouvel attribut qui n'aurait pas de sens pour les entités du formalisme noyau. Par exemple, les attributs *spatial_ reference* (système de référence spatiale) et *spatial_ dimension* (dimension spatiale) n'ont de sens que pour le type de données géométrique. Ce type nécessite donc l'introduction d'une nouvelle entité.
- un nouvel attribut n'est introduit dans une entité existante que dans le cas où aucun attribut existant de l'entité ne peut jouer le même rôle. En effet, les différents formalismes d'ontologies peuvent nommer différemment des constructions qui jouent pourtant le même rôle ; c'est le cas par exemple pour l'identification des classes et des propriétés réalisé par les BSU dans PLIB et les URI dans OWL. Le même attribut peut donc être utilisé. Un nouvel attribut optionnel (de cardinalité, 0..1) pourrait être défini si son usage pourrait dépasser le cadre particulier d'un seul formalisme d'ontologie.

Nous présentons ci-dessous ces mécanismes sur les premières extensions du formalisme noyau que nous avons réalisé par certaines constructions orthogonales, en particulier celles du formalisme d'ontologie OWL Lite.

2.2.1 Extension par modification de l'entité *Class*

Nous présentons dans cette section des exemples d'extensions apportées à l'entité *Class* (cf. figure 3.2 [21]) pour lui permettre de représenter une classe OWL Lite. Lorsque l'on veut représenter une classe non canonique, il faut représenter la description ontologique de la classe et la représentation SQL de son contenu. Ces attributs pouvant avoir de l'intérêt pour d'autres classes, par exemple, la vue SQL peut être utile pour permettre de normaliser en plusieurs tables le contenu d'un classe canonique, nous avons décidé d'ajouter ces attributs à l'entité *Class*. De même un attribut booléen *is_ primitive* permet de spécifier si la classe est primitive ou non, et un attribut

[21] Les attributs de la figure 3.1 ne sont pas répétés.

de type chaîne de caractère *origin* permet de spécifier l'origine de la classe (dans notre cas, PLIB ou OWL).

2.2.1.1 Vues sur les données

Comme nous le verrons dans la section 4.3, OntoDB2 propose une solution originale pour la gestion des DBO non canoniques. En effet ces dernières sont gérées au travers de vues sur les données des classes canoniques représentées dans la BDBO OntoDB2. Cette solution nécessite de définir :

- pour une classe non canonique la vue ontologique qui représente les DBO qui lui sont associées. Cette vue va être définie à partir de la description formelle de la classe non canonique comme nous le verrons à la section 4.3.
- pour toute classe canonique ou non, la vue SQL qui représente l'extension de cette classe canonique. Cette vue est vide dans le cas où la classe (canonique ou non) ne contient aucune instance.

L'entité *class* doit donc être étendue avec (1) deux attributs booléens *has_ontological_view* et *has_sql_view* précisant la méthode d'accès aux DBO et, (2) les deux attributs de type chaînes de caractères *ontological_view* et *sql_view_name* définissant respectivement la vue ontologique et la vue SQL associée aux données.

2.2.1.2 Identification de classes OWL

Nous avons choisi de ne pas introduire d'attributs différents pour représenter chacun des identifiants des classes. Ainsi, notre formalisme noyau étant plus proche de PLIB, nous appelons cet identifiant BSU et les identifiants (URI) des ontologies OWL doivent être mappés sur les BSU dans la BDBO OntoDB2. Cependant, le BSU dans PLIB est obtenu par concaténation d'un certain nombre d'autres attributs (*source, code, version, ...*) comme nous l'avons précisé à la section 2.1.2.3 du chapitre 1. Nous avons choisi de représenter entièrement les URI dans la composante *code* du BSU sans les décomposer. Nous avons également introduit dans l'entité *Class*, un attribut de type chaîne de caractères BSU_or_ID. Cet attribut permet en particulier lors de l'exportation des ontologies, de retourner les identifiants des ontologies autres que PLIB sous leur format d'origine (sans réaliser la concaténation avec les autres attributs composant le BSU).

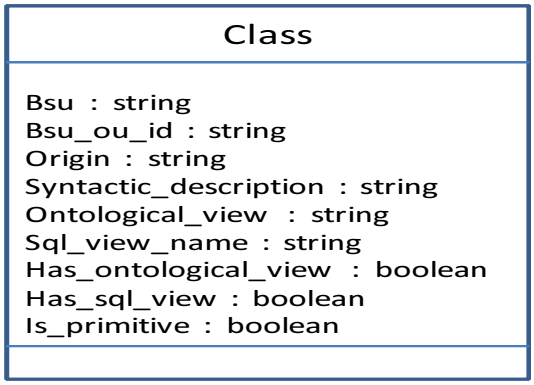


FIG. 3.2 – Extension de l'entité Class par ajout d'attributs

2.2.2 Extension par modification de l'entité *Property*

Nous présentons dans cette section des exemples d'extensions apportées à l'entité *Property* (cf. figure 3.3).

2.2.2.1 Identification des propriétés

Pour les mêmes raisons que pour l'entité l'entité *class*, nous introduisons dans l'entité *property*, un attribut de type chaîne de caractères BSU_OR_ID.

2.2.2.2 Caractéristiques de propriétés

Certaines caractéristiques prédéfinies peuvent être spécifiées pour les propriétés de type objet d' OWL Lite : l'inverse, la transitivité, la symétrie et la fonctionnalité inverse.

A ces caractéristiques, on peut ajouter les caractéristiques d'ordre et de propagation que nous avons identifiés dans le projet e-Wok Hub.

- La caractéristique d'ordre correspond à l'application simultanée des caractéristiques transitive, réflexive et antisymétrique sur la même propriété.
- La caractéristique de propagation correspond à l'association entre deux propriété dont l'une définie un ordre. Cette est une caractéristique importante par exemple dans le domaine de la médecine et le domaine de la géologie.
  La propagation définit la composition entre une propriété et une autre possédant la caractéristique d'être un ordre. Par exemple, du fait que la propriété de *seSubdiviseEn* entre les zones administratives définisse un ordre, le fait qu'un document soit géolocalisé par une zone incluse dans la fermeture transitive de la France, implique que ce même document est géolocalisé par la France. Nous approfondirons la mise en œuvre d'une telle extension au chapitre 5.

Ces différentes caractéristiques des propriétés que nous venons de définir ci-dessus sont importantes pour bon nombre d'applications. Étendre le formalisme d'ontologie noyau afin de pouvoir représenter ces caractéristiques nécessite l'ajout d'un ou de plusieurs nouveaux attributs à l'entité *Property* selon les cas.

- Les caractéristiques symétrique, transitive et inverse-fonctionnelle peuvent être capturées en utilisant un attribut prédéfini de type booléen. Ainsi, étendre le formalisme noyau pour représenter chacune des ces caractéristiques nécessite d'introduire dans l'entité *Property*, une attribut booléen pour chacune d'elles. la figure 3.3 montre le schéma résultant de l'entité *Property* intégrant les attributs de type booléen *is_symmetric, is_transitive* et *is_inversefunctional*.
- La représentation de la caractéristique inverse nécessite l'ajout d'un attribut (*inverseOf*) comme le montre la figure 3.3 qui permet de référencer la propriété inverse. Cette caractéristique est importante car, la théorie des bases de données définit des règles de passage automatiques qui permettent d'optimiser la structure des données en se basant sur la connaissance des cardinalités respectives de chaque classe impliquée dans la relation inverse. Par exemple, si l'on considère un modèle de données où sont définit deux classes : *Personne* et *Entreprise*, et deux propriétés : *dirige* et *estDirigéePar*. La propriété *dirige*

étant déclarée comme l'inverse de la propriété *estDirigeePar* ayant pour domaine la classe *Personne* et pour co-domaine la classe *Entreprise*. La connaissance de la participation de la classe *Personne*, domaine de la propriété (cardinalités inverses, 0..n) et la connaissance de la participation de la classe *Entreprise*, co-domaine de la propriété *dirige* (cardinalités directes, 1..1) permet de déduire que la représentation de la seule propriété *estDirigéPar* permet de représenter toute l'information nécessaire dans la base de données.

Dans un grand nombre de situations, les participations des classes dans une association sont connues bien que cette association ne définisse pas explicitement une inverse. Nous avons étendu l'entité *Property* avec les attributs *inv_ bound1, inv_ bound2* qui lorsque la propriété ne définit pas d'inverse explicite, permettent de représenter les participations de son co-domaine dans l'association lorsque celles-ci sont connues. Ces attributs seront utilisées non seulement pour optimiser le schéma de représentation des DBO dans une approche de représentation horizontale ou binaire par exemple, mais aussi par le langage de requête ontologique pour calculer les propriétés inverses à partir des valeurs de participation comme nous le verrons à la section 4.4.1.3. Notons que cette approche est aussi utilisée dans certains systèmes comme Hibernate ou encore JPA pour optimiser le schéma des bases de données qu'ils génèrent.

– Afin de capturer au niveau du formalisme d'ontologie la relation de propagation entre deux propriétés, on a besoin de trois informations. La première permettant de représenter le fait qu'une propriété définisse une relation d'ordre (*is_ order*), la seconde permettant de définir qu'une propriété est propagée en précisant la ou les références vers les propriétés ayant la caractéristique d'être un ordre qui la propagent (*propagatedBy*) et pour chacune d'elle, la direction de propagation par rapport à l'ordre (*direction*).

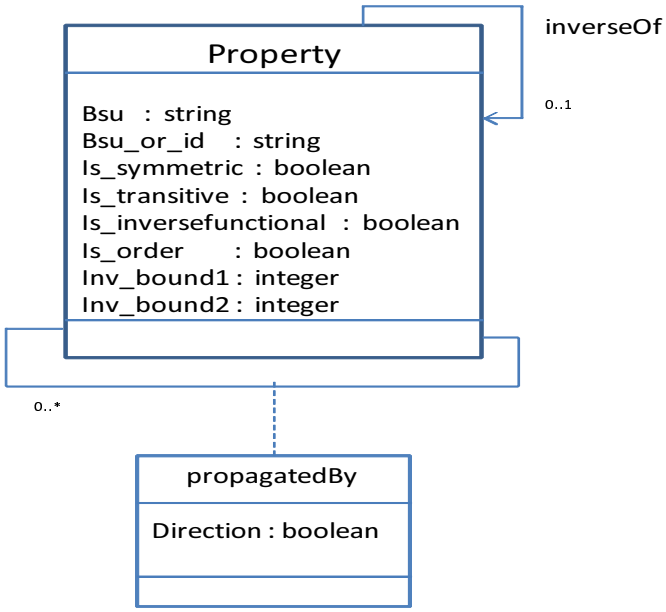


FIG. 3.3 – Extension de l'entité *Property* par ajout d'attributs

2.2.3 Flexibilité du système de types de données

Nous avons noté au chapitre 2, le besoin de pouvoir enrichir les types de données utilisés pour décrire le co-domaine des propriétés des ontologies. Nous avons à titre d'exemple, rencontré ce besoin pour représenter le système de données des Systèmes d'Information Géographique. Nous qualifions ces types de données, non prévus dans les formalismes d'ontologie, de données non standards.

Comme pour les classes et propriétés définies par le formalisme d'ontologie, l'existence d'un méta-schéma permet là aussi de définir de nouveaux types de données. Afin de montrer les capacités d'extension, nous présentons ci-dessous la structure de base offerte par OntoDB2 qui contient outre, les types de données simple usuels (caractères, entier, réel, booléen, etc.), des types de données plus complexes tels que les unités de mesure par exemple. La figure 3.4 montre l'extension de l'entité *Datatype* avec les mesures (*real_ measure*) ayant un attribut spécifiant l'unité de mesure (*unit*) et les géométries (*geometry, point, polygon, ...*) issues de la spécification standardisée des types géométriques définie par OpenGis [22]. Ces dernières sont caractérisées par leur dimension (*spatial_ dimension*), et le système de référence spatiale (*spatial_ reference*) à partir duquel elles sont définies.

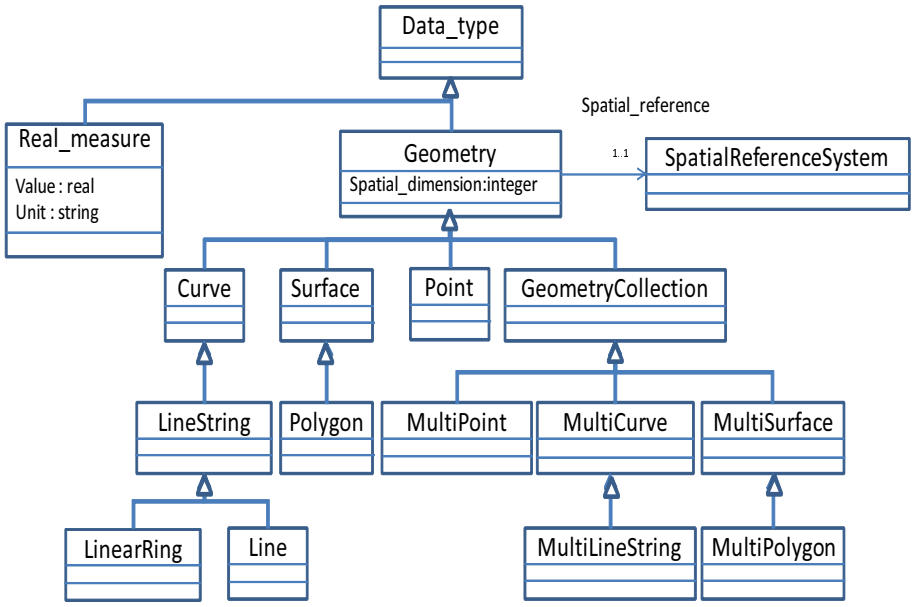


FIG. 3.4 – Extension de l'entité *Data_ type* du formalisme noyau d'OntoDB2

3 Synthèse sur le formalisme d'ontologie d'OntoDB2

Nous avons donc défini pour la BDBO OntoDB2, un formalisme d'ontologie flexible et extensible. Ce formalisme d'ontologie se base sur un noyau constitué de constructions communes aux formalismes d'ontologies PLIB et OWL Lite en respectant leurs spécificités auxquelles s'ajoutent

[22] http ://www.opengeospatial.org

toutes les constructions spécifiques PLIB. Ce noyau a déjà été étendu en ajoutant un certain nombre de constructions spécifiques d'OWL Lite permettant en particulier de représenter les classes non canoniques, des caractéristiques de propriétés ainsi que des entités géométriques mais qui semblaient d'intérêt général.

Par ailleurs le noyau peut encore être étendu par chaque utilisateur tant au niveau classes et propriétés qu'à celui des types de données accessibles en ajoutant de nouveaux attributs aux entités de base (*Class*, *Property*, *Data_ type*) ou en sous-typant ces dernières. Toutefois, les extensions apportées doivent rester compatibles avec les constructions du noyau (PLIB en particulier) ainsi que les extensions préexistantes.

Rappelons que lors de la spécification du formalisme noyau d'OntoDB2, nous avons choisi d'adopter :

Hypothèses de base d'OntoDB2

- l'héritage simple des classes ;
- le typage fort des propriétés ;
- la mono-instanciation ;
- l'usage de méta-descripteurs textuels.

Nous ne voulons pas pour l'instant, remettre en cause ces hypothèses qui sont des hypothèses de PLIB, et à l'origine de son aptitude à traiter de grands volumes d'instances avec un grand nombre de propriétés. Ce sont aussi les hypothèses usuelles dans le domaine des bases de données. Toute nouvelle extension doit donc conserver ces hypothèses aussi bien au niveau ontologique, qu'au niveau données.

Dans les applications que traite le laboratoire, les ontologies utilisées sont décrites suivant le formalisme PLIB. Toutefois, le laboratoire est aujourd'hui impliqué dans un certain nombre de projets transversaux dans lesquels les besoins de modélisation identifiés peuvent s'exprimer en terme de constructions non disponibles dans PLIB, il s'agit en particulier de constructions OWL Lite. Nous avons déjà présenté à la section 2.2 l'extension du formalisme d'OntoDB2 par certaines des constructions issues de ce formalisme. La table 3.1 présente pour l'ensemble des constructions [23] du formalisme OWL Lite, celles qui sont incompatibles ou compatibles avec nos hypothèses et dans ce dernier cas, nous précisons pour chaque construction si elle est mise en œuvre.

Cette table montre qu'un grand nombre de constructions OWL Lite sont compatibles avec le cadre d'hypothèses d'OntoDB2.

- Les constructeurs de classe *Class*, *intersectionOf* et *subclassOf* sont compatibles, à condition d'être utilisés dans un contexte où l'hypothèse d'héritage simple est respectée. Nous avons donc mis en œuvre ces constructions en tenant compte de nos hypothèses.
- La classe prédéfinie *owl :Nothing*, est par définition (1) sous-classe de toutes les classes de l'ontologie et (2) ne possède pas d'instance. Cette classe ne respecte pas l'hypothèse d'héritage simple ; nous l'avons donc pas représentée dans OntoDB2.

[23] source : http ://www.w3.org/TR/2004/REC-owl-features-20040210/#s3
[2] + : oui / - : non
[3] Compatibilité +* : partiellement compatible, certaines utilisations sont incompatibles avec nos d'hypothèses.
[4] Implémentation +* : avec des restrictions liées au cadre d'hypothèse.

Constructeur [234]	Compatibilité	Implementation
Ontologie		
Ontology	+	+
Imports	+	+
backwardCompatibleWith	+	-
inCompatibleWith	+	-
priorVersion	+	-
versionInfo +	-	
Classe		
Class	+*	+
subClassOf	+*	+*
IntersectionOf	+*	+*
DeprecatedClass	+	-
equivalentClass	+	+
owl :Thing	+*	+
owl :Nothing	-	
Restriction		
allValuesFrom	+	+
someValuesFrom	+	+
minCardinality	+	+
maxCardinality	+	+
cardinality	+	+
Propriété		
ObjectProperty	+	+
DatatypeProperty	+	+
TransitiveProperty	+	+
SymmetricProperty	+	+
FunctionalProperty	+	+
InverseFunctionalProperty	+	+
InverseOf	+	+
subPropertyOf	+	-
domain	+	+
range	+	+
DeprecatedProperty	+	-
equivalentProperty	+	-
AnnotationProperty	-	-
OntologyProperty	-	-
Instance		
Individual	+*	+
differentFrom	+	-
allDifferent	+	-
distinctMembers	+	-
sameAs	+*	-
Méta-descripteurs		
rdfs :label	+	+
rdfs :comment	+	+
rdfs :seeAlso	-	
rdfs :isDefinedBy	-	
Types de données		
xsd :datatypes	+	+

TAB. 3.1 – Compatibilité d'OWL Lite avec OntoDB2

- Le constructeur *individual* doit être utilisé en respectant l'hypothèse de mono-instanciation.
- Les méta-descripteurs tels que rdfs :seeAlso, rdfs :isDefinedBy peuvent avoir pour objet des individus, or, nous avons choisi de ne prendre en compte que des méta-descripteurs textuels dans OntoDB2. En conséquence, ces constructions sont incompatibles avec nos choix.
- La spécification de formalisme OWL Lite stipule que les propriétés d'annotation définies par les constructeurs *AnnotationProperty* et *OntologyProperty* ne peuvent pas être associées à un domaine ou à un co-domaine spécifique. Ce sont des propriétés qui ne sont donc pas typées, elles sont donc est incompatible avec l'hypothèse de typage fort que nous avons adopté pour le formalisme noyau.

En ce qui concerne les autres constructions que nous avons implémentées et celles que nous envisageons d'implémenter, elles doivent également être utilisées dans un contexte qui soit compatible avec nos hypothèses.

Nous allons dans la suite, définir comment vont être gérées, dans OntoDB2, les DBO canoniques et non canoniques appartenant aux classes des ontologies définies suivant les constructions du formalisme d'ontologie noyau éventuellement étendu par d'autres constructions.

4 Gestion des DBO canoniques et non canoniques de grande taille

Comme nous l'avons discuté au chapitre 2, la problématique de la gestion des DBO de grande taille dans les BDBO réside (1) dans la représentation des données canoniques et non canoniques et (2) dans l'interrogation des DBO canoniques et non canoniques qui nécessite soit de faire appels à des raisonneurs exploitant pour la plupart des mécanismes non disponibles dans les bases de données traditionnelles, soit de saturer les relations ce qui induit de la redondance et des difficultés de mise à jour.

Afin d'assurer dans OntoDB2 une gestion efficace des DBO associées aux ontologies définies suivant le formalisme noyau étendu par les constructeurs d'équivalence conceptuelles tels ceux de OWL Lite, nous proposons de mettre en œuvre une stratégie originale consistant à

1. calculer la représentation canonique de toutes les instances non canonique pour lesquelles cela est possible,
2. accéder par une vue SQL à chacune des classes non canoniques.

Concernant le premier problème, il convient de noter que toute DBO non canonique ne peut pas forcément se convertir en une DBO canonique, tout simplement parce que cette information n'est pas connue. Si par exemple A et B sont deux classes canoniques et une instance est définie comme appartenant à A∪B ; sans autre information sur cette instance, il est impossible de savoir à quelle classe canonique elle appartient. Inversement, si l'ontologie définie comme classe primitive la classe *Personne[id : integer, genre :{'F', 'M'}, nom : String]* et comme classe définie *Femme = Personne[genre = 'F']*, l'instance *Femme[id = 456789, nom ='Durant']* sera automatiquement convertie lors du chargement de la base de données en *Personne[id = 456789, genre ='F', nom ='Durant']*.

Par contre si nous nous limitons à OWL Lite avec le cadre d'hypothèses définies comme hypothèses de base d'OntoDB2, il est toujours possible de transformer les instances non canoniques en instances canoniques. C'est ce que nous montrons dans ce qui suit.

Concernant le deuxième problème qui est celui de la représentation et de l'interrogation des données non canoniques, nous proposons d'exploiter les mécanismes usuels fournis par les bases de données c'est à dire le mécanisme de vue. Ainsi la traduction des descriptions formelles des classes non canoniques par des vues va permettre à la BDBO de calculer l'extension des classes non canoniques à partir de celle des classes canoniques.

Concernant les autres problèmes de raisonnement de OWL Lite (par exemple les caractéristiques de propriété), nous proposons également d'exploiter les capacités des BDBO, pour réaliser, lorsque cela est possible, les traitements déductifs.

Nous présentons dans la section 4.1, la représentation des données canoniques au sein d'OntoDB2. Ensuite, nous discutons dans les sections 4.2 et 4.3 respectivement la représentation et de l'interrogation des données non canoniques, enfin la section 4.4 présente la solution que nous proposons pour indexer certains traitements déductifs.

4.1 Représentation de l'information canonique

Trois principales approches de représentation peuvent être adoptées pour la représentation des DBO dans les BDBO. Nous avons vu au chapitre précédent que l'approche binaire et l'approche horizontale présentent de meilleures performances que l'approche verticale. Cependant, l'efficacité à la fois de l'approche binaire et de l'approche horizontale dépend fortement du type de données manipulées et du type d'interrogation auquel vise à répondre l'application à mettre en œuvre. La représentation binaire est ainsi plus adaptée lorsque les individus sont décrits par des propriétés différentes alors que l'approche horizontale au contraire, est mieux adaptée lorsque les individus sont décrits par un certain nombre de propriétés communes et, que les requêtes portent également sur plusieurs propriétés [20]. Il est donc approprié de permettre pour chaque application le choix de l'approche de représentation entre l'approche binaire et l'approche horizontale.

Nous considérons cependant dans la suite que les données sont représentées suivant une approche horizontale. Une telle hypothèse est faite d'une part pour simplifier l'exposé, d'autre part parce que c'est la seule approche complètement validée. Notre architecture permet néanmoins, si l'application le nécessite d'utiliser l'approche binaire.

4.2 Représentation de l'information non canonique

Les classes non canoniques sont définies en spécifiant des conditions nécessaires et suffisantes d'appartenance, de telle sorte qu'il est possible de vérifier ces conditions sur tout individu pour affirmer si oui ou non, il est une instance d'un concept non canonique donné. Ce constat nous a amené à conclure que, du fait que les classes non canoniques sont définies à partir de classes canoniques, leurs instances devraient pouvoir être représentées dans la BDBO comme des instances de classes canoniques (migration d'instances). Ceci n'est évidemment vrai que si la donnée non canonique est « complète » c'est à dire si elle contient sa classe d'appartenance de base qui peut être non canonique.

Inversement, si la migration de toutes les instances non canoniques est possible, l'ensemble des instances de ces classes non canoniques peuvent alors être accédées par une vue sur les classes canoniques sur lesquelles elles sont définies.

Représenter les DBO non canoniques dans la BDBO comme des instances de concepts canoniques, nécessite de définir des règles de transformation des DBO non canoniques en DBO canoniques. Ces règles de transformation vont se baser sur la description formelle des concepts non canoniques. Nous vérifions ci-dessous que dans le cadre de nos hypothèses, cela est vrai pour toute population OWL Lite qui les respecte.

Avant de présenter l'approche de migration d'instance que nous proposons, nous rappelons d'abord comment sont modélisées en OWL les classes canoniques et les classes non canoniques.

4.2.1 Classe canonique / Classe non canonique

Une classe canonique est une classe pour laquelle la définition axiomatique est partielle ; c'est à dire que seules des conditions nécessaires d'appartenance sont définies. Pour une classe non canonique, la définition axiomatique est complète ; c'est à dire que les conditions nécessaires et suffisantes d'appartenant des individus sont définies.

En OWL, une définition axiomatique partielle est exprimée par utilisation de l'axiome *subclassOf* ($\subseteq$) et, définition axiomatique complète est exprimée par l'utilisation de l'axiome *equivalentClass* ($\equiv$).
Par exemple, étant données la propriété P et les classes A, B et C, la construction :

- $A \sqsubseteq \forall\ P.\ C$, définit la classe A comme étant une classe canonique alors que,
- $B \equiv \forall\ P.\ C$, définit la classe B comme étant une classe non canonique.

Nous présentons ci-dessous, notre approche de migration d'instances pour les ontologies OWL Lite. Nous considérons pour cela les propriétés P et Q et les classes A, B et C. La fonction *Dom* permet de retourner la classe domaine de la propriété en paramètre.

4.2.2 Classe définie comme une restriction

Le formalisme d'ontologie OWL Lite comporte deux catégories de restrictions :

- les restrictions de co-domaine qui redéfinissent localement le co-domaine d'une propriété
- les restrictions de cardinalité qui définissent localement les cardinalités minimales et maximale pour une propriété multivaluée ;

Considérons par exemple la description $A \equiv \forall\ P.\ C$ qui définit la classe A comme étant équivalente à l'ensemble des individus dont toutes les valeurs sont de type C pour la propriété P. Étant donné un individu d'une telle classe, et en tenant compte du fait que dans OntoDB2, (1) toute propriété possède un unique domaine (hypothèse de typage fort des propriétés) et, (2) toute instance est décrite avec les propriétés applicables à sa classe de base (hypothèse typage fort des instances) ; nous pouvons déduire de la description ci-dessus que les individus appartenant à la classe A appartiennent également à la classe domaine de la propriété P. Les instances de la classe A seront donc stockées dans OntoDB2, comme des instances du domaine de la propriété *Dom(P)*. Un raisonnement similaire va être appliqué pour les autres restrictions.

4.2.3 Classe définie comme une intersection

Dans OWL Lite les membres d'une intersection sont soient des classes nommées, soient des restrictions. Trois cas de figure principaux peuvent dont être rencontrés. Nous présentons ci-dessous, ces différents cas.

4.2.3.1 Intersection de deux classes nommées

Considérons par exemple la description : $A \equiv B \cap C$ qui définit la classe A comme étant équivalente à l'intersection des classes B et C.

1. Si $B \nsubseteq C$ (respectivement $C \nsubseteq B$), alors les instances de la classe A ne respecteront pas l'hypothèse de mono-instanciation.
2. Si $B \subseteq C$ (respectivement $C \subseteq B$), alors la classe A est équivalente à la classe B (respectivement A). Ces instances seront donc converties comme des instances de la classe B (respectivement C).

4.2.3.2 Intersection d'une classe nommée et d'une restriction

Considérons par exemple la description : $A \equiv B \cap \forall\ P.\ C$ qui définit la classe A comme étant équivalente à l'intersection de la classe B et de l'ensemble des individus dont toutes les valeurs sont de type C pour la propriété P.

1. Si $B \nsubseteq$ *Dom(P)* (respectivement *Dom(P)* $\nsubseteq B$), alors les instances de la classe A ne respecteront pas l'hypothèse de mono-instanciation.
2. Si $B \subseteq$ *Dom(P)* (respectivement *Dom(P)* $\subseteq B$), alors les instances de la classe A seront converties en instances de la classe B (respectivement *Dom(P)*).

4.2.3.3 Intersection de deux restrictions

Considérons par exemple la description : $A \equiv \exists\ Q.\ B \cap \forall\ P.\ C$ qui définit la classe A comme étant équivalente à l'intersection de l'ensemble des individus pour lesquels, ils existent des valeurs de type B pour la propriété Q et, de l'ensemble des individus dont toutes les valeurs sont de type C pour la propriété P.

1. Si *Dom(Q)* $\nsubseteq$ *Dom(P)* (respectivement *Dom(P)* $\nsubseteq$ *Dom(Q)*), alors les instances de la classe A ne respecteront pas l'hypothèse de mono-instanciation ;
2. Si *Dom(Q)* $\subseteq$ *Dom(P)* (respectivement *Dom(P)* $\subseteq$ *Dom(Q)*), alors les instances de la classe A seront converties en instances de la classe *Dom(Q)* (respectivement *Dom(P)*).

4.3 Accès aux DBO non canoniques

Une fois les DBO non canoniques transformées en DBO canoniques et insérées dans la base de données, notre objectif est que les données non canoniques puissent être accédées comme si elles étaient représentées. OntoDB2 doit donc fournir des mécanismes pour déduire l'extension des concepts non canoniques à partir des extensions des concepts canoniques qui sont représentés dans la BDBO. Comme pour les constructeurs d'équivalences conceptuelles, il est possible dans les bases de données d'utiliser le langage de requête SQL pour créer des vues permettant de définir le contenu de tables virtuelles, fondées sur les descriptions de tables pré-existantes. Nous

présentons ci-dessous comment les vues sont utilisées dans OntoDB2 pour calculer l'extension des concepts non canoniques. Nous proposons de définir ces vues SQL dès que l'ontologie est lue et de les référencer dans l'ontologie au sein de la BDBO OntoDB2.

Nous présentons dans cette section comment les vues ontologiques sont définies à partir de la définition formelle des concepts non canoniques. Dans les exemples ci-dessous, la fonction ϑ renvoi la vue associée à la classe ou à la propriété passée en paramètre.

4.3.1 Classe définie comme une restriction

Différents cas de figure peuvent être rencontrés pour les restrictions.

4.3.1.1 Restriction de cardinalité

Soit la description $A \equiv = P.\ n$
Cette description définit une classe non canonique A comme étant composée de l'ensemble des individus pour lesquels la propriété P possède exactement n valeurs. Afin de définir la vue correspondant à cette description, nous allons employer la primitive SQL COUNT, qui permet calculer le nombre de valeurs d'un sous-ensemble donné :

Vue SQL d'une restriction de cardinalité

```
SELECT cls.rid
FROM ϑ(Dom(P))[5] AS cls
WHERE  cls.rid IN
            ( SELECT  p.rid
              FROM  ϑ(P) AS prop
              GROUP BY  prop.rid
              HAVING COUNT (prop.P) =n )
```

Cette vue retourne l'identifiant (*rid*) de tous les individus de la classe *Dom(P)* dont le nombre de valeurs pour la propriété P est égal à n.

Dans le cas des restrictions portant sur la cardinalité minimale et la cardinalité maximale d'une propriété, il suffit de remplacer l'opérateur d'égalité ($=$) dans la clause SQL HAVING respectivement par l'opérateur d'infériorité ($\geq$) et par l'opérateur de supériorité ($\leq$).

`Exemple :` *Bilingue* $\equiv =$ *languesParlées 2*
L'exemple ci-dessous définit une classe non canonique *Bilingue* comme étant composées de l'ensemble des individus pour lesquels la propriété *languesParlées* possède exactement deux valeurs. La vue correspondante à la classe *Bilingue* est :

[5]Nous utilisons ϑ(Personne) car, comme nous l'avons expliquer dans la section 4.1, les contenus des classes canoniques sont aussi représentées par des vues.

```
SELECT cls.rid
FROM ϑ(Personne)AS cls
WHERE  cls.rid IN
            ( SELECT  prop.rid
              FROM  ϑ(languesParlées) AS prop
              GROUP BY  prop.rid
              HAVING COUNT (prop.languesParlées) =2 )
```

4.3.1.2 Restriction de co-domaine

Soit la description $A \equiv \forall\ P.\ C$

Cette description permet de définir une classe A dont tous les individus doivent avoir pour la propriété P, des valeurs qui sont uniquement de type C. Il s'agit d'une restriction universelle. Dans ce cas ci également, nous utilisons la primitive SQL COUNT. La vue suivante permet de calculer l'extension d'une telle classe.

Vue SQL d'une restriction universelle

```
SELECT cls.rid
FROM ϑ(Dom(P)) AS cls, ϑ(P) AS prop
WHERE (cls.rid = prop.rid)
GROUP BY (cls.rid)
HAVING COUNT (cls.rid) = (
                          SELECT COUNT(*)
                          FROM ϑ(P) AS prop2
                          WHERE prop2.P IN (
                                            SELECT rid
                                             FROM ϑ(C))
                          AND (prop2.rid = cls.rid)) ;
```

Cette vue retourne l'identifiant (*rid*) de tous les individus de la classe *Dom(P)* dont le nombre de valeur de type C pour la propriété P, correspond exactement au nombre total de valeurs de P pour chaque individu.

`Example :` *Romancier* $\equiv$ *ALL aEcrit. Roman*

L'exemple ci-dessus permet de définir une classe *Romancier* dont tous les individus doivent avoir pour la propriété *aEcrit*, des valeurs qui sont uniquement de type *Roman* (on suppose ici que que la propriété *aEcrit* a pour domaine *Personne*). La vue correspondante à la classe *Romancier* est :

```
SELECT cls.rid
FROM ϑ(Personne) AS cls, ϑ(aEcrit) AS prop
WHERE (cls.rid = prop.rid)
GROUP BY (cls.rid)
HAVING COUNT (cls.rid) = (
                          SELECT COUNT(*)
                          FROM ϑ(aEcrit) AS prop2
                          WHERE prop2.aEcrit IN (
                                                 SELECT rid
                                                 FROM ϑ(Roman))
                          AND (prop2.rid = cls.rid)) ;
```

Dans le cas d'une restriction existentielle : $A \equiv \exists\ P.\ C$
La vue correspondante sera plutôt définie comme suit :

Vue SQL d'une restriction existentielle

```
SELECT cls.rid
FROM ϑ(Dom(P)) AS cls, ϑ(P) AS prop
WHERE (cls.rid = prop.rid)
      AND (prop.P) IN (
                       SELECT rid
                       FROM ϑ(C))
```

Cette vue, permet de retourner tous les individus de la classe *Dom(P)* ayant une valeur pour la propriété P de type C ; ce qui correspond bien à la description de la classe *Scientifique*.

Notons que étant donnée que dans OWL Lite, les restrictions de cardinalité ne porte que sur les valeurs 0 ou 1, la vue correspondante à une restriction de cardinalité minimale 1, peut s'écrire de la même manière que la vue correspondante à une restriction existentielle. Inversement, une restriction existentielle peut être interprétée comme une restriction de cardinalité minimale 1.

`Exemple :` *Scientifique* $\equiv \exists$ *aEcrit articleScientifique*
La vue correspondante est :

```
SELECT cls.rid
FROM ϑ(Personne) AS cls, ϑ(aEcrit) AS prop
WHERE (cls.rid = prop.rid)
      AND (prop.aEcrit) IN (
                            SELECT rid
                            FROM ϑ(articleScientifique))
```

4.3.2 Classe définie comme une intersection

Comme nous l'avons vus à la section 4.2.3, et compte tenu du cadre d'hypothèses OntoDB2, les intersections de classes ne sont pas toutes supportées. Nous présentons ici, comment sont calculés les instances des classes intersection pour les trois cas de figure que nous avons vu à la section 4.2.3. Pour chaque cas, nous ne présentons ici qu'un exemple de configuration compatible avec OntoDB2.

4.3.2.1 Intersection de deux classes nommées

Soit la description : $A \equiv B \cap C$, avec $B \subseteq C$.
La classe A est équivalente à la classe B. La vue ci-dessous servira donc à calculer l'ensemble des instances de la classe A.

Vue SQL d'une intersection de deux classes nommées

```
SELECT cls.rid
FROM ϑ(B) AS cls
```

4.3.2.2 Intersection d'une classe nommée et d'une restriction

Soit la description : $A \equiv B \cap \forall\ P\ C$, avec $B \subseteq$ *Dom(P)*.
Les instances de la classe A ayant été converties (migrées) en des instances de la classe B, la vue associée à la classe A va être calculée dans ce cas, comme pour les restrictions de co-domaine présentées à la section 4.3.1.2. De plus, du fait que $B \subseteq$ *Dom(P)*, cette vue (de la restriction) sera basée sur la classe B et non sur *Dom(P)*.

Vue SQL d'une intersection d'une classe nommée et d'une restriction

```
SELECT cls.rid
FROM ϑ(B) AS cls, ϑ(P) AS prop
WHERE (cls.rid = prop.rid)
GROUP BY (cls.rid)
HAVING COUNT (cls.rid) = (
                          SELECT COUNT(*)
                          FROM ϑ(P) AS prop2
                          WHERE prop2.P IN (
                                            SELECT rid
                                             FROM ϑ(C))
                          AND (prop2.rid = cls.rid)) ;
```

4.3.2.3 Intersection de deux restrictions

Considérons par exemple, la description : $A \equiv \exists\ Q\ B \cap \forall\ P\ C$, avec *Dom(Q)* $\subseteq$ *Dom(P)*.
Les instances de la classe A ayant été converties (migrées) en des instances de la classe *Dom(Q)*, la vue associée à la classe A va être calculée dans ce cas, comme l'intersection des vues associées à chaque restriction comme présenté à la section 4.3.1.2. De plus, du fait que *Dom(Q)* $\subseteq$ *Dom(P)*, la vue de chaque restriction sera basée sur la classe *Dom(Q)*.

Vue SQL d'une intersection d'une classe nommée et d'une restriction

```
SELECT cls.rid
FROM ϑ(Dom(Q)) AS cls, ϑ(Q) AS prop
WHERE (cls.rid = prop.rid)
      AND (prop.P) IN (
                  SELECT rid
                  FROM ϑ(B))
INTERSECT
SELECT cls.rid
FROM ϑ(Dom(Q)) AS cls, ϑ(P) AS prop
WHERE (cls.rid = prop.rid)
GROUP BY (cls.rid)
HAVING COUNT (cls.rid) = (
                          SELECT COUNT(*)
                          FROM ϑ(P) AS prop2
                          WHERE prop2.P IN (
                                      SELECT rid
                                       FROM ϑ(C))
                          AND (prop2.rid = cls.rid)) ;
```

Les exemples que nous venons de présenter, montrent comment dans OntoDB2, les mécanismes de bases de données peuvent être exploités pour permettre l'accès aux DBO. Nous allons à présent voir comment ces mêmes mécanismes peuvent être utilisés pour répondre aux interrogations.

4.4 Autres mécanismes de raisonnements sur les DBO

Dans les DBO OWL Lite, les raisonnements sont souvent reliés aux constructeurs d'équivalence conceptuelle (les classes définies), aux caractéristique de propriétés, et aux instances.

Nous ne traitons pas le niveau instance à la manière de OWL Lite car nous le traitons par le mécanisme de clé (caractéristique inverse fonctionnelle) : deux instances de même clé sont identiques, deux instances de clés différentes sont différentes [70]. En ce qui concerne les classes définies, nous avons vu dans la section précédente comment les vues permettent de traiter ce problème. Nous nous intéressons dans cette section aux caractéristiques de propriétés pour lesquels nous montrons comment les raisonnements correspondant peuvent être réalisés au sein de la BDBO OntoDB2 en utilisant les capacités des bases de données. Nous présentons dans la section 4.4.1

comment les caractéristiques de propriété OWL Lite peuvent être gérées par utilisation des langages procéduraux des bases de données traditionnelles et par l'utilisation de l'interpréteur de requêtes ontologique. Ensuite nous montrons dans la section 4.4.2 comment les caractéristiques d'ordre et de propagation que nous avons identifié dans le projet e-Wok Hub peuvent être gérées en utilisant les mécanismes d'indexation des bases de données traditionnelles et l'interpréteur de requêtes ontologique.

4.4.1 Traitements des caractéristiques de propriétés OWL Lite

Certaines propriétés possèdent des caractéristiques (symétrique, transitive) qui contraignent l'extension de ces propriétés : elles permettent de dériver à partir d'un sous-ensemble de liens, l'ensemble des liens existants. Les systèmes de gestion de bases de données disposent de langages procéduraux qui peuvent être utilisés pour définir si besoin est, des fonctions (ou triggers) permettant de dériver l'extension de propriété possédant certaines caractéristique prédéfinies.

4.4.1.1 Symétrie

La caractéristique symétrique permet de déduire pour une instance donnée i_1 reliée à une autre instance i_2 par une propriété symétrique P, que cette dernière est également reliée à i_1 par la propriété P. Un trigger peut donc être utilisé pour mettre à jour l'instance i_2 suite à l'insertion ou à la mise à jour de la propriété P pour l'instance i_1.

4.4.1.2 Transitivité

La caractéristique transitive permet de déduire pour une instance donnée i_1 reliée à une autre instance i_2 par une propriété transitive P, que s'il existe une instance i_0 tel que i_0 soit reliée à i_1 par P, alors i_0 est également reliée à i_2 par la propriété P. De même s'il existe une instance i_3 tel que i_2 soit reliée à i_3 par P, alors i_1 est également reliée à i_3 par la propriété P. Le même raisonnement va ainsi être répété sur l'ensemble des instances jusqu'a ce qu'aucune nouvelle relation ne puissent être déduite.

Un tel comportement peut être mis en œuvre dans la BDBO en définissant un trigger. Cependant, le calcul de la fermeture transitive d'une relation transitive peut s'avérer coûteux car il nécessite des appels récursifs. Afin d'optimiser le calcul de la fermeture transitive pour une propriété transitive, nous faisons en sorte que l'extension de toute relation transitive soit toujours dans un état saturé dans la BDBO. Il est ainsi possible, à partir de cette hypothèse, de calculer la fermeture transitive de manière non récursive lors de toute nouvelle insertion. Le trigger correspondant peut alors être exprimé comme suit :

A chaque insertion d'une relation (i_1, i_2) dans $\vartheta(P)$:
1. $\forall\ (i_0, i_1) \in \vartheta(P)$
   Ajouter la relation (i_0, i_2) si celle-ci n'existe pas déjà
2. $\forall\ (i_2, i_3) \in \vartheta(P)$
   Ajouter la relation (i_1, i_3) si celle-ci n'existe pas déjà

Ce trigger permet d'éviter le traitement répétitif dans le calcul de la fermeture transitive. A l'insertion d'une relation (i_1, i_2) entre les individus i_1 et i_2 dans la vue associée à la propriété

transitive P, ce trigger rajoute dans cette même vue, tous les nouveaux couples *(i_0, i_2)* et *(i_1, i_3)* calculés si ces derniers n'existaient pas déjà.

Nous envisageons pour toute propriété transitive, d'utiliser une booléen pour spécifier pour tout individu, si sa valeur pour la propriété transitive a été calculée ou non par le système. Ceci devrait nous permettre, lors des mises à jour ou des suppressions, de re-saturer l'extension de la propriété à partir des liens initiaux.

4.4.1.3 Propriété inverse : accès par l'interpréteur de requêtes ontologiques

Une autre manière de répondre à une requête portant sur des informations non canoniques consiste à utiliser l'interpréteur de requête pour transformer la requête et l'exprimer en terme de données canoniques. Le langage de requêtes que nous utilisons, qui s'adresse bien au niveau ontologique, s'appelle OntoQL [36]. Son interpréteur est en cours de modification pour lui permettre, autant que de besoin, de transformer les requêtes adressées en termes de concepts non canoniques. Ceci est, en particulier, utilisé pour traiter les relations entre propriété inverse et relation directe.

Les valeurs des propriétés inverses sont redondantes par rapport aux propriétés directes qui leur correspondent. Elles ne sont pas stockées dans la BDBO OntoDB2, En effet, toute propriété inverse peut être déduite à partir de la représentation la propriété s'appliquant à son co-domaine et dont elle est inverse. Pratiquement, la représentation d'une propriété directe et de son inverse est choisie d'après les cardinalités directes et inverses. Une seule représentation suffit toujours pour les deux.

Afin de déterminer la valeur d'une propriété inverse, il est nécessaire de connaître la propriété s'appliquant à son co-domaine et dont elle est inverse. En effet, c'est cette dernière qui est utilisée pour sauvegarder la représentation en mémoire dans la base de données. Comme nous l'avons déjà précisé, cette information est accessible au travers de l'attributs *inverseOf* de l'entité *Property*. Il est donc nécessaire d'interroger le niveau ontologique pour calculer les propriétés inverses. Ce travail sera réalisé par l'interpréteur de requête ontologique, qui va exploiter les informations de niveau ontologique (participations : *inv_bound1, inv_bound2, dir_bound1 et dir_bound2*) afin de définir la vue à réaliser suivant les cas. Si on considère par exemple la figure 3.5 qui montre le

Vue_Personne		
ID	**nomPrénom**	**aEcrit**
pers#01	Phillipe	Doc#01
pers#02	Jules	
Pers#03	Paul	
...	...	...

Vue_Entreprise		
ID	**Nom**	**estDirigéPar**
ent#01	Mambo	pers#01
ent#02	ET sarl	pers#01
ent#03	Chic & Co	pers#03
...		...

FIG. 3.5 – Exemple de traitement de propriété inverse

schéma associé à l'exemple de la section 2.2.2.2 où les classes *Personne* et *Entreprise* sont reliées par deux propriétés *estDirigéPar* et *dirige* ; cette dernière étant déclarée comme l'inverse de la propriété *estDirigéPar*. Cette figure présente deux vues : une vue associée à la classe *Personne* et une vue associée à la classe *Entreprise*. Dans cette dernière vue, la colonne *estDirigéePar* définit pour une entreprise donnée, l'instance de la classe *Personne* qui la dirige. La propriété *dirige* étant

définie au niveau ontologique comme l'inverse de la propriété *estDirigéePar* avec (*inv_bound1, inv_bound2, dir_bound1 et dir_bound2*) = *(0, N, 1, 1)*, l'interpréteur de requête ontologique sera capable de définir, à partir de ces informations, la vue ci-dessus qui renvoie l'extension de la propriété *dirige*.

```
SELECT estDirigéPar as rid, id as dirige
FROM ϑ(Entreprise)
```

Nous verrons en détails dans le chapitre suivant, les différents autres cas de figures qui peuvent se présenter suivant les valeurs prises par les attributs *inv_bound1, inv_bound2, dir_bound1 et dir_bound2*.

4.4.2 Traitements effectués par un mécanisme d'indexation

La dernière catégorie de traitements que nous avons identifiée porte sur les traitements déductifs que l'on peut remplacer par des raisonnements numériques ou alphanumériques à travers des techniques d'indexation. Nous nous sommes intéressés à titre d'exemple à la propagation d'une relation par une relation d'ordre. Le comportement induit peut être indexé dans la BDBO OntoDB2 en le pré-calculant à l'aide de propriétés numériques ou alphanumériques. L'interpréteur de requête ontologique pourra ainsi lors des interrogations portant sur ces caractéristiques, substituer les requêtes initiales par les requêtes portant sur les propriétés d'index.

Pour indexer la relation d'ordre nous proposons, dans OntoDB2, de généraliser l'approche adoptée au niveau ontologique pour la relation de subsumption. Notre solution consiste donc à substituer la relation d'ordre par l'utilisation d'indexes définis par les techniques d'étiquetages et permettre ainsi de raisonner sur les relations d'ordre dans la BDBO en utilisant des requêtes numériques ou alphanumériques. Le langage de requête ontologique doit donc, pour toute requête portant sur une propriété ayant la caractéristique d'être une relation d'ordre, réécrire cette requête de manière à utiliser les colonnes d'étiquettes pour calculer l'extension de la relation d'ordre. Cette transformation est décrite en détails au chapitre 5.

Concernant la caractéristique de propagation, qui permet de composer une propriété P avec une autre propriété ayant la caractéristique d'être un ordre. Elle définit un comportement tel que, la fermeture transitive de la relation d'ordre propage la propriété P. La fermeture transitive de la propriété propagée P est donc déterminée par celle de la relation d'ordre. La technique d'étiquetage associée à la propriété d'ordre peut donc également être utilisée pour pré-calculer la fermeture transitive de la relation propagée. Cette solution est également décrite en détail au chapitre 5.

4.5 Synthèse sur le support des DBO

Le support des DBO au sein d'OntoDB2 possède deux caractéristiques essentielles.

1. Les données sont représentées sans redondance, sous forme de données canoniques ; toutes les données non canoniques sont converties en données canoniques.
2. L'interpréteur de requête de la base de données est utilisé pour les mécanismes d'accès fondamentaux : requête sur les données canoniques, requêtes sur l'ensemble des données, canoniques et non canoniques et l'accès aux propriétés inverses.

Ces deux caractéristiques garantissent les capacités du système à monter en charge de façon efficace puisque ce sont les mécanismes utilisés dans les bases de données dont les capacités de passer à grande échelle sont bien connues. Il est vrai que notre approche ne permet pas de représenter les données incomplètes, c'est à dire en particulier celles dont la classe de base et/ou la clé n'est pas connue. Mais la comparaison des formalismes d'ontologie montre qu'aucune ontologie n'est « complète »au sens des besoins, puisque chaque formalisme comporte des constructeurs non présents dans les autres formalismes.

Conclusion

Nous avons présenté dans ce chapitre la description de l'architecture de BDBO OntoDB2 que nous proposons. Cette spécification est basée sur l'étude des différents formalismes d'ontologies et des BDBO que nous avons étudiés aux chapitres 1 et 2. Cette étude nous a permis de faire les constats suivants.

- Tous les formalismes d'ontologies permettent de définir des ontologies à l'aide de constructeurs primitifs de classes et de propriétés et de leur associer des identifiants universels permettant de les référencer de l'extérieur. Ils permettent aussi de définir des types de données.
- Les constructions spécifiques offertes par les formalismes d'ontologies visent en particulier à définir des équivalences conceptuelles à partir des concepts canoniques. Les constructeurs d'équivalences conceptuelles sont pour la plupart orthogonaux. Il est ainsi possible de les combiner au sein d'une même architecture sans conflit.
- Les instances de classes définies par les constructeurs d'équivalence conceptuelle peuvent être représentées comme des instances de classes canoniques à condition d'être complètes,

En se basant sur ces constats, nous avons défini une stratégie afin de concevoir la BDBO OntoDB2 capable de supporter différents formalismes d'ontologies. Cette stratégie consiste en quatre points principaux.

1. Définir un formalisme d'ontologie noyau et de structure très simple (principalement constitué des constructeurs canoniques des différents formalismes que nous avons étudié) et l'enrichir suivant les besoins de constructions orthogonales des formalismes d'ontologies que nous avons étudiés.
2. Définir comment ce formalisme noyau peut être étendu par de nouvelles constructions.
3. Identifier des hypothèses suffisantes puis proposer une politique originale et efficace de gestion des DBO au sein d'OntoDB2. Cette politique se base en particulier sur le fait que les instances de classes non canoniques peuvent être dérivées de leur représentation comme des instances de classes canoniques. De ce fait, il n'est donc pas nécessaire de représenter dans la BDBO OntoDB2 les instances de classes non canoniques.
4. Utiliser les mécanismes et les capacités des bases de données pour réaliser certains raisonnements au sein de la BDBO OntoDB2.

Nous avons donc défini un formalisme d'ontologie noyau composé des constructions canoniques communes aux formalismes d'ontologies PLIB, RDFS et OWL Lite. Ces différents formalismes modélisent les domaines en termes de classes et de propriétés qui les caractérisent. Le

domaine de valeurs de ces propriétés est défini par un type de données (qui peut être une classe). Chaque propriété peut être associée soit à une seule valeur, soit à une collection de valeurs. Enfin des méta-descripteurs peuvent être utilisés afin d'enrichir la description textuelle des concepts (classes et propriétés).

Au final, le formalisme noyau a été défini à partir du formalisme d'ontologie PLIB. OntoDB2 bénéficie ainsi du méta-modèle réflexif de ce dernier : EXPRESS. ce méta-modèle, combiné à une architecture de type 3, permet d'avoir un accès générique à la partie ontologie et, confère à OntoDB2 le support de l'évolution du formalisme d'ontologie noyau. La BDBO OntoDB2 est également dotée de capacités originales pour la gestion et l'interrogation des DBO non canoniques.

En effet, dans OntoDB2, les DBO non canoniques (respectant le cadre d'hypothèses défini) sont rattachées à leur classe canonique de base et sont stockées comme des DBO canoniques appartenant à ces dernières. Des vues sont définies pour calculer l'extension des classes non canoniques à partir des autres classes. Cela permet, et c'est une caractéristique originale d'OntoDB2, d'exploiter les capacité des bases de données traditionnelles pour réaliser les inférences sur les DBO.

Afin d'assurer une implémentation possible des ontologies définies suivant ce modèle d'ontologie, nous avons défini un ensemble d'hypothèses que doivent respecter les ontologies et les données pour pouvoir être supportées par la BDBO OntoDB2.

- L'héritage simple au sein de l'ensemble des classes (canoniques et non canoniques).
- Le typage fort des propriétés qui doivent avoir un unique domaine et un unique co-domaine.
- La mono-instanciation : une instance ne peut appartenir qu'a une seule classe de base qui est le minimum pour la hiérarchie de subsumption de l'ensemble des classes auxquelles elle appartient.
- Le typage fort des instances qui doivent spécifier leur classe d'appartenance et qui ne peuvent être décrites que par les propriétés applicables à cette classe.
- La complétude de description des classes, des propriétés et des instances.

Ces hypothèses ne paraissent pas trop limitatives puisque ce sont des hypothèses de la théorie des bases de données traditionnelles qui sont effectuées de façon habituelle, lorsque des schémas ou des données sont chargés dans une base de données.

Enfin, OntoDB2 utilise l'interpréteur de requête ontologique pour réaliser certaines inférences de manière immédiate lorsque cela est possible. C'est en particulier le cas lorsqu'une requête porte sur l'inverse d'une relation canonique. OntoDB2 permet également d'indexer certaines inférences permettant ainsi d'optimiser les traitements lors de l'interrogation des données. En dernier recours, les mécanismes des bases de données traditionnelles, tels que les triggers, sont utilisés pour saturer les données.

Ainsi, par rapport aux BDBO traditionnelles, la BDBO OntoDB2 permet de supporter des ontologies définies par des formalismes d'ontologies différents. Dans le cadre d'hypothèses assez strictes, seules les DBO non canoniques sont représentées dans OntoDB2 et, à la différence des BDBO traditionnelles qui utilisent soit des raisonneurs soit la saturation systématique de toutes les données, OntoDB2 exploite les mécanismes de bases de données pour réaliser les inférences. Afin de valider les choix effectués lors de la spécification d'OntoDB2, nous l'avons implanté sur le SGBD PostgreSQL. Cette implantation est présentée au chapitre suivant.

Chapitre 4

Implémentation de l'architecture de BDBO OntoDB2

Sommaire

Introduction

Nous avons présenté dans le chapitre précédent, les spécifications de la BDBO OntoDB2. Le but de ce chapitre est d'en présenter la mise en œuvre au travers des différentes parties d'OntoDB2.

- La partie méta-schéma qui stocke à la fois, le méta-schéma lui-même et le formalisme d'ontologie. Cette dernière permet d'assurer la flexibilité du formalisme d'ontologie et du système de types de données, et d'offrir un accès générique à la fois à la partie méta-schéma et à la partie ontologie.
- La partie ontologie qui représente les ontologies définies à partir du formalisme d'ontologie. Comme nous l'avons déjà précisé, cette partie doit permettre de représenter efficacement les ontologies PLIB, et offrir un accès rapide à ces dernières. Pour cela, deux transformations du modèle PLIB ont été réalisées ; d'une part pour simplifier sa structure de représentation en bases de données et d'autre part pour faciliter son accès par les programmes et sa compréhension par les utilisateurs.
- La partie donnée où sont stockées les instances des classes des ontologies et mises en œuvre sur les données les politiques de gestion et d'accès définies au chapitre précédent.

Pour chacune de ces parties, nous présentons :

- comment est générée la structure des tables ;
- comment est réalisé l'accès aux informations représentées ;
- l'API permettant de manipuler les informations représentées ;
- comment est réalisée l'alimentation des tables.

Cette implémentation OntoDB2 a pour objectif de valider les spécifications qui ont été définies au chapitre précédent. Le prototype que nous décrivons dans ce chapitre a comme support le SGBD relationel PostgreSQL [24].

Nous commençons par présenter dans la section 1 la conception de la partie méta-schéma. Dans notre cas, le méta-schéma utilisé est celui du langage EXPRESS. Nous présentons donc le langage EXPRESS [62] ainsi que l'environnement de développement associé ECCO [63] que nous avons utilisé pour développer les différents modules nécessaires. La section 2 présente la conception de la partie ontologie, nous y présentons les programmes génériques de nous avons développés pour faciliter l'extension du formalisme d'ontologie et des types de données. La section 3 présente une synthèse sur les parties méta-schéma et ontologie. Dans la section 4 nous présentons les modules développés pour la partie données. Nous présentons dans la section 5 le client OntoWEB que nous avons développé pour permettre la manipulation des ontologies et des DBO associées. Nous présentons dans la section 6 une synthèse sur l'implémentation d'OntoDB2 et nous terminons par une conclusion dans la section 7.

1 La partie Méta schéma

Le formalisme d'ontologie que nous avons proposé incluant pour l'essentiel, PLIB, et ce formalisme étant lui-même défini dans un langage, EXPRESS, doté d'un méta-modèle, nous avons décidé d'utiliser le méta-modèle d'EXPRESS pour la définition de la partie méta-schéma d'OntoDB2. Définir la partie méta-schéma suppose :(1) de définir la structure de table de ce dernier, (2) de peupler ces tables avec le formalisme d'ontologie, et le méta-schéma lui-même car nous souhaitons un méta-schéma réflexif, et enfin (3) de fournir une API d'accès à cette partie.

La réflexivité du méta-schéma, permet de décrire ce dernier comme une instance de lui-même, au même titre que le formalisme d'ontologie qui est une instance du méta-schéma. Le formalisme d'ontologie d'OntoDB2 ayant la particularité d'être évolutif, la conception de la partie ontologie doit donc permettre son évolution à moindre coût. Pour cela, la conception de cette partie doit être générique par rapport au formalisme d'ontologie, de sorte qu'une modification du formalisme d'ontologie, n'entraîne pas de modification des modules réalisés. Nous avons donc utilisé des techniques d'Ingénierie Dirigée par les Modèles (IDM) pour réaliser les modules nécessaires pour la méta-schéma et la partie ontologie. Pour ce faire, ces modules ont été programmés au niveau méta-schéma. L'utilisation d'une approche générique, nous a permis d'utiliser les mêmes programmes à la fois pour la définition de la partie méta-schéma et pour la définition de la partie ontologie d'OntoDB2. Nous ne présenterons donc que la mise en œuvre des composants de la partie ontologie. Avant de présenter la définition de la partie ontologie, nous définissons d'abord le principe de l'Ingénierie Dirigé par les Modèles qui permet de définir des programmes génériques. Ensuite, nous présentons le langage EXPRESS que nous avons utilisé et, l'environnement D'IDM d'EXPRESS :ECCO.

[24] http ://www.postgresql.org/

1.1 Ingénierie dirigée par les modèles (IDM)

L'Ingénierie Dirigée par les Modèles, ou IDM, correspond à un paradigme relativement récent dans lequel le code source n'est plus considéré comme l'élément central d'un logiciel, mais comme un élément dérivé. Un des aspects de l'IDM, consiste à automatiser la transformation des modèles en générant le code par programmation au niveau du méta-modèle. Ceci permet par exemple de transformer le code d'un langage en un autre, ou une modélisation abstraite en une structure de classes, ou même un modèle de données en un autre modèle tout en assurant que les propriétés des données sont conservées lors de la transformation en s'appuyant sur leur méta représentation. Un modèle de données sera alors défini comme une instance d'un autre modèle général, appelé méta-modèle. Nous avons utilisé dans nos travaux l'environnement d'IDM EXPRESS (ECCO Tool Kit) où nous avons programmé les modules de transformation de modèles en utilisant le langage EXPRESS. Le méta-schéma express est présenté en annexe.

1.2 EXPRESS

EXPRESS [32, 31] est un langage de modélisation conceptuelle et de spécification des données conçu dans le cadre du projet STEP [25] (STandard for the Extange of Product model data), officiellement connu sous la référence ISO 10303. Son objectif principal est la description de schémas en vue de l'échange de données conformes à ces schémas [62, 24]. EXPRESS n'est pas seulement une notation permettant la modélisation des données, c'est aussi un formalisme de spécification, c'est à dire qu'il permet une description complète, non ambiguë et traitable par machine.

EXPRESS est également une série de normes définissant un environnement constitué de :

- un langage de modélisation de l'information : EXPRESS [32] ;
- un format d'instances qui permet l'échange de données entre systèmes [34] ;
- une infrastructure de méta-modélisation associée à une interface d'accès normalisée, appelée SDAI, pour accéder et manipuler simultanément les données et le schéma de n'importe quel schéma EXPRESS. Cette interface associée à un méta-schéma d'EXPRESS en EXPRESS a d'abord été définie indépendamment de tout langage de programmation [61] puis des implémentations spécifiques ont été spécifiées pour le langage C++ (2000), JAVA (2000) et C (2001) ;
- un langage déclaratif de transformation de modèles EXPRESS-X [33].

Enfin, le langage EXPRESS possède un langage procédural complet (analogue à PASCAL) pour l'expression de contraintes. Au prix de quelques extensions mineures, ce langage peut également être utilisé comme langage impératif de transformation de modèles. Dans l'environnement que nous utilisons, cette extension porte le nom d'EXPRESS-C. C'est ce langage que nous avons utilisé pour réaliser nos différents modules. C'est aussi ce langage qui doit être utilisé si l'on souhaite effectuer des modifications majeures dans le formalisme d'ontologie. Par modification majeure, nous entendons l'ajout de nouvelles entités, mais aussi l'ajout de nouveaux comportements.

Dans le langage EXPRESS, l'accent principal est mis sur la précision du modèle et tout particulièrement sur les contraintes que doivent respecter les données pour être acceptées comme conformes au modèle. Ceci assure la fiabilité de l'information représentée. Par contre, à la diffé-

[25] http ://www.steptools.com/library/standard/

rence des formalismes de modélisation objets tel UML ou OMT, il n'est pas destiné à modéliser des systèmes informatiques faisant intervenir une composante dynamique. En EXPRESS, les objets ne possèdent pas de méthodes. La connaissance procédurale représentable s'exprime exclusivement soit sous forme de contraintes d'intégrité, soit sous forme de fonctions de dérivation exprimant comment un attribut se calcule à partir d'autres attributs. Nous présentons ci-dessous les concepts du langage EXPRESS (entités, attributs, contraintes, fonctions et procédures).

1.2.1 Les entités

Une entité correspond à ce qui possède, dans le domaine à modéliser, une existence autonome. Le langage EXPRESS étant un formalisme de modélisation orienté objet, il permet de modéliser les entités structurelles sous forme de hiérarchies associées à un mécanisme de factorisation/héritage. Il autorise différents types d'héritages : simple, multiple et répété (lors d'un héritage répété, un attribut n'existe qu'une fois). Ce mécanisme d'héritage permet d'étendre les propriétés définies dans une entité à toutes ses descendantes. De même, les contraintes liées à l'entité mère sont héritées par toutes les entités filles.

1.2.2 Les attributs

Le langage EXPRESS définit deux types d'attributs :

1. Les **attributs autonomes** qui sont indépendants de tout autre attribut ; ils peuvent être associés à une valeur optionnelle ;
2. Les **attributs dérivés ou calculés** qui dépendent fonctionnellement d'autres attributs et peuvent s'évaluer par une fonction explicite, exprimée en EXPRESS.

Dans la table 4.1, la classe *Personne* est caractérisée par les attributs autonomes obligatoires, *nom* et *prenom* de type chaîne de caractères et un attribut dérivé *nomComplet*, de type chaîne de caractères, dont la valeur est calculée par la fonction *computeNomComplet*. L'attribut *aEcrit* dans l'entité *Personne* est de type ensemble (*set*). Enfin, l'attribut *aEcrit* a un attribut inverse (*auteurs*) qui précise la cardinalité inverse de l'association *aEcrit*. Cette cardinalité est 1 à N. Notons que quand un attribut inverse n'est pas défini, cela signifie que la cardinalité inverse est 0 :N.

1.2.3 Les types

Ils définissent les domaines auxquels doivent appartenir les attributs, les variables ou les paramètres d'une fonction ou une procédure. Les types peuvent être les suivants : les types simples, les collections, les types nommés, les types *select*, les types *enumeration*.

- **Type simples** : On retrouve ainsi les booléens (*boolean*), les logiques tri-valués (*logical*), les nombres (*number*) les réels(*real*), les entiers (*integer*), les chaînes de caractères (*string*).
- **Type collection** : Les types peuvent être des collections d'entités, de types simples ou même d'autres collections. Les collections peuvent être de nature différente, on retrouve les listes (*list*), les ensembles (*set*), les ensembles avec repétition (*bag*) et les tableaux (*array*). Dans la table 4.1 par exemple, l'attribut *aEcrit* a pour co-doomaine une collection de type ensemble ayant entre 1 et n (?) valeurs.

```
SCHEMA exemple;

TYPE t_age = INTEGER;
WHERE wr0 : SELF >= 0 AND SELF <= 200;
END_TYPE;
TYPE t_genre = ENUMERATION OF(feminin,masculin);
END_TYPE;

ENTITY Personne;
ABSTRACT SUPERTYPE OF (Etudiant);
        nom : STRING(256);
        prenom : STRING(256);
        age : OPTIONAL t_age;
        genre : t_genre;
        aEcrit : SET[0 :?] OF Document;
        DERIVE
                nomComplet : STRING(512) :=computeNomComplet(nom,prenom);
        WHERE
                wr1 : NOT (nom[1] IN ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']);
END_ENTITY;

ENTITY Etudiant;
SUBTYPE OF Personne;
END_ENTITY;

ENTITY Document;
       titre : STRING(256);
       auteurs : SET[1 :?] OF Personne FOR aEcrit;
END_ENTITY;

FUNCTION computeNomComplet(nom :string;prenom :string) :STRING;
       LOCAL
              res :STRING;
       END_LOCAL;
       res := nom + ' ' + prenom;
       RETURN (res);
END_FUNCTION;

END_SCHEMA;
```

TAB. 4.1 – Exemple de schéma EXPRESS : notation textuelle

- **Type nommé** : Il est en effet possible de créer des types à partir de types existants ou d'autres types utilisateurs. Par exemple, le type *t_age* (cf. table 4.1) basé sur le type entier, restreint l'ensemble des valeurs permises à l'intervalle [0 .. 200].
- **Union de type** (*select*) : Il permet de définir un domaine de valeurs pour un type comme union de domaines de valeurs d'autres types.
- **Type énuméré** (*enumeration*) : Il permet de définir une énumération de valeurs. Par exemple dans la table 4.1, le type *t_genre* n'admet que les valeurs *féminin* et *masculin*.

1.2.4 Les contraintes

Le langage EXPRESS offre un langage de description de contraintes très élaboré. L'on distingue deux types de contraintes : les contraintes locales qui s'appliquent individuellement sur chacune des instances de l'entité où elles sont définies (WHERE) et les contraintes globales qui nécessitent une vérification globale sur l'ensemble des instances d'une entité donnée.

Les contraintes locales (WHERE) sont définies au travers de prédicats que chaque instance de l'entité doit vérifier. Ces prédicats permettent par exemple de limiter la valeur d'un attribut en définissant un domaine de valeurs ou encore d'imposer certaines relations entre les valeurs de différents attributs d'une même entité. Par exemple, la contrainte *wr1* de la table 4.1 spécifie que le nom d'une personne ne doit pas commencer par un chiffre.

Les contraintes globales comportent trois constructions :

1. La contrainte d'unicité (UNIQUE) contrôle l'ensemble d'une population d'instances issues d'une même entité pour s'assurer que l'attribut auquel s'applique la contrainte possède pour chaque instance une valeur différente.
2. La contrainte de cardinalité inverse (INVERSE) permet de spécifier le nombre d'entités d'un certain type qui référencent une entité donnée dans un certain rôle. L'attribut inverse exprime l'association entre une entité sujette et des entités référençant l'entité sujette par un attribut particulier.
3. Les règles globales (RULE) ne sont pas déclarées au sein des entités mais sont définies séparément. Elles permettent de vérifier des fonctions logiques qui nécessitent pour leur évaluation, de parcourir l'ensemble des instances d'un type donné.

1.2.5 Les fonctions et procédures

Les fonctions et les procédures sont introduites respectivement par les mots clés *function* et *procedure*. Le langage de description s'apparente au langage PASCAL. Des variables peuvent être déclarées localement (*local*) et les structures de contrôle classiques (*repeat*, *if*,...) sont disponibles. De telles constructions ne sont utilisables que pour définir des contraintes ou des fonctions de dérivation.

1.2.6 Représentation des instances : le fichier physique

Les instances d'un modèle en EXPRESS sont décrites et échangées à travers un fichier textuel [34] aussi appelé le fichier physique. Dans ce fichier, chaque instance est identifiée par un

OID (#i). Elle est caractérisée par le nom de l'entité instanciée et est décrite par la suite des valeurs de ses attributs représentées entre parenthèses. Les collections d'éléments sont définies entre parenthèses. Les attributs optionnels qui n'ont pas de valeur ont leur valeur représentée par '$' et, les types énumérés par une notation encadrant l'identificateur par un point à chacune de ses extrémités. Les attributs dérivés et inverses ne sont pas représentés puisqu'ils peuvent être calculés par le système à partir du schéma. Un exemple d'instances du modèle de données que nous avons décrit à la table 4.1 est présenté dans la table 4.2. Dans cette table, l'instance d'*oid #5* représente l'étudiant de *nom Magnet* et de *prénom Philippe*. L'attribut dérivé *nomComplet* est représenté par le caractère *$*. Cet étudiant est de genre *masculin* et a écrit deux livres dont les *oid* sont *#1* et *#2*.

```
ISO-10303-21;
HEADER;
FILE_DESCRIPTION((''),'2;1');
FILE_NAME('person.p21', '2009-08-24T11:22:30',(''),(''),
        'ECCO RUNTIME SYSTEM BUILT-IN PREPROCESSOR V3.1.5',
        'ECCO RUNTIME SYSTEM V3.1.5', '');
FILE_SCHEMA(('PERSON'));
ENDSEC;

DATA;
#1=DOCUMENT('Rapport des zones sismologiques de France');
#2=DOCUMENT('Mémoire de thèse de doctorat');
#3=DOCUMENT('Le vieil homme et la mer');
#4=ETUDIANT(' Hemingway','Ernest', $, .MASCULIN., (#3));
#5=ETUDIANT('Magnet', 'Philippe', $, .MASCULIN., (#1,#2));
#6=ETUDIANT('Semma','Dorothé',$, .FEMININ., $);
ENDSEC;
END-ISO-10303-21;
```

TAB. 4.2 – Exemple de fichier d'instance EXPRESS : format physique p21

1.2.7 La notation graphique EXPRESS-G

La représentation textuelle des schémas EXPRESS est essentielle pour le traitement automatique des schémas. C'est elle qui est exploitable par machine pour en vérifier la correction et la conformité des instances. Elle est cependant, difficilement lisible. Un formalisme graphique EXPRESS-G a donc été défini pour donner une vue synthétique des schémas de données, et faciliter leur conception dans les phases initiales d'analyse des problèmes à modéliser. Ce symbolisme de représentation est partiel et ne permet pas d'exprimer les contraintes d'intégrité. La nature des attributs (optionnel, inverse ou dérivé), ainsi que leur type peuvent par contre être représentés.

La figure 4.1 montre la représentation graphique du schéma de la table 4.1. Cette figure montre par exemple que les entités sont représentées par des rectangles, les attributs obligatoires sont représentés par un trait plein terminé par un cercle et les attributs optionnels par un trait interrompu terminé également par un cercle. La nature des attributs est précisée par l'utilisation des mots clés DER et INV et leur cardinalité par S (set), L (list), A (array), et B (bag).

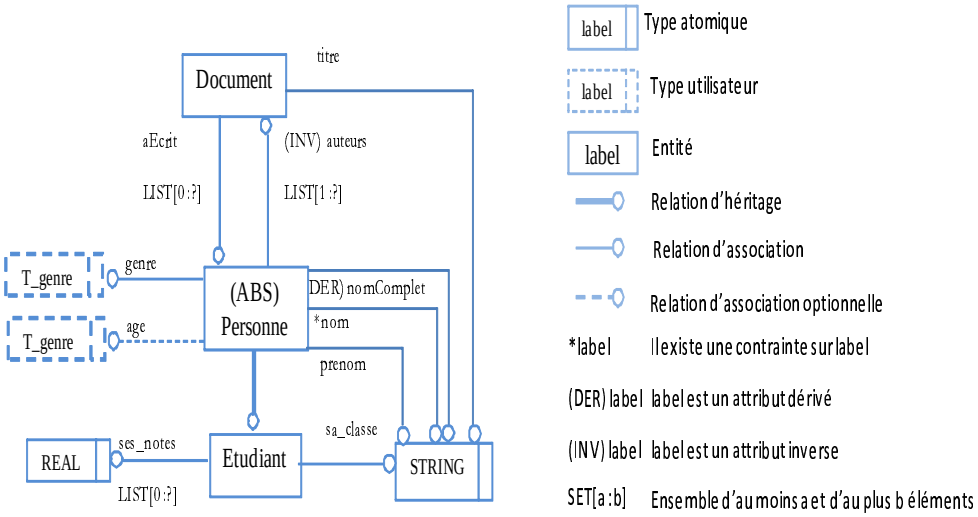


FIG. 4.1 – Exemple de schéma EXPRESS : notation graphique EXPRESS-G

1.3 Génération et exploitation de la représentation du méta-schéma EXPRESS

ECCO Tool Kit [63] est un environnement de développement permettant la traduction d'une spécification écrite en EXPRESS en un programme exécutable permettant de représenter des ensembles d'instances conformes à la spécification en mémoire centrale et de vérifier leur correction. ECCO offre beaucoup de possibilités parmi lesquelles :

- l'accès à la description du schéma lui-même sous forme d'instances du méta schéma d'EXPRESS
- l'édition, la vérification de la syntaxe et la sémantique des modèles,
- la vérification des contraintes sur les instances.

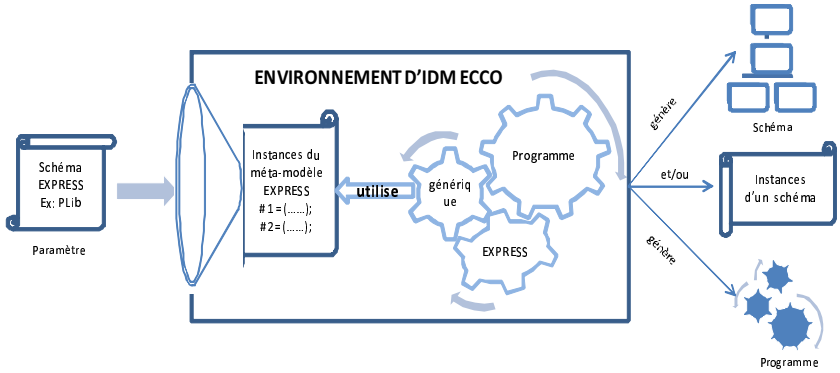


FIG. 4.2 – Environnement d'IDM EXPRESS

La figure 4.2 montre la structure de l'environnement d'IDM EXPRESS. Un schéma est fourni comme point de départ, ce schéma est ensuite réifié par le compilateur ECCO. Il peut alors être accessible par les primitives de haut niveau sous-forme d'instances du méta-schéma. Un programme peut alors accéder à ces instances afin de générer du code par exemple.

Par exemple, dans le programme de la figure 4.3-a, la primitive *sch.entity_ types* de la première instruction *repeat* permet d'accéder à toutes les entités du schéma courant (*sch*). Et, pour chaque entité (*ent*), la primitive *ent.explicit* de la seconde instruction *repeat* permet d'accéder à l'ensemble de ses attributs. Ainsi, appliqué au modèle de la figure 4.1, ce programme parcourt toutes les entités de ce modèle, génère pour chacune d'elle une classe JAVA avec un constructeur par défaut, déclare les attributs comme privés et définit une méthode d'accès (accesseur) et une méthode de mise à jour (modificateur) pour chaque attribut. Le résultat de ce programme est visible à la figure 4.3-b.

```
PROCEDURE generateJAVA(sch : scheme_info;Out: file_output);
    -- parcours des entités
    REPEAT FOREACH ent IN sch.entity_types;
        Out.write_nl('package exemple;');
        Out.write_nl('');
        Out.write_nl('public class '+ ent.name + '{');
        -- constructeur par défaut
        Out.write_nl('public '+ ent.name + '(){');
        Out.write_nl('}');
        --parcours des attributs
        REPEAT FOREACH att IN ent.explicit;
            Out.write_nl('private Object '+att.name + ';');
            --modifieur
            Out.write_nl('public void set' + att.name +'(Object obj){');
            Out.write_nl('      ' + att.name + '= obj;');
            Out.write_nl('}');
            --accesseur
            Out.write_nl('public Object get'+ att.name +'(){');
            Out.write_nl('      return '+att.name+';');
            Out.write_nl('}');
        END_REPEAT;
        Out.write_nl('}');
    END_REPEAT;
END_PROCEDURE;
```

(a) : extrait de programme générique

```
package exemple;

public class Document{
Public Document(){
}
private Object titre;
public void setTitre(Object obj){
    titre = obj;
}
public Object getTitre(){
    return titre;
}
}
```

(b) : résultat

FIG. 4.3 – Exemple de programme générique en EXPRESS-C

2 La partie ontologie

Cette section décrit la mise en œuvre des différents modules développés pour la représentation des ontologies dans OntoDB2. Nous commençons par présenter ci-dessous, l'approche de transformation du modèle PLIB que nous avons réalisée afin d'assurer une représentation efficace du formalisme d'ontologie et, de faciliter sa compréhension par les utilisateurs.

2.1 Conception des schéma de représentation des ontologies

Le schéma logique de la partie ontologie est très proche de la structure du formalisme d'ontologie PLIB, que nous avons étendu pour accepter les ontologies d'autres formalismes d'ontologie (RDFS, OWL). Cependant, Il se pose le problème de la gestion de la représentation des mécanismes objet au sein de la base de données. La première implantation du formalisme d'ontologie PLIB dans le SGBD PostgreSQL, faite dans OntoDB, a consisté à analyser et implémenter en l'état tout le modèle PLIB dans le SGBD PostgreSQL [19]. Ainsi, tous les concepts de PLIB et donc ceux du langage objet EXPRESS, ont été mappés dans l'environnement PostgreSQL. Ledit SGBD implémentant déjà la notion d'héritage de tables, les choix effectués [19] ont porté sur la mise en œuvre

du polymorphisme, à travers la création de systématique pour tout attribut de type objet d'une table d'aiguillage déterminant le type réel de la classe référencée au niveau des instances, sur la représentation des relations d'agrégation, ainsi que sur la représentation des types de données non supportés par le système (union de type, type énuméré et autre type défini par l'utilisateur).

Plusieurs problèmes d'efficacité (performance, espace de stockage) de la base de données résultante ont été décelés lors de la manipulation des données. Ces problèmes sont dus à la structure de la base de données générée à partir des règles de mapping utilisées.

Les deux principaux problèmes d'efficacité sont les suivants :

1. la lenteur dans l'exécution des requêtes impliquant l'ontologie, ceci étant dû au nombre important de jointures entre tables et tables d'aiguillages pour représenter le polymorphisme dans l'ontologie ;
2. la complexité du schéma de la base de données dû au nombre de tables d'entités et de tables d'aiguillages correspondant aux différentes entités du formalisme d'ontologie PLIB ; en effet, le formalisme d'ontologie PLIB comporte un grand nombre d'entités (217 entités et 118 types définis), sa transposition dans OntoDB a généré 568 tables dont 160 tables d'aiguillage.

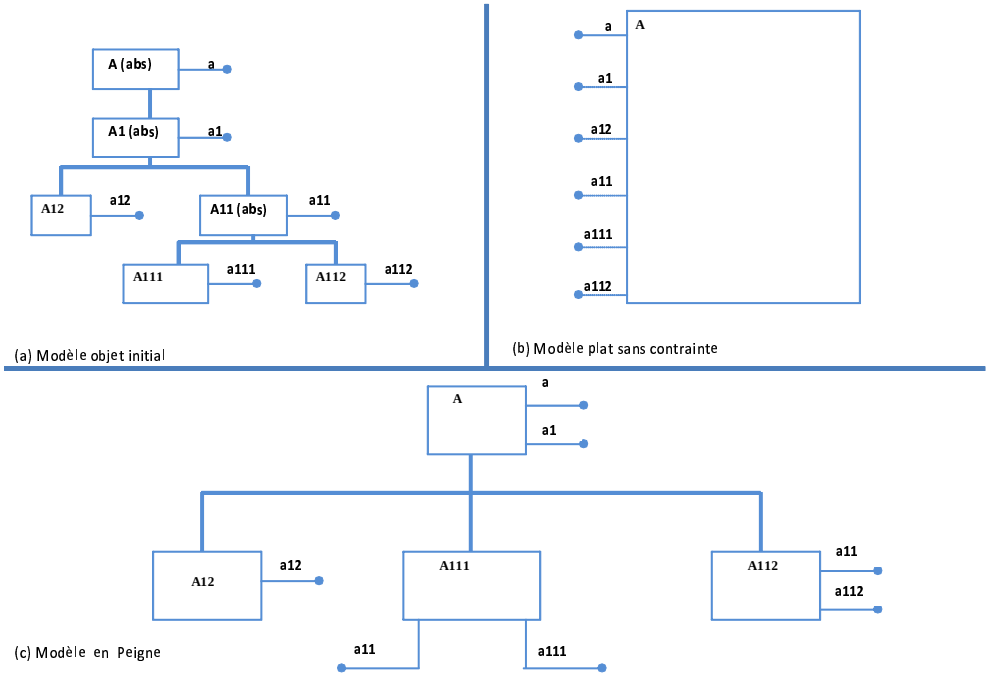


FIG. 4.4 – Transformation du modèle PLIB

Pour résoudre ces deux problèmes, le modèle PLIB a subi deux transformations comme le montre, de façon simplifiée, la figure 4.4. En effet, nous avons décidé que le modèle PLIB ne devait pas être utilisé en l'état, car il était trop complexe tant du point de vue efficacité puisqu'il créait trop de tables, que du point de vue compréhension puisque le modèle devait être facilement compréhensible par les utilisateurs qui souhaiteraient modifier le modèle pour lui ajouter leur propres extensions. Néanmoins, le fait que le modèle soit exprimé en EXPRESS était très commode

pour permettre d'utiliser les techniques d'IDM. Notre stratégie a donc consisté à transformer le modèle EXPRESS de PLIB en un modèle EXPRESS ayant le même pouvoir d'expression mais comportant beaucoup moins d'entités à la fois pour le rendre plus facilement compréhensible, et pour augmenter la rapidité des requêtes au niveau ontologique. Cette simplification a été obtenue principalement par la mise à plat des hiérarchies de généralisation/spécialisation définissant ainsi une structure « plate » que nous appelons *Flatlib*. La figure 4.4-b illustre sur un exemple simple, la mise à plat de la hiérarchie du modèle de la figure 4.4-a. Cette figure montre que toutes les classes de la hiérarchie du modèle objet initial sont fusionnées dans la seule entité *A* du modèle plat sans contrainte.

C'est donc la structure plate *Flatlib*, qui est représentée dans la base de données. Pour la partie ontologie, toute hiérarchie d'héritage est donc représentée dans une table unique. Par contre, si cette structure plate est adaptée à la gestion optimale, en termes de complexité temporelle d'accès aux données objets qui représentent chaque ontologie, elle n'est malheureusement pas adaptée à la programmation aisée : toutes les entités d'une hiérarchie étant regroupées, il est difficile de savoir quels attributs doivent être valués ou non valués.

Une troisième représentation du formalisme d'ontologie PLIB, à un niveau de représentation des hiérarchies que nous appelons le *Peigne*, et bien adaptée à la gestion par des programmes objets a donc été définie. Cette nouvelle structure sert de base pour générer une API objet sur *Flatlib*. Comme l'illustre le modèle de la figure 4.4-c, les entités concrètes *A12*, *A111* et *A112* du modèle initial sont représentées sur le modèle en *Peigne* et, chacune d'elle déclare les attributs qui lui sont spécifiques, ce qui n'est pas le cas dans le modèle plat de la figure 4.4-b.

Enfin, comme nous avons défini des règles de transformation pour passer de PLIB à *Flatlib*, nous avons établi des règles de correspondance entre la structure d'accès en *Peigne*, et la structure de représentation *Flatlib*.

Ainsi, la définition de la partie ontologie de OntoDB2 a nécessité la mise en œuvre des modules permettant :

1. la définition de *Flatlib* (représentant la structure des tables) dans lequel les concepts de l'ontologie seront représentés,
2. la définition du *Peigne* (représentant le schéma objet d'accès pour les applications),
3. la mise en correspondance entre *Flatlib* et le *Peigne*,
4. la définition d'une API d'accès,
5. l'importation des ontologies pour alimenter la partie ontologie de la BDBO OntoDB2.

Nous détaillons ces cinq parties dans les sections 2.2 à 2.6.

2.2 Schéma de représentation des ontologies : *Flatlib*

Nous présentons succinctement dans cette section les règles de mise à plat de la structure objet initiale du formalisme d'ontologie PLIB, définies au laboratoire [60]. Nous avons repris ce travail pour réaliser le modèle d'accès et, la transformation des instances de PLIB à *Flatlib* comme nous le verrons aux sections 2.3 et 2.6.

La transformation de la structure initiale de PLIB vers une structure plate, a été effectuée de façon à conserver le plus possible la sémantique du formalisme d'ontologie PLIB : cohérence des entités regroupées et des contraintes. Pour cela, (1) des règles systématiques ont été développées

et, (2) des techniques d'IDM ont été utilisées pour transformer la structure initiale du formalisme d'ontologie PLIB en une structure plate appelé *Flatlib*, dépourvu de relations d'héritage, mais de pouvoir d'expression au moins égal à la structure initiale.

Les transformations qui ont été faite sur la structure initiale de PLIB visaient à éliminer totalement les relations d'héritage et à réduire, autant que possible, les relations d'agrégation entre les entités du modèle. La définition de règles formelles de transformation a donc porté sur les points suivants.

- Quelle simplification peut-on effectuer sur les hiérarchies d'héritage d'un schéma objet tout en conservant la possibilité de polymorphisme pour les instances du schéma ?
- Comment simplifier les relations d'agrégation /composition entre les différentes classes du formalisme tout en conservant sa sémantique ?

Nous présentons ci-dessus quelques règles de simplification de PLIB.

2.2.1 Simplification des hiérarchies

La relation d'héritage est un des mécanismes essentiels de l'approche objet. Elle permet la généralisation et la spécialisation des classes, ainsi que le polymorphisme des instances. Dans une hiérarchie de classes, on peut distinguer des classes abstraites et des classes concrètes. Les classes abstraites ne sont jamais instanciées car elles ne représentent qu'une abstraction de plusieurs classes concrètes. Elles servent, par contre, à factoriser des propriétés ou des comportements. Elles servent également pour référencer de façon générique des instances appartenant à n'importe quelle sous classe. Nous présentons ici la règle de suppression des classes abstraites et, la règle de mise à plat des hiérarchies.

2.2.1.1 Suppression de l'abstrait

La figure 4.5 illustre la règle permettant de supprimer les classes abstraites. L'énoncé est le suivant :

Une classe abstraite sera supprimée : (1) si elle n'est co-domaine d'aucun attribut pouvant avoir pour domaine (même par polymorphisme) une classe concrète, et/ou (2) si toutes les classes abstraites qui la référencent peuvent être supprimées. Dans ce cas, tous ses attributs sont descendus au niveau de ses sous classes directes. Si un attribut est redéfini comme dérivé dans l'une des sous classes, il ne sera pas représenté dans cette sous classe.

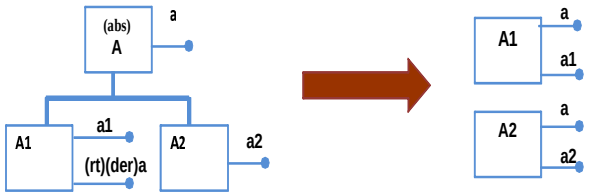


FIG. 4.5 – Suppression de l'abstrait dans une hiérarchie

2.2.1.2 Mise à plat d'une hiérarchie

La figure 4.6 illustre la règle permettant de mettre à plat une hiérarchie dont l'énoncé est le suivant :
Si aucune règle de simplification ne peut être appliquée à une hiérarchie, alors toutes les sous classes sont remontées et fondues dans la racine. Si un attribut est redéfini dans l'une des sous classes, cette redéfinition ne sera pas représentée dans la racine.

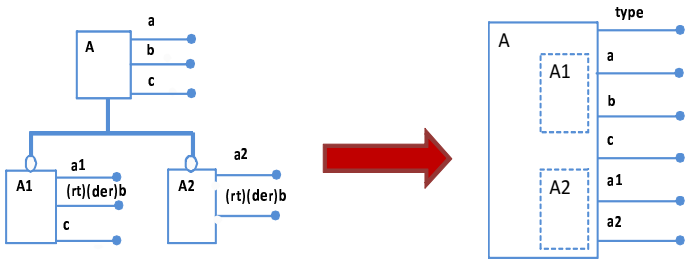


FIG. 4.6 – Mise à plat sans condition

En appliquant la règle de mise à plat sans condition à la hiérarchie de racine A (figure 4.6), tous les attributs des sous types de *A* sont remontés à son niveau.

Notons que dans la racine, l'ajout d'un attribut discriminant (type) permet de déterminer le type réel des instances. Cet attribut va ainsi prendre le nom de l'entité initiale dans le formalisme d'ontologie PLIB (ici *A1* et *A2*).

2.2.2 Simplification des aggrégats

La figure 4.7 illustre la règle d'absorption multiple permettant de supprimer une relation d'aggrégation. L'énoncé de cette règle est comme suit :

*(1) Si toutes les références à une classe sont de cardinalités ([1 :1],*INV*[0 :1]) ou ([0 :1],*INV*[0 :1])* [26]*, et (2) si toute instance de cette classe est effectivement liée a une seule instance référençante (contrainte « ou exclusif »ou « used_ once »), et (3) si cette entité n'a pas de sous-type, alors, cette entité sera absorbée par chacune des entités la référençant.*

Les simplifications apportées à la structure initiale du formalisme d'ontologie PLIB ont permis d'obtenir dans *Flatlib*, 43 entités et 80 types définis à la place des 217 entités et 118 types définis dans le modèle initial PLIB.

2.2.3 Schéma des correspondances entre PLIB et *Flatlib*

La transformation de PLIB à *Flatlib* a été réalisée en utilisant des techniques d'IDM. La correspondance entre ces deux structures a elle-même été représentée dans un schéma de correspondance. Ce schéma de correspondance est lui-même représenté comme un schéma EXPRESS.

[26] [1 :1] et INV[0 :1] représentent respectivement la cardinalité minimale et maximale directe et inverse (INV)

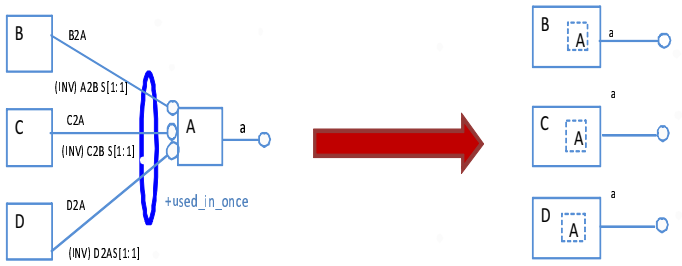


FIG. 4.7 – Absorption multiple

Il est donc également géré par le même méta-schéma, qui est le méta-schéma d'EXPRESS en EXPRESS. Le schéma de correspondance (figure 4.8) contient à la fois les entités du schéma source (*source_ entity*) et les entités du schéma résultant de l'application des règles (*computed_ entity*). Le schéma final est constitué des entités calculées ainsi que des entités du schéma initial qui n'ont pas disparues lors de l'application d'une règle.

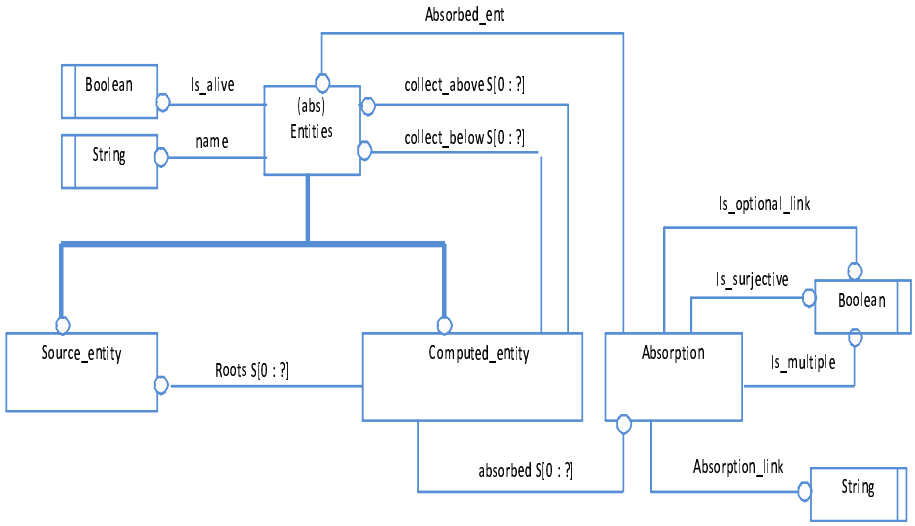


FIG. 4.8 – Schéma de correspondance des transformations de PLIB à Flatlib

Les attributs ne sont pas représentés puisque leurs évolutions sont parfaitement définies par chaque triplet <*classe_ absorbante, classe_ absorbée, nom_ de_ règle*>. L'entité *entities*, supertype de l'entité *source_ entity* et de *computed_ entity*, est une entité qui représente toutes les entités du modèle et leur état a un moment donné. L'attribut *name* est son nom, et l'attribut *is_ alive* spécifie si l'entité existe encore dans le schéma à un moment donné. Toute transformation élémentaire comporte : une entité « *absorbante* », dans laquelle une ou plusieurs autres entités sont fusionnées, donnant lieu à une nouvelle entité. La nouvelle entité est représentée par l'entité *computed_ entity*. L'entité absorbante est représentée soit par une entité *source_ entity* lors de la première transformation de cette *source_ entity*, soit par l'entité *computed_ entity* elle-même dans les transformations suivantes. L'entité fusionnée est représentée par les autres attributs de

computed_ entity :

- *collected_ above* dans le cas d'une suppression d'abstrait ;
- *collected_ below* dans le cas d'une concrétisation d'abstrait, ou d'une mise à plat ;
- *absorbed* dans le cas d'une absorption.

Ce schéma permet donc de représenter, sous forme d'instances de ses entités, les transformations apportées au schéma source au fur et à mesure de l'application des règles. L'historique du mapping est conservé à travers ces instances. Nous utilisons en particulier ce schéma de correspondance à la section 2.6.1, pour l'importation des ontologies PLIB dans la partie ontologie d'OntoDB2.

2.3 Schéma d'accès à la partie ontologie : Le *Peigne*

Le développement d'applications est destiné à être effectué dans un langage objet (JAVA, C++, C#, ...) dont des avantages principaux sont la spécialisation/généralisation et le polymorphisme. La structure plate (*Flatlib*) définie ci-dessus est un schéma simple. Cette simplicité est due essentiellement à la suppression des entités abstraites du modèle objet, ce qui diminue considérablement le nombre important d'entités du schéma initial. *Flatlib* supporte les associations polymorphes, car dans la mise à plat de la hiérarchie, un attribut discriminant est ajouté pour spécifier la valeur effective de l'entité instanciée. Cependant, *Flatlib* n'est pas adapté pour la manipulation par les applications. Les applications manipulent les schémas objets en représentant chaque entité comme une classe JAVA. Mais avec *Flatlib* où la même entité représente en fait plusieurs classes, le programmeur ne peut instancier correctement les classes, ni contrôler des attributs à valuer. En effet, à l'issue de la mise à plat, les attributs obligatoires des classes concrètes deviennent optionnels car un attribut obligatoire pour une classe peut ne pas exister pour une classe sœur absorbée dans la même entité. Ceci augmente considérablement le risque d'erreurs de saisie de champs optionnels. *Flatlib* n'est donc pas adapté à la manipulation par les applications. Nous avons donc défini une troisième structure du formalisme d'ontologie. Cette structure, que nous appelons le *Peigne*, va servir pour la manipulation des données par les applications. Pour cela, le *Peigne* doit posséder une structure adaptée à la modélisation par les objets, tout en restant simple. De plus, c'est sur le *Peigne* que les extensions seront rajoutées. Le *Peigne* devant donc également être évolutif. Ceci amène à limiter les classes abstraites qui rendent la structure des schémas objets difficile à maîtriser.

Afin de permettre de contrôler l'accès aux données, chaque entité concrète existant dans la structure initiale de PLIB, doit être représentée. Cela va permettre d'instancier correctement les éléments et permettre au système de vérifier la correction des attributs obligatoires. Le *Peigne* devant supporter l'héritage et le polymorphisme pour préserver les nombreuses associations polymorphes de la structure objet initiale, toute hiérarchie d'héritage représentée, au niveau *Flatlib* comme une entité abstraite va être représenté dans le *Peigne* par un arbre à un niveau d'hiérarchie comportant :

- une racine (abstraite ou concrète) qui permet les références polymorphes. Cette racine contient les attributs des entités absorbées lors de la transposition de la structure initiale en *Flatlib*, si elles appartiennent à toutes les entités concrètes de la hiérarchie. Dans notre cas, les entités racines du *Peigne* incluent en particulier : *Class*, *Property*, *Data_ type*, ...
- des entités concrètes représentant individuellement chacune des entités concrètes de la hiérarchie.

La figure 4.9 représente un exemple de génération de schéma en *Peigne*. Nous retrouvons au niveau de la racine tous les attributs communs à toutes les entités concrètes. Chaque entité déclare en plus les attributs qui peuvent être valués lors de son instanciation et qui ne sont pas dans l'entité racine.

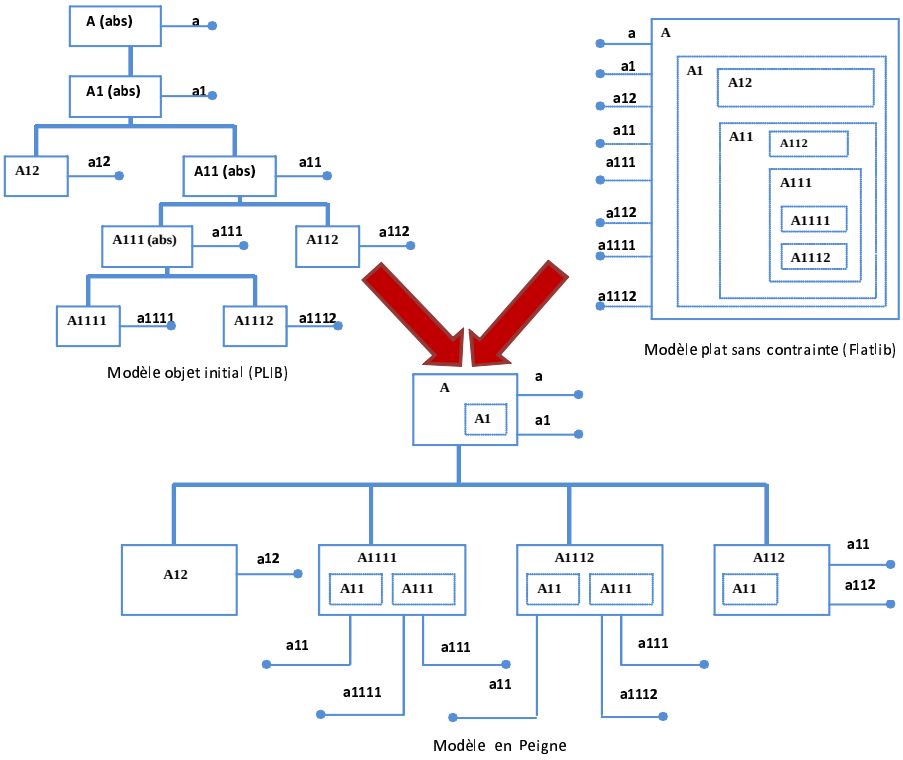


FIG. 4.9 – Principe de génération du Peigne

La figure 4.10 montre l'architecture logicielle du module de définition du schéma *Peigne*. Comme points d'entrée sont fournis les schémas PLIB et *Flatlib*. Ces deux schémas sont utilisés par le programme *générer_ Peigne* pour générer en sortie le schéma *Peigne*.

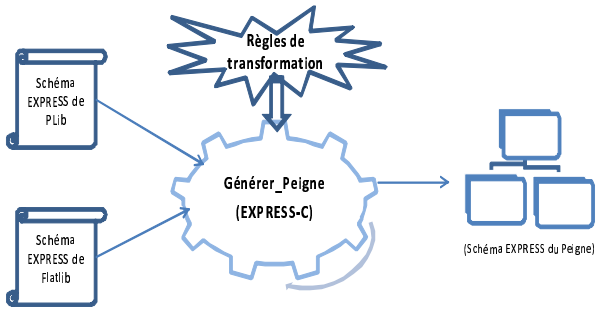


FIG. 4.10 – Architecture logicielle de définition du schéma *Peigne*

L'algorithme du programme *générer_ Peigne* est le suivant :

1. Créer un nouveau schéma nommé *Peigne*.

2. Pour chaque entité de *Flatlib*, créer une entité racine du *Peigne* de même nom.
3. Pour chaque entité concrète du schéma initial remontée dans l'entité *Flatlib*,créer un sous type de l'entité racine du *Peigne* de même nom.
4. Définir les attributs communs à toutes les entités concrètes comme attributs de la racine dans le *Peigne*
5. Ajouter à chaque sous-type dans le *Peigne* les attributs qui s'applique à elle et ne sont pas représentés dans la racine du *Peigne*.

Une fois le schéma de représentation des données et le schéma d'accès disponibles, il reste à réaliser la correspondance entre ces deux schémas. Cette correspondance doit permettre de transformer des instances du schéma *Peigne* vers le schéma *Flatlib* ; cette transformation va par exemple être réalisée lors de la sauvegarde des objets manipulés par les applications. Et, inversement, la correspondance pour transformer les instances du schéma *Flatlib* en des instances du schéma *Peigne* est nécessaire lors de la lecture des données par les applications.

Les correspondances entre le *Peigne* (représentation objet utilisée par les applications) et *Flatlib* (représentation relationnelles qui permet de stocker des instances dans une bases de données) rentrent dans ce qu'il convient d'appeler les *« mappings »Objets/Relationnels*. Nous expliquons dans la section suivante la stratégie que nous avons suivie.

2.4 Correspondances entre le *Peigne* et *Flatlib*

Le problème de la transposition relationnelle d'un schéma objet consiste à mapper les objets d'une application dans une base de données relationnelle. Ce problème présente de nombreuses difficultés. Le code nécessaire pour gérer le mapping est généralement répétitif, sujet aux erreurs et difficile à maintenir. Des solutions logicielles ont donc été développées pour simplifier ce problème. Nous pouvons citer par exemple JDO[27], EJB [28] ou encore Hibernate [29].

Ces différentes couches de persistance permettent de prendre en charge toutes les interactions (mappings) entre l'application et la base de données. Elles offrent les avantages suivants :

- L'accès transparent aux données des bases de données sans pour autant écrire une ligne de code SQL.
- L'abstraction des correspondances complexes : la couche de persistance fournit les fonctionnalités de recherche, de chargement et de mise à jour des objets dans la base de données (prise en compte de l'héritage, langage de requêtes objet puissant, mécanismes d'optimisation des temps d'accès à la base de données relationnelle (cache objet)).
- La portabilité de l'application : de nombreuses infrastructures, implémentent une couche de persistance qui permet l'utilisation transparente de la majorité des SGBD du marché.

Nous avons décidé de travailler avec la bibliothèque de mapping Objets/Relationnel Open Source Hibernate, que nous présentons ci-dessous.

[27] http ://access1.sun.com/jdo/
[28] http ://java.sun.com/
[29] http ://www.hibernate.org

2.4.1 Hibernate

La bibliotheque JAVA Hibernate est un outil de mapping Objet/relationnel gratuit et de source libre (licence LGPL). Il nécessite de préciser les correspondances entre classes JAVA et tables de la base de données, et entre les types de données JAVA et les types de données SQL, de manière simple par des fichiers XML ou par annotation de classes JAVA. Il permet ensuite de faire abstraction de la base de données sous-jacente en permettant de manipuler les objets JAVA adéquats et en prenant en charge la mise à jour de la base de données (programmation objet dans un environnement relationnel). Enfin, Hibernate augmente considérablement les performances en utilisant un cache objet. Les fichiers XML permettant de définir les correspondances entre schéma objet et schéma relationnel sont appelés fichiers de mapping. Le vocabulaire de mapping est orienté JAVA, ce qui signifie que les mappings sont construits autour des classes JAVA et non autour des déclarations de tables. Ces mappings peuvent ensuite être utilisés pour la génération automatique du code SQL de la base de données, et la génération automatique des classes JAVA.

Hibernate permet de projeter tous les concepts objet dans une base de données relationnelle et cela en respectant une DTD particulière.

2.4.1.1 Le mapping de L'héritage

Hibernate supporte les trois stratégies d'héritage (cf. figure 4.11) :

- Mapping d'une hiérarchie sur une seule table avec une colonne discriminante pour spécifier le type réel de l'enregistrement. C'est cette stratégie que nous utilisons compte tenu de la structure plate de *Flatlib*.
- Mapping de chaque classe concrète sur une table.
- Mapping de chaque classe sur une table.

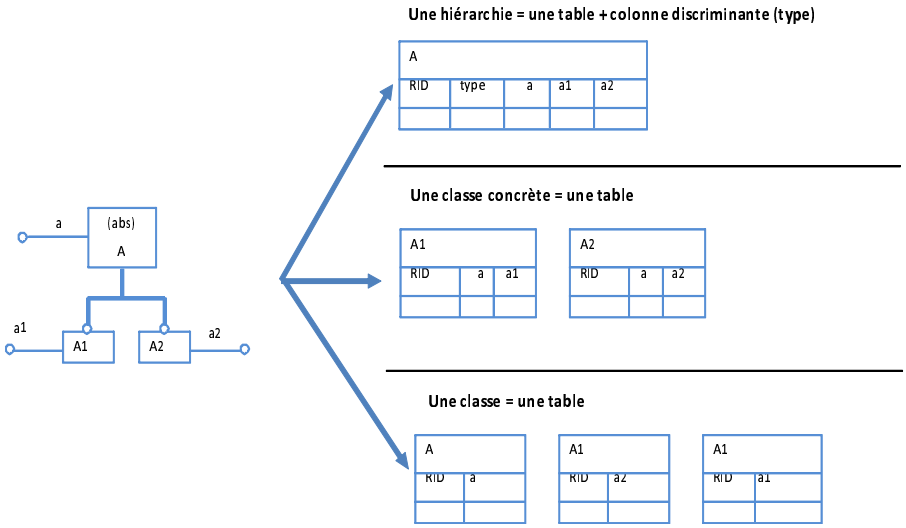


FIG. 4.11 – Stratégie de mapping de l'héritage

2.4.1.2 Le mapping des collections

Les collections sont mappées par les balises suivantes :
- <set> correspondant à une collection d'éléments non ordonnés et sans répétition.
- <list> correspondant à une collection d'éléments ordonnés.
- <map> correspondant à une collection de paire (clé, valeur).
- <bag> correspondant à une collection non triée, non ordonnée qui peut contenir le même élément plusieurs fois.
- <array> correspondant à un tableau indexé d'éléments.

Notons que le mapping des collections tient compte des cardinalités des propriétés (un à un, un à plusieurs, plusieurs à plusieurs). Plusieurs propositions [52, 40] de représentation des collections sont proposées.

- *Mapping d'une relation un à un.* Une relation un à un est une relation où le nombre maximum d'objets participant de chaque côté est un. Dans une relation de type un à un, la clé étrangère doit être dans l'une des tables référençant celle de l'autre table. Si cette relation est une composition, on doit également pouvoir intégrer, la table composant dans la table composite.
- *Mapping d'une relation un à plusieurs.* Dans une relation un à plusieurs, le nombre maximum d'objets participant à la relation est de un dans un sens et supérieur à un dans l'autre. Pour implémenter une relation un à plusieurs, une clé étrangère doit être ajoutée dans la table du côté plusieurs vers celle du côté un.
- *Mapping d'une relation plusieurs à plusieurs.* Une relation plusieurs à plusieurs est une relation où le nombre d'objets participant à la relation de chaque côté est supérieur à un. Elle est implémentée au niveau d'une base de données relationnelle à travers la création d'une table d'association qui contient la combinaison des clés primaires des tables participant à la relation.

2.4.1.3 Autres mappings

Hibernate permet également la prise en compte des propriétés dérivées et des propriétés inverses. Pour les propriétés dérivées, une expression de calcul peut être spécifiée dans le fichier de mapping. Les propriétés inverses sont prises en compte dans Hibernate qui utilise les participations de la propriété directe pour déterminer automatiquement les valeurs de son inverse. Les propriétés dérivées et les propriétés inverses ne doivent donc pas être fournies par les applications, elles ne sont également pas stockées dans la base de données ; Hibernate se charge d'initialiser temporairement leurs valeurs lors de l'accès aux données.

2.4.1.4 Le cache objet

Le cache objet est le mécanisme de la couche de persistance qui se charge d'optimiser les temps d'accès à la base de données. Il permet la manipulation des instances en mémoire afin d'éviter le surcoût lié à l'exécution d'une requête et conserve la cohérence des instances générées. Ce cache objet permet essentiellement de répondre aux questions suivantes :

- comment conserver des instances, déjà extraites de la base de données, en mémoire afin d'éviter le surcoût lié à l'exécution d'une requête ;

- au chargement de l'objet adéquat en mémoire, les objets en relations avec celui-ci seront-ils chargés en mémoire ou pas ?

Le cache objet gère aussi les accès concurrentiels aux objets. Il permet ainsi de distinguer les accès en lecture/écriture pour automatiser la mise à jour des instances. Il fournit également un ramasse miette (garbage collector). Pour ce qui du chargement des objets en mémoire le cache objet utilise principalement deux méthodes :

- Le chargement passif (lazy loading) : c'est la technique qui consiste à ne charger que les éléments correspondants à la demande spécifique du programmeur.
- Le chargement agressif (aggressive loading) : c'est la technique qui consiste à charger tous les attributs de l'objet en question et tous les objets (enregistrements) qui ont une relation avec cet objet, provenant de tables différentes.

Le chargement passif est la technique la plus utilisée, c'est également celle que nous avons adopté car elle est en général plus rapide, et moins gourmande en ressources que le chargement agressif.

La notion de cache d'objet en Hibernate est transparente, chaque session contient un cache des objets persistants. Il est utilisé lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant. Les objets persistants sont mis à jour dans la base de données en utilisant les méthodes du cache *save* et *saveAll*, ou bien après la validation de la transaction en cours par un appel à la primitive *commit*.

2.4.1.5 La gestion des transactions

Hibernate n'est pas une base de données, c'est un outil léger de mapping objet relationnel. La gestion des transactions est déléguée à la connexion à base de données sous-jacente. Si la connexion est enregistrée dans JTA (JAVA Transaction API), les opérations effectuées par la session Hibernate sont des parties atomiques de la transaction JTA. Hibernate propose également les fonctionnalités classiques assurant la gestion des transactions (*beginTransaction*, *commit*, *rollback*).

2.4.1.6 Le langage d'interrogation

Hibernate fournit un langage d'interrogation nommé HQL et qui ressemble à SQL. Cependant, HQL est totalement orienté objet : il utilise la notation pointée et intègre les notions d'héritage, de polymorphisme et d'association.

2.4.1.7 SchemaExport / SchemaUpdate / CodeGénérator

Hibernate fournit un certain nombre d'outils parmi lesquels les outils *SchemaExport*, *SchemaUpdate* et *CodeGenerator* que nous avons utilisé.

- L'outil *SchemaExport* permet de générer le schéma de bases de données (fichier « DDL »pour Data Definition Langage), à partir des fichiers de mapping et du fichier de configuration (*hibernate.properties*) où sont définies les informations sur le SGBD cible (dialecte SQL, ...). Le schéma généré inclut les contraintes d'intégrité référentielles (clés primaires et étrangères).
- L'outil *SchemaUpdate* permet de mettre à jour un schéma de bases de données. Notons que si ce dernier possède des données, les mises à jour ne doivent pas invalider les données

préexistantes. Par exemple, un attribut ayant la contrainte « not null » ne peut être ajouté à une entité contenant des données.

– L'outil *CodeGénarator* permet de générer les classes JAVA (ou POJO) permettant de manipuler les données contenues dans la base de données. Un POJO est défini pour chaque entité ou sous-entité définie dans les fichiers de mapping. Chaque POJO déclare les attributs de son entité de référence et fournit un accesseur et un modificateur pour chaque attribut.

2.4.2 Module de Génération des fichiers de mapping

En choisissant parmi les règles formelles de mapping fournies par Hibernate, nous avons écrit un programme générique, destiné à générer les fichiers de mapping permettant de réaliser la correspondance entre la structure d'accès (*Peigne*) et la structure de représentation (*Flatlib*) d'OntoDB.

La figure 4.12 montre l'architecture logicielle du module de génération des fichiers de mapping. Le programme générique *Générer_ Mapping* applique les règles de correspondance aux entités du *Peigne* pour générer les fichiers de mapping *modèle objet/modèle* SQL en XML. Ensuite, l'outil *SchemaExport* va être appliqué à ces fichiers pour créer les tables de la partie ontologie. L'outil *CodeGenerator* appliqué à ces mêmes fichiers de mapping permet de générer les classes JAVA (POJO) pour la partie ontologie qui vont être manipulées par les applications.

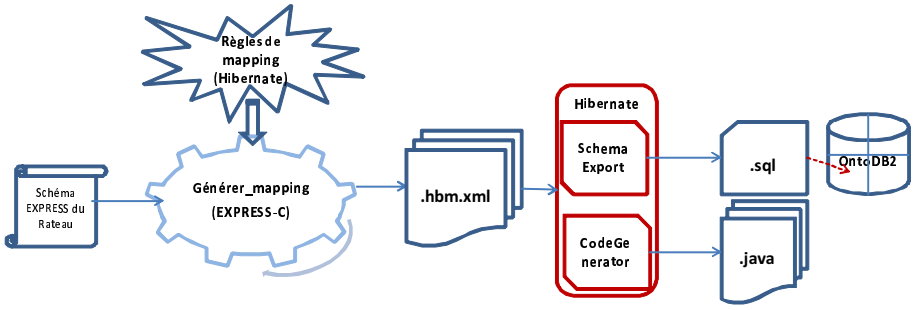


FIG. 4.12 – Module de génération des fichiers de mapping Hibernate

Le programme *Generer_ Mapping* génère les fichiers de mapping XML en respectant la DTD de l'infrastructure Hibernate. Ces fichiers décrivent la correspondance entre le *Peigne* (qui reflète le schéma des classes JAVA) et les tables la base de données relationnelle (reflétées par *Flatlib*) en respectant les règles de mapping d'Hibernate. Pour ce faire, nous adoptons toujours les techniques d'IDM. Le principe du programme générique de génération des fichiers de mapping est le suivant :

1. Toutes les entités du *Peigne* sont récupérées ; pour chaque entité, on fait la correspondance avec sa représentation en base de données (*Flatlib*).
2. Les attributs de chaque entité du *Peigne* sont récupérés, la correspondance entre les attributs de chaque entité est décrite selon leurs types et les colonnes de la base de données.

Nous présentons ci-dessous les différentes correspondances entre le *Peigne* et la structure des tables.

2.4.2.1 Traitement des hiérarchies

Nous commençons par décrire la correspondance entre les entités du *Peigne*, générées par le programme *generer_Peigne* (cf. section 2.3), et les tables de la base de données relationnelle en utilisant la balise <*hibernate-mapping*>. La correspondance se fait selon les règles de passage d'Hibernate. Le processus de traduction suit l'algorithme suivant :

1. Chaque entité racine du *Peigne* est décrite dans un fichier de mapping, qui porte le nom de l'entité avec l'extension *.hbm.xml*. Nous déclarons dans ce fichier, la version XML utilisée et l'emplacement de la DTD externe d'Hibernate qui doit être vérifiée pour que le fichier soit valide.
2. Chaque sous-entité est décrite dans le fichier de l'entité racine de sa hiérarchie.
3. Pour chaque sous-entité, nous déclarons la correspondance entre la clé primaire de la base de données, l'identifiant de la classe et le mode de génération.

Rappelons que pour représenter une hiérarchie dans OntoDB2, nous avons adopté la solution « une table par hiérarchie avec une colonne discriminante ». Par contre, nous avons représenté chaque hiérarchie par un schéma objet à un niveau de profondeur. Nous décrivons donc, dans le fichier XML, la correspondance entre ces deux représentations. Ceci nécessite d'utiliser les balises suivantes (cf. figure 4.13) :

- La balise <*class*> sert à faire la correspondance entre la classe racine et la table de la base de données OntoDB2, en spécifiant les valeurs des attributs suivants :
  - *name* pour indiquer le nom de la classe racine,
  - *table* pour indiquer le nom de la table représentant l'ensemble de la hiérarchie dans la base de données,
  - *discriminator-value* pour indiquer la valeur que prend la colonne discriminante lors de l'instanciation de la classe racine au cas ou la classe racine est concrète,
- La balise <*discriminator*> indique que la solution adoptée est « une table par hiérarchie avec une colonne discriminante », et déclare les deux attributs :
  - *column* qui indique le nom de la colonne discriminante (la colonne discriminante, dans notre cas, est RTYPE),
  - *type* qui indique le type de la colonne discriminante dans la base de données (le type de la colonne discriminante, dans notre cas, est STRING).

Nous décrivons après les balises déclarées précédemment, la correspondance entre les propriétés de la classe racine (y compris l'identifiant de la classe) et les colonnes de la base de données. Pour déclarer les sous-types concrets, nous utilisons la balise <*subclass*> en spécifiant les valeurs des deux attributs ci-dessous :

- *name* pour indiquer le nom de la classe concrète JAVA,
- *discriminator-value* pour indiquer la valeur que prend la colonne discriminante lors de l'instanciation de cette classe.

Ensuite, nous décrivons la correspondance entre les propriétés propres à la classe concrète et les colonnes de la base de données. La correspondance des attributs dépend de leurs domaines de valeurs.

La figure 4.13 montre un extrait d'un fichier de mapping, en particulier, cette figure illustre le mapping d'un héritage avec la stratégie une table par hiérarchie.

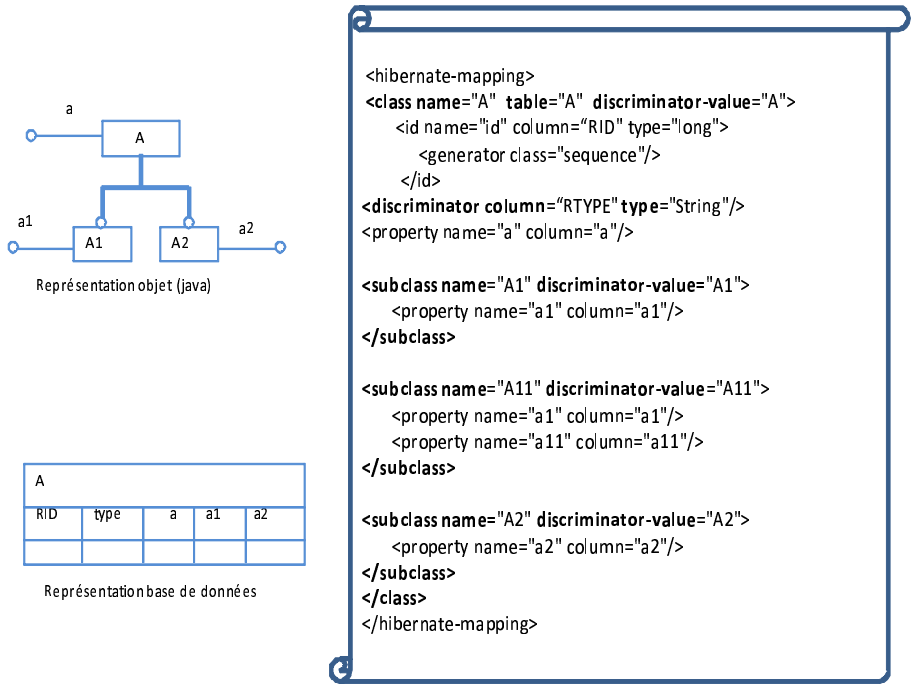


FIG. 4.13 – Mapping de l'héritage dans OntoDB2

2.4.2.2 Traitement des attributs

La correspondance entre les attributs et les colonnes des tables sont décrites ci-dessous.

Attribut monovalué de type simple

Pour un attribut monovalué de type simple, nous utilisons la balise <*property*> en spécifiant les valeurs des attributs suivants :

- *name* pour indiquer le nom de l'attribut dans la classe JAVA,
- *column* pour indiquer le nom la colonne correspondante dans la base de données,
- *type* pour indiquer le type JAVA de l'attribut,
- *length* pour indiquer la taille (en base de données) pour un l'attribut de type chaîne de caractère.

Le tableau 4.3 ci-dessous montre la correspondance entre les types simples EXPRESS et les types simples JAVA.

EXPRESS	JAVA
string	String
boolean	boolean
integer	int
real	float
binary	bit

TAB. 4.3 – Correspondance entre types simples EXPRESS et JAVA

Attribut multivalué de type simple

Pour un attribut multivalué c'est-à-dire une collection de type simple, nous utilisons selon le cas la balise <*set*>, <*list*>, <*bag*> ou encore <*array*> en précisant les valeurs des attributs suivants (cf. figure 4.14) :

- *table* pour indiquer le nom de la table représentant l'attribut,
- *column* de la sous balise <*key*> pour indiquer l'identifiant de l'instance référencée. Dans la figure 4.14, la colonne de base de données *A_*RID de la table *A_ _ a2*, est une clé étrangère qui référence la colonne RID de la table *A*.
- *column* de la sous balise <*element*> pour indiquer une valeur de l'attribut pour cette instance,
- *column* de la sous balise <*index*> pour indiquer le rang de cette valeur dans le cas où la balise principale est <*list*> ou <*array*>.

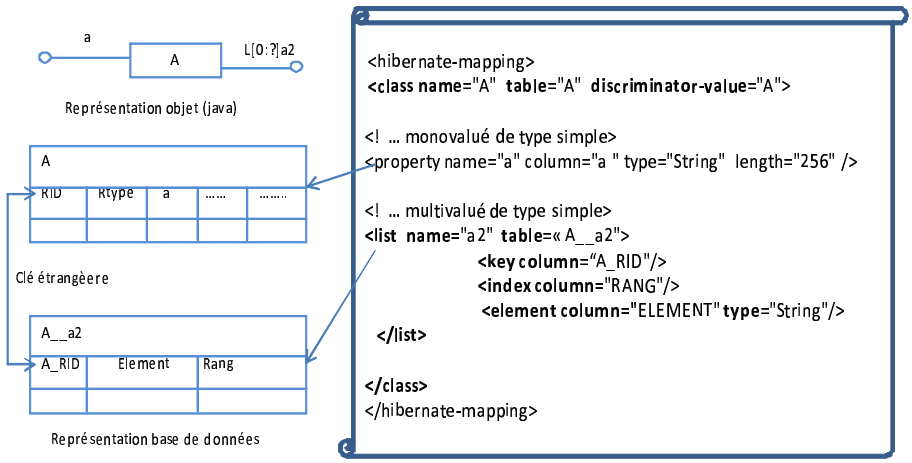


FIG. 4.14 – Mapping des attributs de types simples

Attribut de type objet monovalué

Un attribut de type objet monovalué est considéré comme une relation un-vers-un dans la base de données. Nous utilisons la balise <*one-to-one*> en spécifiant les valeurs des attributs *name*, *column* et *class* qui indique la table de l'entité co-domaine de l'attribut(clé étrangère).

Attribut de type objet multivalué

Pour la correspondance d'un attribut de type objet multivalué, nous devons tenir compte de l'existence éventuelle d'un inverse afin de sélectionner la règle de correspondance la plus appropriée. Nous distinguons ainsi les deux cas suivant :

1. **Pas d'inverse ou inverse multivalué**. L'attribut est considéré comme ayant une association plusieurs-vers-plusieurs dans la base de données. Le mapping est défini comme pour un attribut de type simple multivalué à la différence que la sous balise <*element*> est remplacée par la sous balise <*many-to-many*> en spécifiant les valeurs des attributs :
   - *class* qui indique la table de l'entité co-domaine,
   - *column* qui précise la colonne (identifiant) référencée dans cette table.

2. **Inverse monovalué**. L'attribut est considéré comme ayant une association *un-vers-plusieurs* dans la base de données. Le mapping est défini comme ci-dessus mais cette fois ci avec la sous-balise <*many-to-one*> en spécifiant la valeur de l'attribut *class* et *column*.

La figure 4.15 illustre le mapping d'attributs d'objets.

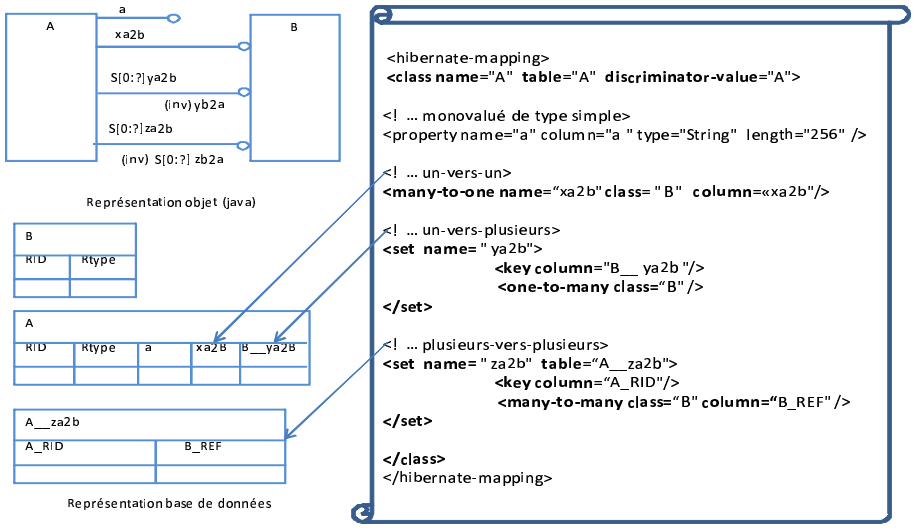


FIG. 4.15 – Mapping des attributs de type objet

2.4.2.3 Traitement des types spécifiques

Types enumérés

Chaque type énuméré est décrit dans un fichier de mapping. La table correspondante dans la base de données comporte deux colonnes : une colonne pour l'identifiant (nommée RID) et une colonne pour les valeurs (nommée VALUE) du type énuméré.

Types nommées

Le type nommé n'a pas de correspondance explicite dans le langage JAVA, tout attribut de type nommé, sera traité comme un attribut du type de base qu'il redéfinit.

2.4.2.4 Méta-descripteurs

Les méta-descripteurs sont utilisés pour décrire les entités de façon informelle. Ces derniers peuvent être définis dans un nombre quelconque de langues. Tout méta-descripteur sera dupliqué en autant d'attributs de types chaîne de caractères, que de langues existantes dans la base de données. Des propriétés de type booléen sont rajoutées pour indiquer les langues existantes dans la base de données (cf. figure 4.16). Les langues de la BDBO devant être connues au moment de la génération des fichiers de mapping, nous avons défini le français (*fr*) et l'anglais (*en*) comme langues par défaut.

La figure 4.16 ci-dessous montre un extrait d'un fichier de mapping définissant la correspondance des descripteurs.

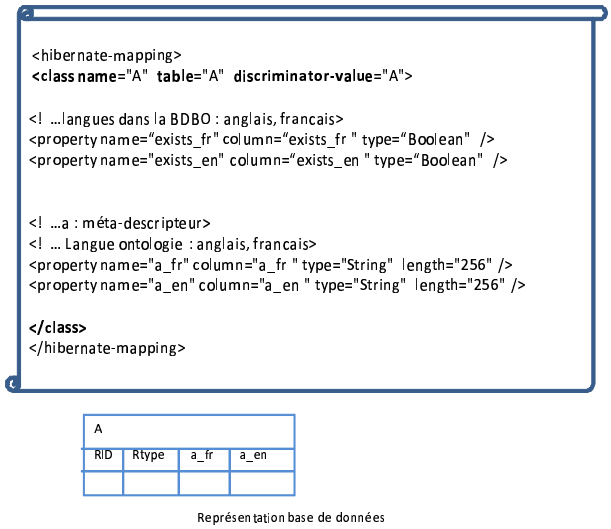


FIG. 4.16 – Mapping des méta-descripteurs

Une fois les fichiers de mappings crées, le script des tables est obtenu en utilisant l'outil hibernate *SchemaExport*. Ce dernier utilise le fichier *hibernate.properties* qui contient les informations sur le SGBD cible. L'outil *CodeGenerator* est quant à lui utilisé pour générer les POJO d'accès pour les applications.

2.5 API d'accès aux ontologies

L'API d'accès JAVA *OntoLib*, générée à partir de l'outil *CodeGenerator* possède la même structure que le *Peigne* (un niveau de hiérarchie).

1. Chaque entité racine du *Peigne* est représentée par une classe JAVA portant le même nom. Dans cette classe sont déclarés l'identificateur RID (clé primaire) de l'entité dans la base de données, et les propriétés communes à toutes les classes concrètes. Ces propriétés sont privées (mot clé *private* en JAVA), et ne sont accessibles que par des méthodes définies au niveau de cette classe et de ses sous-classes. C'est la notion d'encapsulation des données.
2. Les sous-types d'une entité racine sont représentés par des classes JAVA, et héritent des propriétés et méthodes de la classe correspondant à l'entité racine en utilisant le mot clé *extends*. Dans chaque sous-classe sont déclarées les propriétés qui doivent être valuées lors de son instanciation, et qui ne sont pas présentes dans la classe racine. Comme les propriétés de la classe racine, les propriétés de chaque classe concrète sont privées, et ne sont accessibles que par des méthodes définies au niveau de cette classe.
3. Des constructeurs sont générées afin de permettre d'instancier les objets des classes.
4. Une méthode « accesseur » et une méthode « modificateur » sont générées pour chaque attribut afin d'accéder aux valeurs des propriétés de la classe ou pour les modifier.

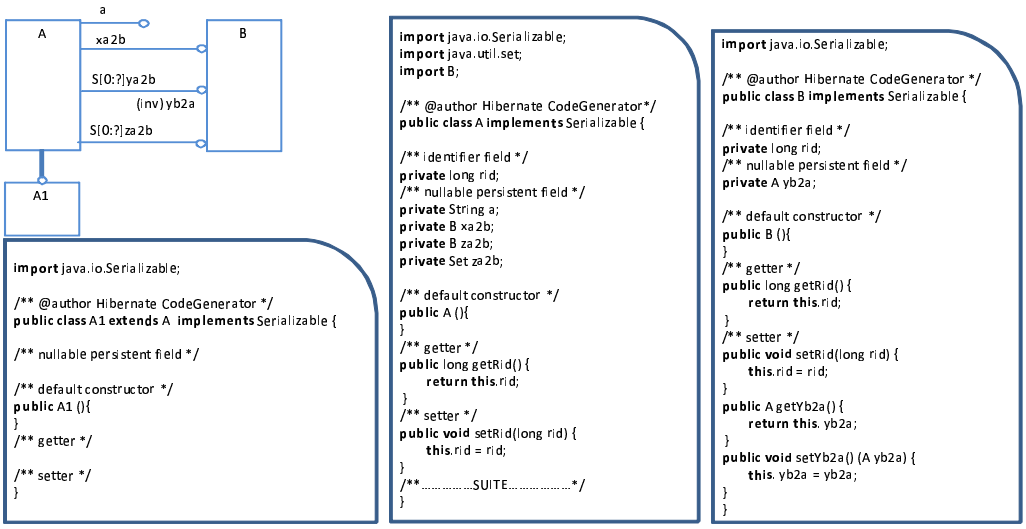


FIG. 4.17 – Exemple de POJO pour l'accès à la partie ontologie.

La figure 4.17 illustre un exemple de classe JAVA généré pour la partie ontologie.

Une fois la structure des tables et les classes de manipulation de la partie ontologie définies, nous allons à présent alimenter la partie ontologie d'OntoDB2 en important des ontologies dans cette dernière.

2.6 Importation des Ontologies

La BDBO OntoDB2 est fondée sur un formalisme d'ontologie permettant d'offrir un support pour des ontologies issues de différents formalismes d'ontologies. Il est donc nécessaire de définir des modules permettant d'importer les ontologies de leur format d'origine vers la partie ontologie de OntoDB2. Nous présentons ici les modules d'importation que nous avons mis en œuvre pour les ontologies PLIB et OWL Lite.

2.6.1 Ontologie PLIB

Le formalisme d'ontologies sur lequel se fonde OntoDB2 est proche de PLIB, la correspondance entre ces deux structures est donc immédiate. Par contre, la structure des tables de la partie ontologie d'OntoDB2 est semblable au schéma *Flatlib* disponible en EXPRESS. Les ontologies PLIB étant disponibles sous forme d'instances de fichiers physiques P21, nous avons donc à réaliser, en utilisant des techniques d'IDM, la transformation des ontologies PLIB, de la structure initiale vers la structure plate *Flatlib*, en nous aidant du schéma de correspondance présenté à la section 2.2.3. Une fois les ontologies PLIB disponibles sous-forme d'instances de *Flatlib*, nous avons, à l'aide d'un autre programme générique, transformé à leur tour ces dernières sous une forme facilement exportable en bases de données : le format CSV. Ce dernier est optimisé pour le chargement de grands volumes de données dans les bases de données. En effet, la majorité des SGBD offrent un mécanisme de chargement par lots (*Bulk Load*) efficace des données à partir de fichiers CSV.

La figure 4.18 présente l'architecture logicielle du module d'importation des ontologies PLIB

dans OntoDB2. A partir d'une ontologie fournie en entrée, des schémas PLIB, *Flatlib* et du schéma de correspondance entre ces deux derniers schémas, le module *Générer_instances_plat* transforme les instances du schéma PLIB en des instances du schéma *Flatlib*. Ces derniers sont ensuite fournies comme paramètre au module *Générer_*CSV, qui en appliquant des règles de transformation que nous avons défini, génère en sortie une représentation de ces instances dans des fichiers au format CSV.

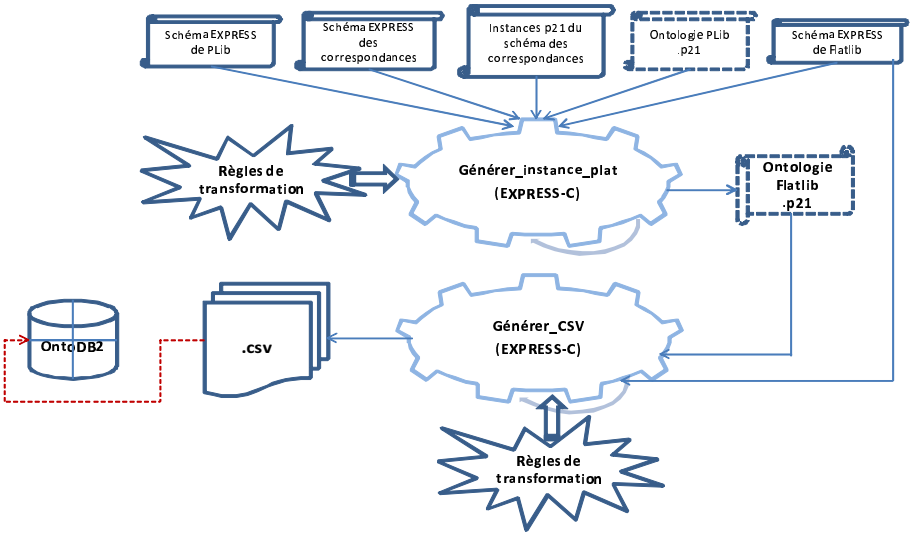


FIG. 4.18 – Module d'importation d'ontologies PLIB

2.6.1.1 Algorithme de transformation d'instances PLIB en *Flatlib*

1. Déterminer les entités concrètes de la structure initiale du formalisme d'ontologie PLIB remontées dans chaque racine de *Flatlib*.
2. Pour chacune de ces entités, déterminer la liste des attributs à valuer.
3. Créer une instance de la racine *Flatlib* pour chaque instance d'entité concrète.
4. Initialiser la valeur de l'attribut discriminant dans *Flatlib* au nom de l'entité concrète.
5. Pour chaque attribut valué dans l'instance de l'entité PLIB :
   - déterminer l'attribut correspondant dans la racine *Flatlib* ;
   - si c'est un attribut simple initialiser sa valeur ;
   - sinon si c'est un attribut d'objet, initialiser sa valeur avec l'instance correspondante dans *Flatlib* si cette dernière a déjà été transformée ; sinon marquer la référence comme non résolue.
6. Résoudre les références non résolues au premier passage.

2.6.1.2 Algorithme de création des fichiers CSV

La transformation des instances de *Flatlib* du format P21 au format CSV est réalisée suivant le principe suivant :

1. Chaque type énuméré est décrit dans un fichier CSV. L'entête de ce fichier comporte deux colonnes : une colonne identifiant (*rid*) et une colonne pour les valeurs du type énuméré (*value*).
2. Chaque entité de *Flatlib* est décrit dans un fichier CSV qui porte son nom et dont l'entête est constitué d'une colonne identifiant représentant la clé dans la base de données (*rid*), une colonne discriminant (*rtype*), et les noms des attributs monovalués de l'entité *Flatlib*.
3. Chaque attribut multivalué est décrit dans un fichier CSV. L'entête de ce fichier comporte deux colonnes : une colonne identifiant l'instance et une colonne pour les valeurs de la collection. Si l'attribut est de type objet, cette deuxième colonne contient l'identifiant (base de données) de l'individu référencé.
4. Les lignes des fichiers CSV des entités et des attributs multivaluées sont initialisées avec les valeurs utilisées pour décrire les instances (pour chaque instance de classe, l'attribut *is_primitive* est initialisé à vrai et l'attribut *origin* à 'PLIB').

2.6.2 Ontologie OWL Lite

Pour l'importation d'ontologie OWL, nous avons réalisé un module JAVA (cf. figure 4.19) et, utilisé l'API de source libre Jena [30] qui permet de manipuler les ontologies OWL décrites dans des fichiers au format RDFS/XML. Le module que nous avons programmé se base sur un ensemble de règles de correspondance entre OWL Lite et le *Peigne* qui est le schéma d'accès pour les applications. Nous présentons ci-dessous ces correspondances pour les constructions supportées par OntoDB2.

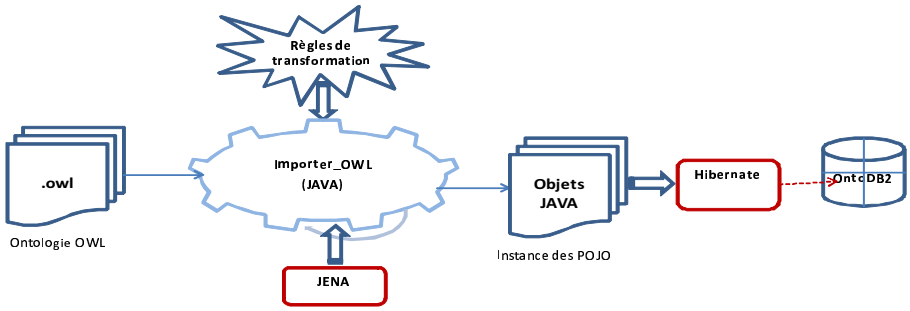


FIG. 4.19 – Module d'importation d'ontologies OWL

Notons qu'une instance de la classe *Class* du *Peigne*, de nom préféré OWLR*ootClass*, est créée pour toute ontologie OWL. Cette classe correspond à la racine implicite OWL *:Thing* de cette ontologie. La classe OWLR*ootClass* est caractérisée par une propriété URI permettant l'identification universelle des instances comme c'est le cas dans le formalisme OWL.

2.6.2.1 Classe

Nous présentons ci-dessous la mise en correspondance des classes nommées.

[30] http ://sorceforge/jena.org

Classe primitive

Toute classe primitive OWL Lite est mise en correspondance avec la classe *Class* du *Peigne* en initialisant l'attribut *is_primitive* à vrai et l'attribut *origin* à 'OWL'.

Classe intersection

Comme nous l'avons vu lors de la spécification d'OntoDB2, l'intersection est interprétée en OWL *l*ite comme l'héritage multiple. Cependant dans le cadre d'hypothèses que nous avons adopté pour OntoDB2, l'héritage multiple n'est pas autorisé. Toutefois, nous avons également vu que dans certains cas de figure, les instances de classes intersection pouvaient être converties et représentées dans OntoDB2. Dans ce contexte, toute classe intersection est mise en correspondance avec la classe *Class* du *Peigne* en spécifiant la vue ontologique qui lui correspond, et en initialisant l'attribut *is_primitive* à faux et l'attribut *origin* à 'OWL'.

2.6.2.2 Héritage

Rappelons que nous ne supportons actuellement que l'héritage simple. Nous distinguons trois cas pour la mise en correspondance de l'héritage

Pas de superclasse

Si une classe ne possède pas de superclasse, sa correspondante est déclarée comme sous-classe de la classe racine OWLRootClass.

Héritage simple d'une classe nommée

Si une classe possède une seule superclasse nommée, sa correspondante est déclarée comme sous-classe de la correspondante de sa superclasse.

Héritage de classe anonyme (de type restriction)

- Si une classe possède une superclasse anonyme de type restriction, et si la propriété sur laquelle porte la restriction a comme domaine cette même classe, alors il s'agit de la spécification (1) soit du co-domaine de la propriété (cas d'une restriction *owl :allValuesFrom*), (2) soit de la spécification des cardinalités de la propriété (cas d'une restriction *owl :minCardinality, owl :maxCardinality, owl :cardinality*). Dans le premier cas, le co-domaine de la correspondante de la propriété restreinte est initialisé à la correspondante de la classe spécifiée dans la clause *allValuesFrom*. Dans le second cas, les cardinalités de la correspondante de la propriété restreinte sont initialisées aux valeurs spécifiées dans la restriction.
- Dans les autres cas, il d'agit d'une redéfinition locale de co-domaine ou des cardinalités d'une propriété dans une sous-classe de son co-domaine. Nous ne proposons pas de correspondance dans ces cas. Notons cependant que ces informations pourraient être utilisées dans la base de données pour vérifier des contraintes sur les DBO en utilisant par exemple la clause SQL CHECK.

2.6.2.3 Propriété

Nous présentons ci-dessous la mise en correspondance des propriétés.

Propriété de type simple

Une propriété de type simple (de domaine de valeurs : chaîne de caractère, entier, réel, booléen, ...) est mise en correspondance comme une propriété autonome et son co-domaine est initialisé au type simple correspondant du *Peigne*. La tableau 4.4 montre la correspondance entre les types simples XML et les types simples du formalisme d'ontologie.

OWL	ontoDB2
boolean	Boolean_type
double, float	Real_type
integer, nonNegativeInteger, positiveInteger, long, int, unsignedInt, short, unsignedShort, byte, unsignedByte	Int_type
string, normalizedString, token, NMToken,	String_type

TAB. 4.4 – Correspondance entre les types simples OWL et ontoDB2

Pour une propriété de domaine de valeurs défini par le constructeur *owl :Datarange*, selon que ce dernier soit une énumération d'entiers ou de chaînes de caractères, le co-domaine de la propriété sera une instance de l'entité *non_ quantitative_ int_ type* ou *non_ quantitative_ code_ type*.

Propriété de type objet

Une propriété de type objet est mise en correspondance comme propriété autonome du *Peigne*. Son co-domaine est défini par le constructeur *class_ instance* et référence la classe correspondante du co-domaine de la propriété.

2.6.2.4 Caractéristique de propriété

Comme nous l'avons vu au chapitre 3, les caractéristiques de propriétés ont été représentées dans le formalisme d'ontologie. Intégrer les caractéristiques de propriété a consisté à étendre l'entité *Property* avec de nouveaux attributs.

Fonctionnelle

Une propriété ayant la caractéristique fonctionnelle est définie selon son type (simple ou objet) comme présenté à la section 2.6.2.3. Dans le cas contraire, sa correspondante est déclarée comme étant multivaluée. Son co-domaine est défini par le constructeur *aggregate_ type*.

Symétrique

Pour une propriété ayant la caractéristique symétrique, l'attribut *is_ symmetric* de sa correspondante est initialisée à vrai.

Transitive

Pour une propriété ayant la caractéristique transitive, l'attribut *is_ transitive* de sa correspondante est initialisée à vrai.

Inverse

Pour une propriété ayant la caractéristique inverse, l'attribut *inverseOf* de sa correspondante est initialisée à la correspondante de son inverse.

2.6.2.5 Domaine/co-domaine

Les propriétés OWL ne précisent pas toujours explicitement leur domaine ou leur co-domaine ; cependant, il est possible de les déduire partir des caractéristiques de propriétés [44].
En effet :

- le domaine et le co-domaine d'une propriété symétrique ou transitive sont identiques ;
- le domaine et co-domaine d'une sous-propriété sont inclus dans ceux de sa super-propriété.
- deux propriétés inverses l'une de l'autre, ont également leur domaine et leur le co-domaine inversés.

2.6.2.6 Identifiants

Les identifiants (URI) des concepts seront mis en correspondance avec l'attribut *code* (qui est une composante du BSU dans PLIB). Le nom local des concepts sera mis en correspondance comme étant le nom préféré (*preferred_ name*) pour la langue courante de l'ontologie.

2.6.2.7 Méta-descripteurs

Nous présentons ici la mise en correspondance des méta-descripteurs des classes et des propriétés. Si ces derniers sont traduits, ils le seront pour les langues prévues dans la base de données.

Labels

- Tous les « premiers »labels de chaque langue (à l'exception du label de la langue courante) seront mis en correspondance avec la propriété *preferred_ name* de la même langue.
- Les autres seront mis en correspondance avec la propriété *synonymous_ name* de la langue correspondante.

Commentaires

- Tous les « premiers »commentaires de chaque langue seront mis en correspondance avec la propriété *definition* de la même langue.
- Les autres seront mis en correspondance avec la propriété *remark* de la langue correspondante.

Notons que les algorithmes pour la transformation des méta-descripteurs ne sont pas déterministe ; à l'issu de deux interrogations identiques avec le raisonneur Jena par exemple, les méta-descripteurs peuvent apparaître dans deux ordres différents.

2.6.2.8 Algorithme d'importation d'ontologies OWL

L'algorithme d'importation des ontologies OWL, se basant sur les règles décrites ci-dessus est donc définit comme suit.

- Chaque classe OWL est mise en correspondance avec une classe OWLClass.

- Les attributs *syntactic_ description*, *has_ ontological_ view* et *ontological_ view* sont initialisés respectivement à la description textuelle (au format RDF/XML) de la classe ; à la valeur vrai si la classe est une classe non canonique et dans ce cas, l'attribut *ontological_ view* est initialisé à la définition de la vue ontologique correspondante à la classe.
- Si une classe ne possède pas de superclasse, sa correspondante est définie comme une sous-classe de la classe OWLRootClass. Sinon, sa correspondante est définie comme une sous-classe de la correspondante de sa superclasse.
- Les propriétés de chaque classe sont mises en correspondances avec des propriétés du *Peigne* en spécifiant suivant les cas, les attributs correspondant à certaines caractéristiques.
- Si le domaine d'une propriété ne peut être déduit, le domaine de sa correspondante est initialisé avec la classe OWLRootClass.
- Si le co-domaine d'une propriété de type objet ne peut être déduit, le domaine de sa correspondante est initialisé avec la classe OWLRootClass.

Les vues associées aux classes sont définies suivant les règles décrites à la section 4.3 du Chapitre 3.

3 Synthèse sur les parties Méta-schéma et Ontologies

Nous avons présenté ci-dessus, les différents modules développés pour la mise en œuvre de la partie Ontologie. La persistance des ontologies est assurée par l'intermédiaire du framework Hibernate. Grâce à la possibilité de paramétrer les caractéristiques de SGBD support, cette approche permet ainsi d'assurer la portabilité des parties méta-schéma et ontologie de la BDBO OntoDB2 sur d'autres SGBD.

Notons cependant que, quelques triggers ont été définis dans la partie ontologie. Ces triggers permettent en particulier d'indexer la relation de subsumption, et de définir des valeurs par défaut pour certaines colonnes. Ces triggers pourraient toutefois être remontés du coté applicatif, ce qui permettrait de garantir la portabilité totale pour la partie ontologies.

Les différents modules que nous avons développé ont été réalisés en utilisant des techniques d'IDM. Ce choix s'avère approprié dans notre cas car le formalisme d'ontologie de la BDBO OntoDB2 est susceptible d'être étendu suivant les besoins.

L'évolution du formalisme d'ontologie n'est actuellement pas dynamique. En effet, le choix des constructions nécessaires pour une application donnée doit être réalisé sur le modèle en *Peigne* avant la phase de génération des fichiers de mapping qui servent de base à la définition de la structure des tables de la partie ontologie et des POJO de gestion. Une fois les fichiers de mapping générés, toute évolution du formalisme d'ontologie, bien que n'ayant pas d'impact sur les modules développés, nécessite de re-générer les fichiers de mapping.

Le langage de requête ontologique OntoQL est actuellement en cours de modification pour permettre l'extension dynamique du formalisme d'ontologie et des types de données. Nous envisageons également de définir une interface dynamique permettant de réaliser les extensions. Dans tous les cas, les extensions apportées ne peuvent pas invalider des ontologies pré-existantes de la BDBO. Par exemple, l'ajout d'un attribut ayant la contrainte d'être obligatoire, ne peut pas réaliser sur la BDBO si l'entité étendue possède des instances.

Comme nous l'avons mentionné plus haut, les modules définis pour la partie ontologie sont

réutilisés en l'état pour la partie méta-schéma. A la différence du formalisme d'ontologie PLIB, le méta-schéma ne comporte que très peu d'entités (16 entités et 1 type défini). Le méta-schéma a donc été utilisé à l'identique, sans mise à plat de sa structure (avec la stratégie de mapping « une table par classe »). La figure 4.20 présente l'architecture logicielle globale de mise en œuvre de la partie méta-schéma.

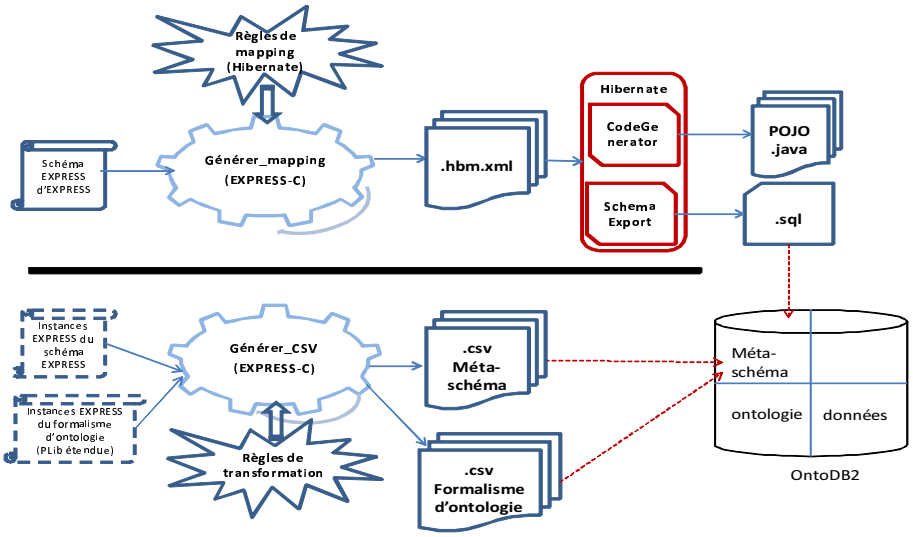


FIG. 4.20 – Architecture logicielle de mise en œuvre de la partie méta-schéma

Nous présentons dans la section suivante la mise en œuvre de la partie données d'OntoDB2.

4 La partie données

Développer la partie données de l'architecture OntoDB2 consiste à concevoir les modules permettant de :

- définir la structure des tables : c'est-à-dire le schéma de représentation des DBO (par exemple horizontal ou binaire) ;
- définir un schéma d'accès aux DBO canoniques et non canoniques ;
- définir une méthode d'accès aux données ;
- importer les instances des ontologies existantes dans la partie données d'OntoDB2.

4.1 Schéma de représentation des données

Le stockage des instances de classes dans OntoDB2 se fait en générant les requêtes de création des différentes tables de la base de données. La génération de ces requêtes passe par la spécification de règles permettant de représenter les classes et types de l'ontologie obtenu dans le SGBD support. Nous présentons, ci dessous, les règles de correspondances que nous avons utilisées pour la génération de la structure de la partie donnée, suivant l'approche de représentation horizontale (cf. section 3.2.3 du chapitre 2), et suivant l'approche de représentation binaire (cf. section 3.2.2

du chapitre 2). Ensuite, nous présentons comment sont prises en compte les caractéristiques associées aux propriétés.

4.1.1 Approche horizontale

Nous présentons ci-dessous la correspondance entre les instances de l'ontologie et l'approche de représentation horizontale dans une base de données relationnelle.

4.1.1.1 Classe

Chaque classe correspond à une table au niveau de la base de données relationnelle. Cette table possédera une clé primaire de type entier générée par le SGBD (cette clé est nommée *rid*). Les autres champs de la table dépendront du type des propriétés applicables de la classe sélectionnée pour définir son extension et seront donc étudiés dans les sections suivantes.

4.1.1.2 Propriété autonomes

Les propriétés autonomes (à l'exception des propriétés de type *Level_type*) d'une classe sont représentées comme des champs de la table correspondant à la classe dans laquelle ils sont définis ou des clés étrangères ou encore des tables d'associations.

Propriété simple

Toute propriété de type simple (de cardinalité 0 :1 ou 1 :1) est représentée dans la table correspondant à la classe qui la définit comme un champ (une colonne). Son type sera un type de base du SGBD utilisé, correspondant au type atomique OntoDB2.

La table 4.5 montre la correspondance entre les types simples du formalisme d'ontologie dOntoDB2 et les types simples SQL.

OntoDB2	SQL
String_type, Non_Quantitatine_Int_Type	varchar
Int_type, Int_Currency_type, Int_Measure_type	bigint
Real_type, Real_Currency_type, Real_Measure_type	float
Boolean_type	boolean

TAB. 4.5 – Correspondance entre les types simples d'OntoDB2 et SQL

Propriété collection de type simple

Pour chaque propriété autonome de type *collection* (ensemble, sac, tableau, liste), une table d'association portant le nom de la classe concaténé au nom de la propriété (*classe__propriete*) est créée. Les champs sont : une clé étrangère référençant la table correspondant à la classe définissant la propriété. Un second champ contenant chacune des valeurs de la collection. Dans le cas d'une liste ou d'un tableau, Un champ *rang* est rajouté. Ce dernier fait partie de la clé de la table et permet de conserver l'ordre des éléments de la collection.

Propriété géométrique

Toute propriété de type géométrique (de cardinalité 0 :1 ou 1 :1) est représentée dans la table correspondant à la classe qui la définit comme un champ (une colonne). Son type est un type géométrique de base de l'extension géométrique PostGIS du SGBD PostgreSQL.

La table 4.6 montre la correspondance entre les types géométriques dans OntoDB2 et les types de PostGIS[31] (l'extension géométrique de PostgreSQL).

OntoDB2	PostGIS
Geometry_Type	geometry
Linestring_Type	linestring
Line_Type	line
Linearring_Type	linearring
Polygon_Type	Polygon
Geometrycollection_Type	geometrycollection
Multipoint_Type	multipoint
Multilinestring_Type	multilinestring
Multipolygon_Type	multipolygon

TAB. 4.6 – Correspondance des types géométriques d'OntoDB2 et PostGIS

Propriété de type objet

Le problème qui se pose avec les propriétés de type objet est la mise en œuvre du polymorphisme. En effet, losrqu'une propriété a pour type *class_ instance_ type*, elle peut référencer des instances de n'importe quelle sous-classe de la hiérarchie du co-domaine de la propriété.

Nous proposons d'associer à chaque propriété de type objet, une colonne qui précise le nom de la table associée à la classe de l'instance. Ainsi, toute propriété référençant une classe (sans être une collection) est représentée dans la table correspondant à la classe qui la définit comme une clé étrangère référençant la clé primaire de la table correspondant à la classe référencée. Le nom de la colonne sera suffixé par la chaîne « _ref ». Une colonne supplémentaire, suffixée par la chaîne « _ref_tablename », sert à spécifier la table associée à l'instance référencée.

Propriété collection de type objet

Les collections de type objet (à l'exception des propriétés d'ordre) sont mises en correspondance en tenant compte des cardinalités directes et inverses des classes impliquées dans la relation.

Cas de collection de type objet représentant une association 1 :N

Toute propriété de type collection (ensemble, sac, tableau, liste) représentant une association 1 :N, est représentée dans la table correspondant à la classe référencée par une clé étrangère référençant la clé primaire de la table correspondant à la classe référençante, c'est à dire celle qui définit la propriété. Une colonne suffixée par la chaîne « _ref_tablename », sert à spécifier la table associée à l'instance référencée.

[31] hppt ://www.PostGIS.fr

Cas de collection de type objet représentant une association N :M

Pour chaque propriété de type collection représentant une association N :M (ensemble, sac, tableau, liste), on créera une table d'association portant le nom de la classe concaténé au nom de la propriété (*classe_ _propriete*). Les différents champs de cette table sont : une clé étrangère référençant la table correspondant à la classe domaine de la propriété. Le second champ contient l'identifiant de l'instance cible et le troisième champ sert à spécifier la table qui contient cette instance. Les noms de ces champs seront suffixés comme ci-dessus. La clé primaire de la table est constituée des deux premiers champs.

Dans le cas d'une liste ou d'un tableau, un champ *rang* est rajouté. Ce dernier fait partie de la clé de la table et permet de conserver l'ordre des éléments de la collection.

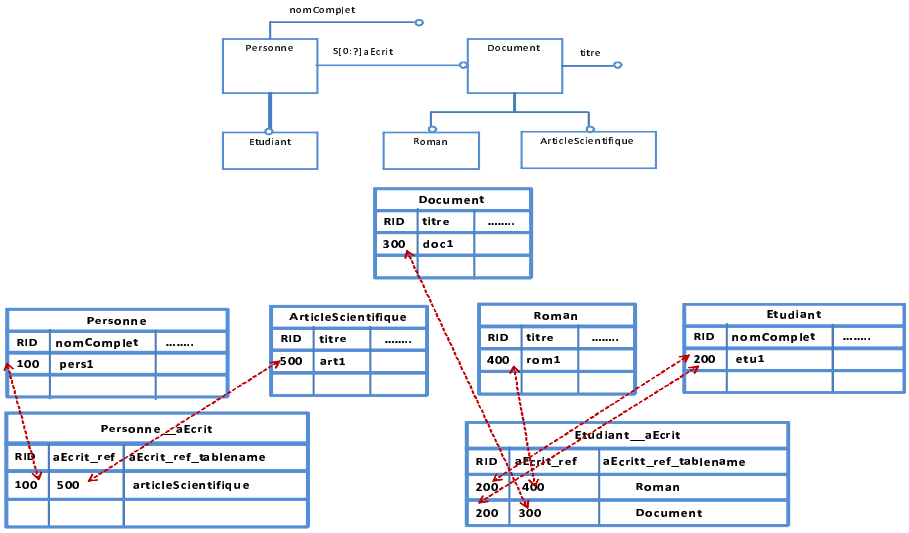


FIG. 4.21 – Représentation d'une collection d'objet dans la partie donnée

Propriétés dépendantes

Une propriété dépendante est représentée comme une collection simple. A la table associée est rajoutée une colonne pour chacun des paramètres de contexte dont elle dépend. Les colonnes des paramètres de contexte font partie de la clé de la table.

La figure 4.22 illustre la représentation d'une propriété dépendante.

Propriétés de type Level type

Le type *Level_ type* permet de qualifier des valeurs d'un type quantitatif, Une valeur de *level_ type* comporte quatre valeurs (optionnelles) caractérisées par les qualificatifs : *min* (minimal), *max* (maximal), *nom*(nominatif) et *typ* (typical). Pour une propriété de type *Level_ type*, une colonne sera créée pour tout qualificatif utilisé pour décrire les instances. Le nom de chaque colonne est défini en suffixant la propriété par : *_ min*, *_ max*, *_ nom* ou *_ typ* suivant l'élément de type *Level* utilisé dans la description des instances.

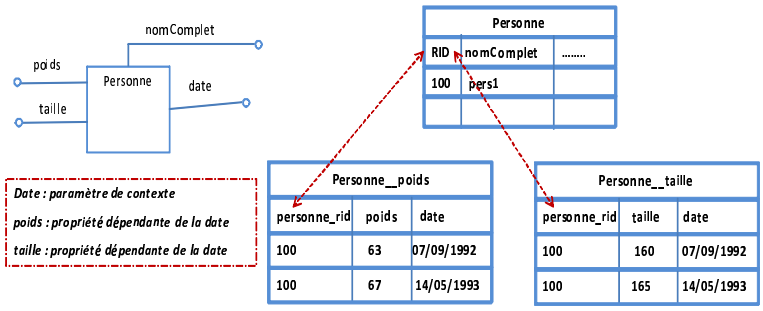


FIG. 4.22 – Représentation de propriété dépendante dans la partie donnée

4.1.2 Approche binaire

Nous présentons ci-dessous la correspondance entre les instances de l'ontologie et l'approche de représentation binaire dans une base de données relationnelle. Cette approche n'est pas encore validée. Nous avons choisi de mettre en œuvre la variante NOISA qui définit une table unaire pour chaque classe et une table binaire pour chaque propriété et ce sans héritage de table (cf. section 3.2.2 du chapitre2).

4.1.2.1 Classe

Chaque classe correspond à une table au niveau de la base de données relationnelle. Cette table possède une colonne de type entier (nommée *rid*) générée par le SGBD qui sert d'identifiant d'instances.

4.1.2.2 Propriété de type simple

Chaque propriété autonome de type simple (entier, réel, booléen, chaîne de caractères) ou de type géométrie est mise en correspondance par une table d'association portant le nom de la classe concaténé au nom de la propriété (*classe__propriete*). Les champs de cette table sont : une colonne nommée *rid* (clé primaire de la table) représentant l'identifiant de l'instance, une colonne contenant les valeurs de propriété. Pour gérer le polymorphisme, une troisième colonne nommée *rid_tablename* de type chaîne de caractère permet de référencer la table de l'individu.

4.1.2.3 Propriété collection de type simple

Une propriété collection de type simple (ensemble, tableau, liste) est mise en correspondance comme ci-dessus (cf.section 4.1.2.2). La clé de la table est composée de la colonne *rid* et de la colonne associée à la propriété. De plus dans cas d'une collection de type liste ou tableau, un champ (*rang*) est rajouté. Ce dernier fait également partie de la clé de la table et permet de conserver l'ordre des éléments de la collection.

4.1.2.4 Propriété de type objet

Pour chaque propriété de type objet, une table d'association est créée. Celle-ci porte le nom de la classe concaténé au nom de la propriété (*classe__propriete*). Les champs de cette table sont : une colonne *rid* (clé primaire de la table) représentant l'identifiant de l'instance source, une

colonne représentant l'identifiant de l'instance cible. Deux colonnes de types chaîne de caractères servent à spécifier la table des instances source et cible.

4.1.2.5 Propriété collection de type objet

Une propriété collection de type simple (ensemble, tableau, liste) est mise en correspondance comme ci-dessus (cf.section 4.1.2.4). La clé primaire de la table est constituée de la colonne *rid* et de la colonne représentant l'identifiant de l'instance cible car système assume l'unicité des identifiants. De plus, dans le cas d'une collection de type liste ou tableau, un champ (*rang*) est rajouté. Ce dernier fait partie de la clé de la table et permet de conserver l'ordre des éléments de la collection.

La figure 4.23 illustre la représentation des données suivant l'approche binaire.

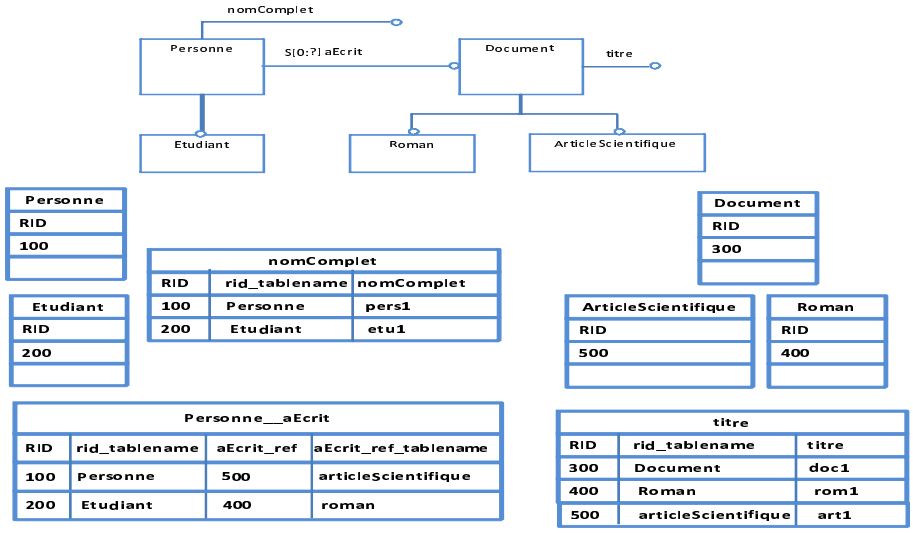


FIG. 4.23 – Représentation suivant l'approche binaire dans la partie donnée

Propriétés dépendantes

Une propriété dépendante est représentée comme une propriété simple suivant l'approche binaire. A la table associée est rajoutée une colonne pour chacun des paramètres de contexte dont elle dépend. Les colonnes des paramètres de contexte font partie de la clé de la table.

Propriétés de type Level type

Une propriété de type *Level_ type* est représentée comme une propriété simple suivant l'approche binaire. A la table associée est rajoutée une colonne pour chacun des qualificatifs utilisés pour décrire les instances. Le nom de chaque colonne est défini en suffixant la propriété par : *_ min*, *_ max*, *_ nom* ou *_ typ* suivant l'élément de type *Level* utilisé dans la description des instances.

4.1.3 Prise en compte des caractéristiques de propriété

La prise en compte des caractéristiques de propriété est nécessaire pour automatiser le comportement de certaines propriétés, en particulier, la création de trigger comme défini à la sec-

tion 4.4 du Chapitre 3. Nous présentons dans cette section la représentation des caractéristiques de propriétés simultanément pour la représentation horizontale et verticale.

4.1.3.1 Symétrie / Transitivité

Pour toute propriété symétrique ou transitive, nous créons un trigger attaché à la table où est représentée la propriété et, qui permet de saturer automatiquement cette dernière à chaque insertion. Avant toute insertion, ce trigger vérifie que les valeurs à insérer n'existent pas déjà dans la table.

4.1.3.2 Ordre

Toute propriété d'ordre est une propriété de type objet. En plus, nous lui ajoutons les colonnes nécessaires pour l'indexer avec la technique d'étiquetage active dans le système pour cette propriété. Une contrainte d'unicité est définie sur les colonnes d'indexation. Un trigger (associé à la table de son domaine) est également défini afin de fournir automatiquement les valeurs des colonnes d'index à chaque insertion. La figure 4.24 illustre, suivant l'approche de représentation horizontale, la représentation d'une propriété d'ordre associée à une technique d'étiquetage par intervalle. Sur cette figure, la propriété *seSubdiviseEn* est définit une relation d'ordre, les colonnes d'index sont *seSubdiviseEn_ bound1* et *seSubdiviseEn_ bound2*.

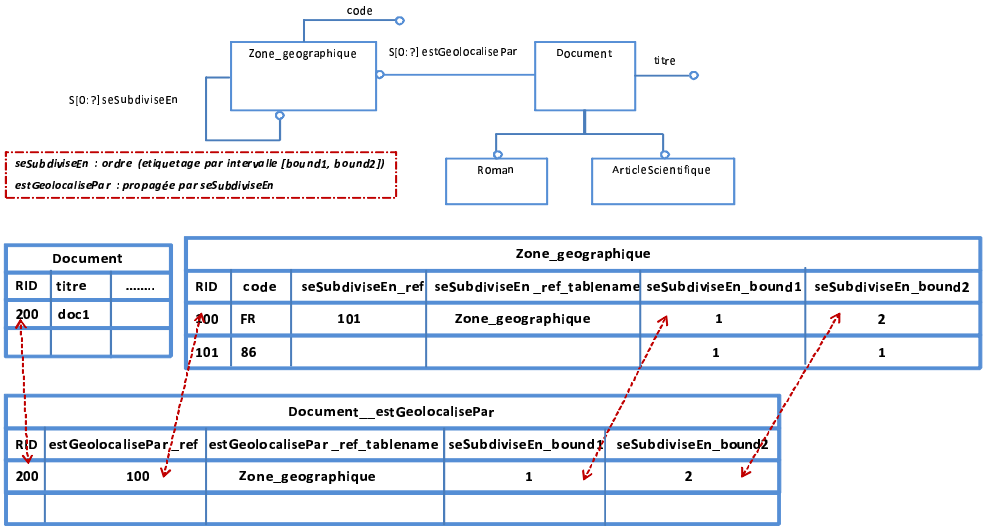


FIG. 4.24 – Représentation de propriété d'ordre/propagée dans la partie donnée

4.1.3.3 Propagation

Pour toute propriété propagée, nous ajoutons à la table correspondante, les colonnes nécessaires pour l'indexer avec la technique d'étiquetage active associée à la propriété d'ordre qui la propage. Un trigger (associé à la table de la propriété propagée) est également défini afin de fournir automatiquement les valeurs des colonnes d'index à chaque insertion. La figure 4.24 illustre, suivant l'approche de représentation horizontale, la représentation de la propriété *estGeolocalisePar* propagée par l'ordre *seSubdiviseEn*.

4.1.4 Choix des index

Afin d'optimiser le schéma des données défini à partir des règles ci-dessus, nous avons choisi de définir des index sur les colonnes correspondantes à certaines propriétés. Ainsi, nous définissons automatiquement :

- un index sur chaque colonne référençant une table (clé étrangère) ;
- un index sur chaque colonne d'une table d'association ;
- un index sur chaque colonne d'étiquetage définie pour une propriété d'ordre ou pour une propriété propagée.

Nous ne définissons pas d'index sur la clé primaire pour la table associée à chaque classe ; ceci car le SGBD PostgreSQL défini automatiquement un index sur toute colonne définie comme clé primaire. Pour les tables d'association, PostgreSQL défini des index multi-colonnes. Cependant, nous avons observé que ce type d'index n'est pas souvent utilisé lors du traitement des requêtes. Nous avons donc défini dans le cas des tables d'association un index mono-colonnes sur chaque colonne membre de la clé. Enfin, les indexs sur les colonnes d'étiquetage sont utilisés lors du traitement de requêtes impliquant une propriété d'ordre ou une propriété propagée comme nous le verrons au Chapitre 6.

4.2 Structure d'accès aux données

L'accès aux données est réalisé au travers de vues sur les différentes tables de la partie ontologie. Ceci afin de fournir une API de gestion et de manipulation des DBO harmonieuse des DBO canoniques et non canoniques.

4.2.1 Vues sur les classes canoniques

Dans le cas d'une représentation horizontale, une vue SQL est associée à une classe canonique possédant une extension. Une vue est également associée à chaque table d'association correspondant à une propriété multivaluée.

Une des difficultés majeures avec les vues est de permettre la modification des tables sous-jacentes. Certains SGBD comme Oracle et PostgreSQL, disposent de moyens permettant de modifier la base de données via certaines vues. Oracle, par exemple, décrit précisément les restrictions sur les vues que le système est capable de mettre à jour automatiquement. Ces vues doivent être de structure simple, de sorte qu'Oracle soit capable de déduire de façon unique les modifications à faire sur les tables. Elles ne doivent en particulier pas comporter : des opérateurs ensemblistes, des fonctions d'agrégation, de clause *group by* ou *order by*, des sous-requêtes, et des collections dans la clause *select*. Il est toutefois possible, pour les vues complexes, de définir des triggers *instead of* qui implémentent le comportement que l'on désire obtenir. C'est cette seconde approche qui est adoptée par PostgreSQL pour permettre la modification des vues.

Dans PostgreSQL, c'est à l'utilisateur de définir ses propres règles d'insertion, de mise à jour et de suppression dans les vues. Nous avons donc défini des règles pour réaliser ces différentes opérations. Par exemple, dans l'approche horizontale, nous ne manipulons que des vues simples, qui comportent chacune une clause *select* portant sur les colonnes de l'unique table référencée dans la clause *from*. Les règles que nous avons définies dans ce cas sont donc des règles simples.

4.2.2 Vues sur les classes non canoniques

Les vues associées aux classes non canoniques sont définies à partir de leur expression comme spécifié à la section 4.3 du chapitre 3.

Notons que, les vues ontologiques définies à partir de l'expression formelle des classes, ne peuvent pas être, en l'état, exécutées sur les données. En effet, lors de la définition de la vue SQL correspondante, il est nécessaire, de s'assurer que les classes impliquées dans les vues ontologiques possèdent bien une table d'extension dans la partie donnée et, que les propriétés impliquées dans la définition des vues (ou leurs inverses) sont bien utilisées pour caractériser dans la partie données les classes auxquelles elles s'appliquent.

4.3 Lien entre ontologie et données

Les ontologies et les données étant gérées dans des tables séparées, il est nécessaire d'établir un lien entre les concepts (classes et propriétés) de la partie ontologie et les tables de la partie données et inversement. Ce lien va permettre (1) d'interroger les DBO à partir de l'ontologie, (2) de présenter les données en termes de l'ontologie et (3) d'assurer la cohérence des données par rapport aux contraintes définies au niveau de l'ontologie.

Pour établir le lien entre les ontologies et les données, nous pouvons utiliser les noms ou les identifiants des classes et propriétés. Cependant, ces identifiants doivent respecter la longueur et le format (absence de caractère spéciaux) exigés par les SGBD. Cette solution n'est donc pas appropriée car ces contraintes ne peuvent pas toujours être vérifiées.

Les concepts classes et propriétés sont associées à un identifiant interne (*rid*) dans OntoDB2. Nous proposons de présenter la liaison entre les ontologies et les données en se basant sur ces identifiants internes.

- la table d'extension d'une classe est nommée en préfixant l'identifiant de cette classe par le caractère 'E' (pour extension).
- la vue permettant de calculer l'extension d'une classe est nommée en préfixant l'identifiant de cette classe par le caractère 'V' (pour vue).
- la nom de colonne correspondant à une propriété dans la table (ou la vue) de sa classe est obtenu en préfixant l'identifiant de cette propriété par le caractère 'P' (pour propriété).

Inversement, le concept ontologique définissant une table, une vue ou une propriété dans la partie donnée est déterminée à partir de l'identifiant interne de ce concept qui est obtenu en ôtant le préfixe 'E', 'V' ou 'P' du nom de la table, la vue ou la propriété dans la partie données.

4.4 API d'accès aux données

Pour la gestion des DBO, nous avons codé une API JAVA qui implémente l'ensemble des règles définies ci-dessus pour la définition des tables de la partie données. Cette API permet :

- la gestion de l'extension des classes ;
- la création, la modification et la suppression des DBO ;
- la création des vues sur les tables de la partie données.

La mise en œuvre de cette API nécessite d'accéder aux informations de niveau ontologique. Ces accès sont réalisés au travers de l'API JAVA *OntoLib* de la partie ontologie décrite à la section 2.5. La table 4.7 montre les signatures de quelques fonctions définies dans cette API. Par exemple, la

fonction *Create_Extension* crée dans la partie données d'OntoDB2, la ou les tables d'extension de la classe passée en paramètre, les propriétés utilisées pour les décrire sont également passées en paramètre.

.....API EXTENSION.....

// création de la table d'extension : classe, propriétés
Create_Extension : CLASS × [PROPERTY]$^+$;
 	
// création des colonnes suivant les types et les caractériqtiques : classe, propriété
CreateTable_For_Propagated_Property : CLASS × PROPERTY ;
Add_Labelling_Columns_to_Property : CLASS × PROPERTY ;
Create_Table_For_Dependent_Property : CLASS × PROPERTY ;
Create_Table_Column_For_Geometry_Type : CLASS × PROPERTY ;
Create_Table_Column_For_Level_Type : CLASS × PROPERTY ;
Create_Table_Column_For_Class_Instance_Type : CLASS × PROPERTY ;
 	
//creation des trigger : table , clé primaire, clé étrangère
ontoDB2_ext__create_transitive_trigger : STRING × STRING × STRING
 	
//création des vues : classe, propriétés
Create_View_For_Extension : CLASS × [PROPERTY]$^+$;
 	
// création des règles de gestion des vues : table, vue, propriétés
ontodb2_ext__create_delete_rule STRING × STRING
ontodb2_ext__create_insert_rule : STRING × STRING × [PROPERTY]$^+$
ontodb2_ext__create_update_rule : STRING × STRING × [PROPERTY]$^+$
// .....SUITE.....

TAB. 4.7 – API d'accès aux données de la partie ontologie d'OntoDB2

4.5 Importation des DBO

Nous avons réalisé un programme permettant d'importer les instances des ontologies OWL Lite dans la partie données de OntoDB2. Ce programme fait appel aux POJO de l'API *OntoLib* de la partie ontologie et à l'API *Extension*. Il permet d'importer les instances explicitement définies comme appartenant soit à une classe primitive, soit à une autre classe définie respectant les hypothèses d'OntoDB2 (cf. section 4 du chapitre précédent). L'algorithme d'importation est réalisé suivant le principe suivant :

1. Sélectionner les classes primitives (*owl :Thing* inclus) possédant des individus.
2. Calculer pour chaque classe, l'ensemble des propriétés utilisées pour décrire les instances de cette classe.
3. Créer la table d'extension de chaque classe avec les propriétés utilisées.
4. Créer un fichier CSV pour chaque classe (primitive) avec comme entête une colonne identifiant (rid) et les identifiants (préfixés par 'P') des propriétés de cardinalités 0 :1 ou 1 :1.
5. Créer un fichier CSV pour chaque propriété de type collection

6. Initialiser les lignes des fichiers CSV avec les valeurs des propriétés des instances.
   - Pour une propriété simple, l'initialiser à la valeur de la propriété
   - Pour une propriété de type objet, l'initialiser à l'identifiant de l'individu associé si ce dernier a déjà été traité, sinon marque la référence comme étant non résolue.
7. Sélectionner les classes définies possédant des individus.
8. Pour chaque classe définie,
   - si la classe primitive à partir de laquelle elle est définie possède un extension, mettre à jour l'ensemble des propriétés utilisées par cette classe
   - sinon, créer la table d'extension de la classe à partir de laquelle elle est définie avec les propriétés utilisées la classe définie.
   - Créer le ou les fichiers CSV nécessaire pour sa classe de base primitive (voir étapes 4 et 5).
9. Convertir les instances non canonique et les insérer dans les fichiers CSV de la classe primitive (voir étape 6).
10. Résoudre les références non résolues au premier passage.
11. créer la vue SQL associée à chaque classe primitive et à chaque classe définie.

4.6 Synthèse sur la partie donnée

Nous avons présenté ci-dessus, les différents modules développés pour la mise en œuvre de la partie donnée. Le développement de ces modules a été entièrement réalisé en JAVA. La gestion de la partie données est réalisée via une API JAVA qui implémente directement sur la base de données cible, l'ensemble des règles de mise en correspondance entre les concepts des ontologies et la base de données. Les données sont manipulées au travers de vues.

A la différence de la partie ontologie qui repose sur le framewok Hibernate pour assurer sa portabilité, l'API de manipulation de la partie données est dépendante de la base de données cible. Hibernate n'a pas été utilisé pour le développement de l'API de la partie données du fait que (1) des fonctions spécifiques doivent être réalisées pour définir le comportement de certaines propriétés (triggers pour la transitivité, ordre, propagation, ...), (2) la structure de la partie données est assez dynamique (les tables sont définies au fur et à mesure qu'une extension est associée à une classe ou qu'une nouvelle propriété est choisie pour décrire les instances d'une classe). Il aurait donc fallu arrêter l'application pour compléter le fichier de mapping et re-générer les POJO à chaque modification.

5 L'application graphique de gestion : OntoWEB

Afin d'illustrer le niveau de finalisation du prototype d'OntoDB2, nous présentons le client d'accès OntoWEB que nous avons développé sur les API décrites dans ce chapitre pour la gestion des ontologies et des données.

Notre interface est développée à l'aide de la bibliothèque J2EE[32] et utilise le modèle MVC [33]. MVC est un modèle d'architecture qui propose de séparer une application en trois parties :

[32] JAVA.sun.com/JAVAee/
[33] JAVA.sun.com/blueprints/patterns/MVC.html

1. le modèle, qui représente les données et les règles métier.
2. la vue, qui correspond à l'interface avec laquelle l'utilisateur interagit.
3. le contrôleur, qui interprète les requêtes de l'utilisateur et appelle le modèle et la vue nécessaires pour répondre à la requête.

Le modèle MVC est une des composantes essentielles d'une application Web ; il assure une maintenance plus aisée des applications. En effet, contrôleur et modèle étant indépendant de la couche graphique, une remise en cause de cette couche de l'application n'aura que peu d'impact sur l'application, puisque seuls les composants de la vue seront à modifier. De la même manière, une modification de la structure du système d'information (le modèle) n'aura pas d'impact sur la vue.

Notre interface est destinée à être accessible via des clients légers. Les utilisateurs n'ont besoin, d'aucun environnement particulier pour accéder à l'application. Ils ont juste besoin d'un navigateur, quel qu'il soit. En effet, les documents au format HTML assurent l'indépendance entre le système d'exploitation et les informations échangées.

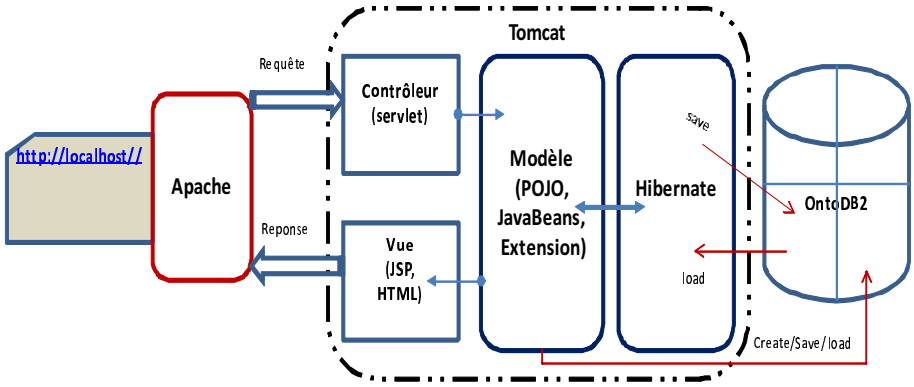


FIG. 4.25 – Architecture logicielle de l'application OntoWEB

La figure 4.25 présente l'architecture générale d'OntoWEB. Du point de vue statique, les composants sont les suivants.

La vue, (coté client) représente la partie visible. Elle est composée de pages HTML, de pages JSP (JAVA Server Page) dynamiques présentées après exécution comme des pages HTML et, de librairies personnalisées, ou taglibs, que nous avons écrits pour étendre les balises de base de JSP. Ces taglibs nous permettent d'alléger les pages JSP en factorisant les actions qui se répètent dans différentes pages. Ils permettent ainsi de réduire l'utilisation et le volume de code JAVA dans les pages JSP par la présentation sous forme de simples balises XML des actions complexes qui auraient nécessité l'écriture de nombreuses lignes de code dans les pages JSP. Par exemples : l'affichage du contenu d'une classe ou encore la génération de la hiérarchie des classes de l'ontologie.

Le contrôleur (coté serveur) est assuré par une servlet (Super-Contrôleur) qui est chargé d'initialiser l'application, toutes les actions que l'application permet de réaliser et, d'aiguiller les requêtes vers les différentes classes d'actions. Le formalisme adopté pour nommer les classes d'action est : *[type_ action]Nom_ ClasseAction*. La valeur du terme entre crochets est optionnelle. Par exemple : *SavePropertyAction* comme son nom l'indique prend en charge l'opération

d'enregistrement d'une propriété.

Le modèle (coté serveur) est composé de différents packages :

- Le package *utils* responsable des sorties.
- Le package *beans* qui implémente des classes responsables de la capture des entrées des formulaires utilisateurs, et encapsule les actions sur la base de données comme sauvegarder des informations, en faisant appel aux classes du package *OntoLib*.
- Le package *OntoLib* qui contient les POJO manipulés par Hibernate
- Le package *extension* qui implémente des classes de gestion du contenu des classes de l'ontologie (niveau données de l'ontologie).

La couche mapping ou de persistance assurée par Hibernate.

Nous résumons ci-dessous, les capacités d'OntoWEB en présentant quelques unes de ses fenêtres utilisateurs.

5.1 Fenêtre principale

La figure 4.26 est une capture d'écran de la fenêtre principale de notre interface d'accès. Son ouverture fait suite à une opération d'authentification de l'utilisateur. Son organisation est comme suit :

- Sur la partie gauche (1), une arborescence représente la hiérarchie des classes de l'ontologie.
- Sur la partie supérieure (2), un menu permet d'avoir accès à des fonctions générales de manipulation de l'ontologie.
- Sur la partie supérieure droite (3) , un drapeau (facultatif) indique la langue dans laquelle est représentée l'ontologie

Sur la partie droite on distingue deux zones :

1. Une zone supérieure (4), donnant accès aux attributs définissant la classe de l'ontologie recherchée dans la hiérarchie,
2. Une zone inférieure (5) définissant les attributs des propriétés caractéristiques sélectionnées dans la zone décrivant la classe sélectionnée.

Nous décrivons maintenant, de façon précise, le rôle de chacune de ces différentes zones.

5.2 Gestion des ontologies

La hiérarchie des classes (partie 1) permet de parcourir les différentes classes d'une ontologie. L'accès à une classe se fait soit en déroulant la hiérarchie jusqu'à atteindre le nœud matérialisé par le nom préféré de la classe, soit en effectuant une recherche basée sur toute ou partie du nom préféré de la classe à partie de la zone de recherche situé sur le coin supérieur gauche de la partie 1.

Dans la hiérarchie, le nom de la classe est associé à un lien hypertexte permettant d'accéder à la partie descriptive de la classe (partie droite de la fenêtre).

5.2.1 Description de classe

La partie descriptive des classes (partie 4) définit les informations suivantes (voir figure 4.27) :

- le nom de la classe (classe),

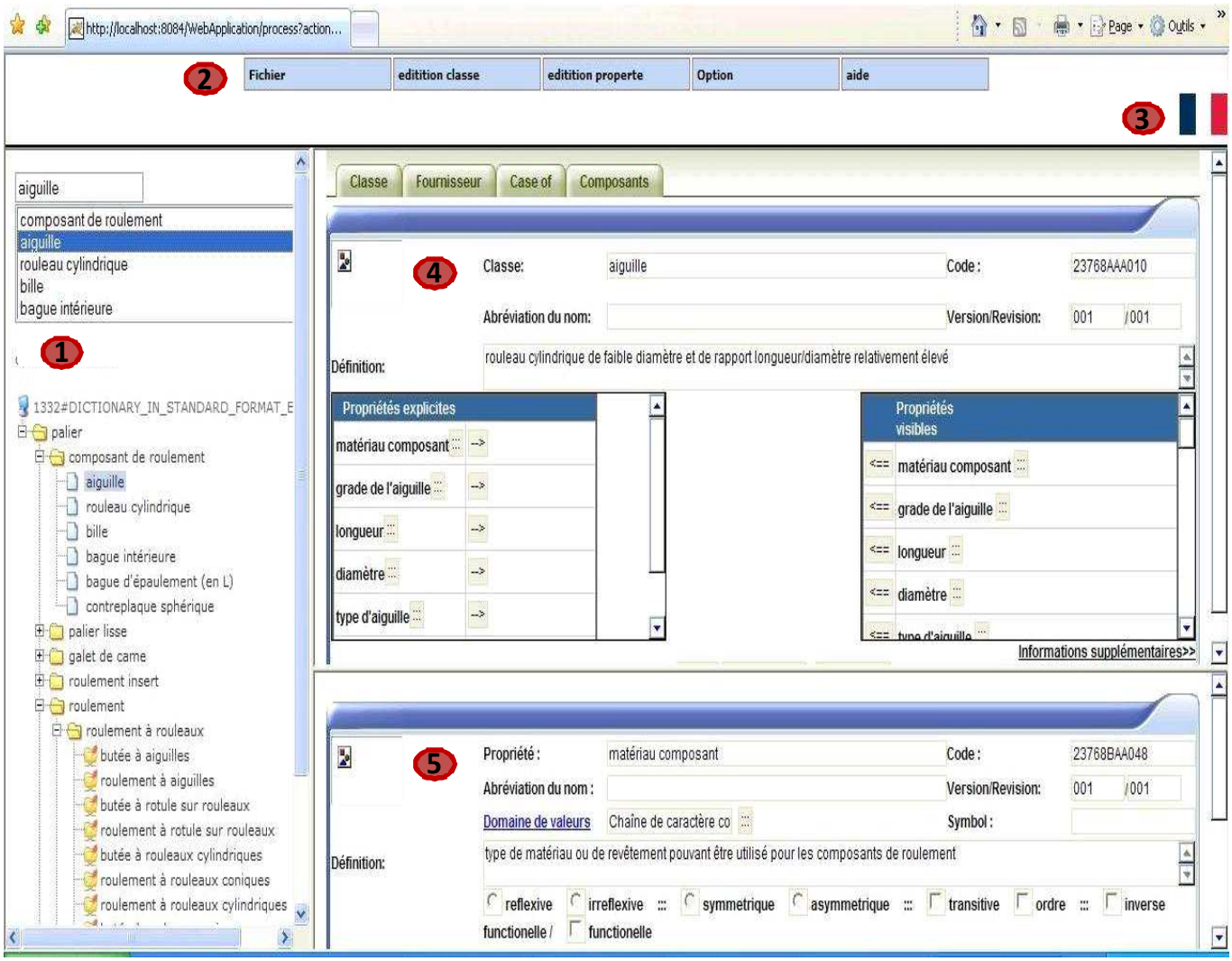


FIG. 4.26 – Fenêtre principale de l'interface OntoWEB

- la description de la classe (Définition),
- les propriétés utilisables pour cette classe (listées dans deux objets graphiques de type tableau) :
  - Les propriétés visibles : c'est l'ensemble constitué de toutes les propriétés (qui peuvent être héritées) ayant un sens au niveau de la classe courante.
  - les propriétés explicites ou applicables : celles des propriétés qui constituent les conditions d'appartenance à la classe ou à ses sous classes. Le fait qu'une propriété soit applicable ne signifie par qu'elle sera valuée dans la représentation d'une instance particulière de la classe. Cela signifie que la caractéristique comportementale doit exister conceptuellement pour tout objet de la classe.
- un lien (« informations supplémentaires ») permet d'accéder à des informations complémentaires (cf. figure 4.28) associées à la description de cette même classe (exemple : note, remarque, document source,..).

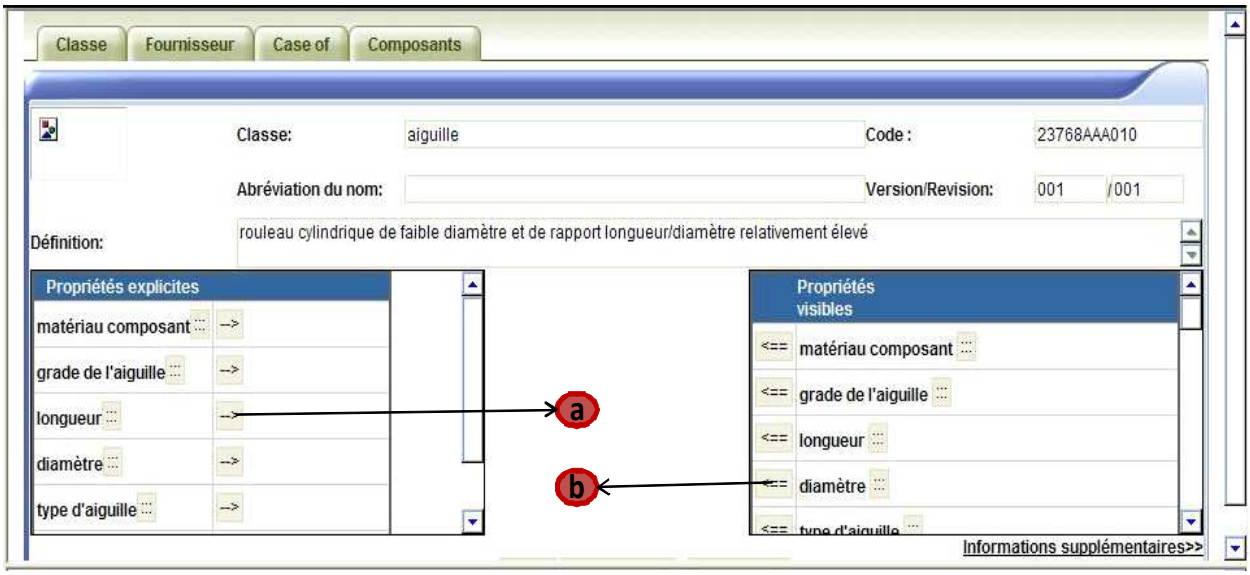


FIG. 4.27 – Visualisation d'une classe avec l'interface OntoWEB

On pourra noter la présence d'information d'identification (*Code*), de gestion de version (*Version/Revision*) ou encore un nom court (« Abbréviation du nom »). Dans l'exemple de la figure 4.27, nous présentons la visualisation d'une ontologie concernant des éléments de roulement mécaniques (roulement à billes, roulement à aiguille, ...). Par ailleurs, la classe « aiguille » a été sélectionnée. Ceci permet de visualiser les caractéristiques de cette classe, ainsi que les propriétés qui lui sont rattachées.

La première propriété explicite, « matériau composant »(cf. figure 4.26), est affichée dans la partie propriétés applicables (suite à la sélection de la classe), et ses caractéristiques sont celles présentées dans la partie 5 (voir figure 4.29). Pour visualiser une propriété, il suffit de cliquer sur le nom de la propriété. Les boutons a et b permettent de mettre à jour la liste des propriétés

explicites.

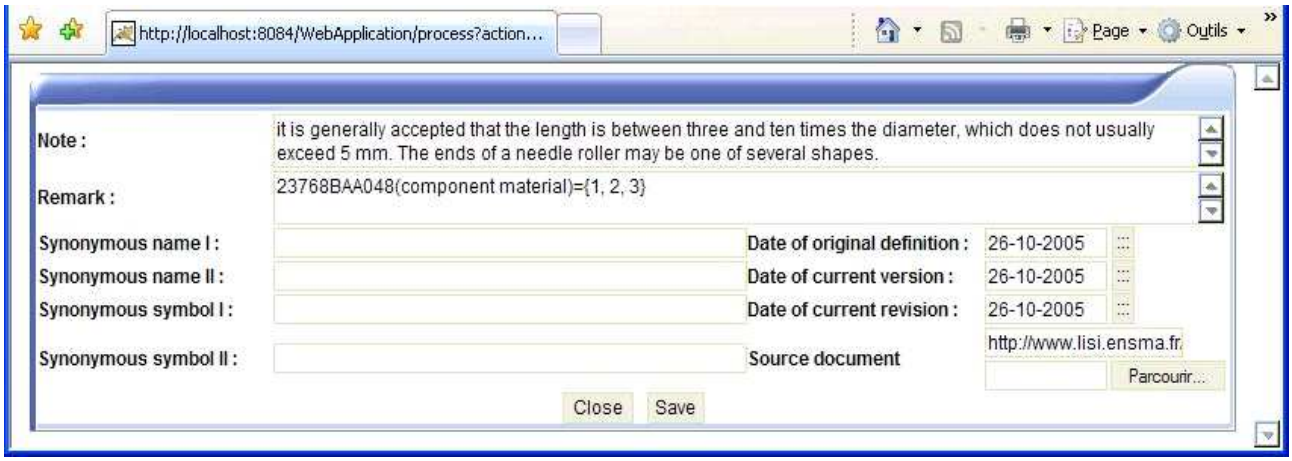


FIG. 4.28 – Visualisation des informations complémentaires d'une classe avec l'interface OntoWEB

5.2.2 Description de propriétés

Les propriétés représentent la connaissance descriptive associée aux classes du modèle d'ontologie. Elles permettent de décrire les instances de ces classes en leur associant des valeurs. Les propriétés caractérisant la classe courante sont listées dans la partie descriptive de celle-ci. Le nom d'une propriété est associé à un lien hypertexte permettant d'accéder à sa propre partie descriptive.

La partie descriptive d'une propriété décrit (cf. figure 4.29) :

- son nom (Propriété),
- sa définition (Définition),
- son type (Domaines de valeurs) et,
- dans le cas des propriétés dont la valeur dépend d'un contexte de mesure, les propriétés qui caractérisent le contexte,
- les caractéristiques de la propriété (réflexive, transitive, ordre, ...) ;
- un lien situé au bas du cadre (« informations supplémentaires ») permet d'accéder à des informations complémentaires associées à la description de cette même propriété (exemple : note, remarque, document source,..).

Tout comme pour les classes, là encore, des informations d'identification (Code), de gestion de version (Version/Revision), et de nommage (Abbréviation du nom, Symbole) complètent la description des propriétés.

5.2.3 Les types

Le domaine de valeur d'une propriété pouvant être très complexe, nous proposons sur la zone de description d'une propriété (cf. figure 4.29), d'accéder à la partie descriptive de son domaine de valeur dans une fenêtre indépendante à partir du lien *Domaine de valeurs*.

FIG. 4.29 – Visualisation d'une propriété avec l'interface OntoWEB

Les différents types de données manipulés : booléen, naturel, chaîne de caractères, entier, réel, mesure entière, mesure réelle, code énuméré, entier énuméré, monétaire entier, monétaire réel, borné, agrégat (liste, tableau, ensemble, sac), géométrie, polygone, ligne, classe, entité (fichiers externes) et les types définis par l'utilisateur, sont visualisés au travers d'une interface particulière développée pour chacun d'entre eux.

La figure 4.30 présente une fenêtre de description d'une propriété de type *entier*. La seule information complémentaire à la définition de ce type est le format de la valeur (*Format*) destiné à réaliser un affichage des valeurs selon un patron particulier sur un système s'appuyant sur l'ontologie décrite.

FIG. 4.30 – Visualisation d'un type entier avec l'interface OntoWEB

La figure 4.31 présente la fenêtre de description d'un type *code énuméré*, dans laquelle, pour chaque code, on précise des informations complémentaires telles l'icône, le nom, le nom court ou encore la source documentaire où est défini le code.

La figure 4.32 présente les informations associées à un type collection. Il s'agit ici d'une collection (de type ensemble) d'objets. Le nom préféré de la classe référencée est affichée sur zone intermédiaire de cette fenêtre et, le bouton « Atteindre »permet de parcourir la hiérarchie des classes située sur la zone inférieure pour atteindre cette classe.

FIG. 4.31 – Visualisation d'un type énumérée avec l'interface ONTOWEB

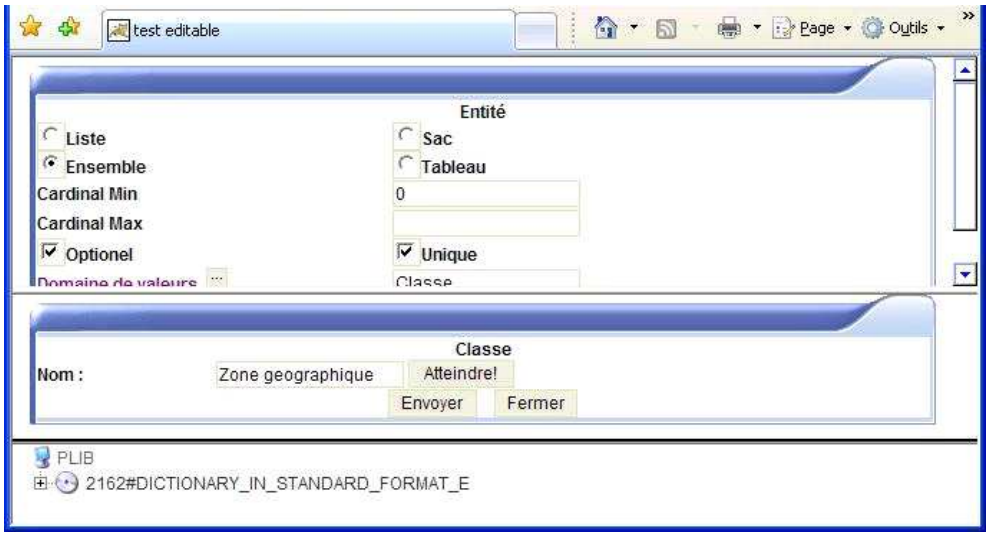


FIG. 4.32 – Visualisation d'un type collection d'objets avec l'interface ONTOWEB

5.2.4 Le Multilinguisme

Les ontologies que nous manipulons sont des ontologies multilingues. Les aspects textuels de la description de chacun des concepts de l'ontologie (exemple : classe, propriété) peuvent apparaître dans un nombre quelconque de langues. Nous proposons à travers le « menu option » deux choix de langues afin de permettre à l'internaute de modifier la langue de présentation des informations. La langue courante de présentation est indiquée par le biais d'un drapeau situé au coin supérieur droit de l'interface générique. La figure 4.33 présente la fenêtre utilisateur de la figure 4.26, mais cette fois ci en anglais. Le support du multilinguisme est en effet assuré tant au niveau des données ontologiques (ontologie multilingues) qu'au niveau de la présentation (interface utilisateur multilingues). Notons dans ans cet exemple, que la langue de l'ontologie est représentée par un « drapeau »dans la partie supérieure droite.

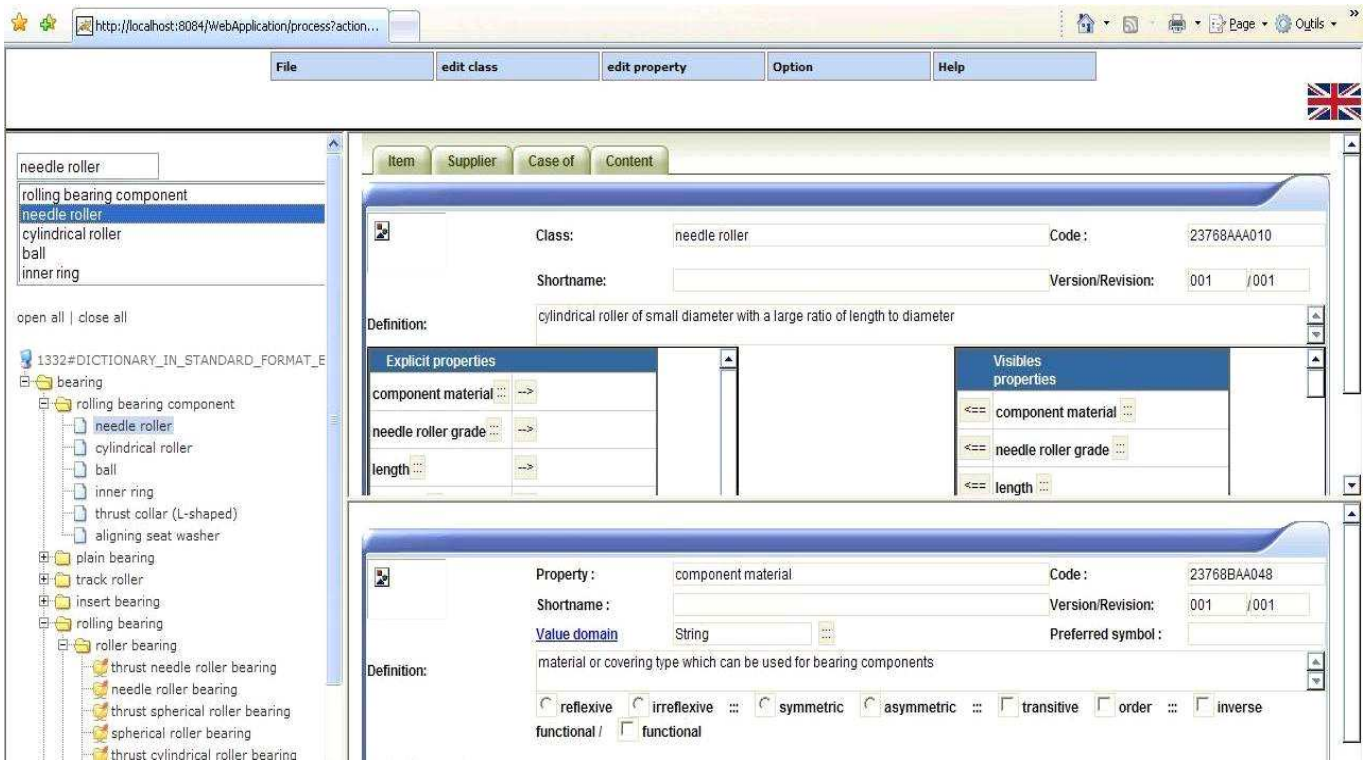


FIG. 4.33 – Visualisation de l'interface principale OntoWEB :multilinguisme

5.3 Gestion des DBO

A partir de la sélection d'une classe d'une ontologie, nous pouvons accéder à sa population (si elle existe) via l'onglet « composants ». La figure 4.34 présente la population de la classe *Vis hexagonale à rondelle élastique*.

Cette population est constituée d'un ensemble d'instances représenté sous une forme tabulaire. Chaque colonne de la table correspond à une propriété prise parmi les propriétés disponibles (applicables) de la classe sélectionnée de l'ontologie. Chaque colonne possède un entête décrivant la propriété. OntoWEB offre la possibilité de passer du niveau donnée au niveau connaissance (accès à la sémantique des données défini par l'ontologie) par des entêtes de colonnes « actives »qui sont des liens hypertextes permettant d'accéder à la partie descriptive de la propriété représentée par les symboles apparaissant dans les cellules d'une colonne de la table.

Notons que les instances sont affichées par pages et que des boutons de navigation situés à droite immédiatement après le tableau permettent de parcourir les pages. L'utilisateur peut trier les données (ordre croissant/décroissant) suivant chaque colonne et chaque instance est manipulable au moyen d'un ensemble de fonction accessibles sous la forme de boutons immédiatement après le tableau de valeurs. Les actions disponibles sont :

- Editer : permet d'afficher dans une fenêtre indépendante les valeurs des propriétés pour une instance donnée en lecture seule ;
- Dupliquer : permet d'ajouter une nouvelle instance à partir d'une instance existante dont

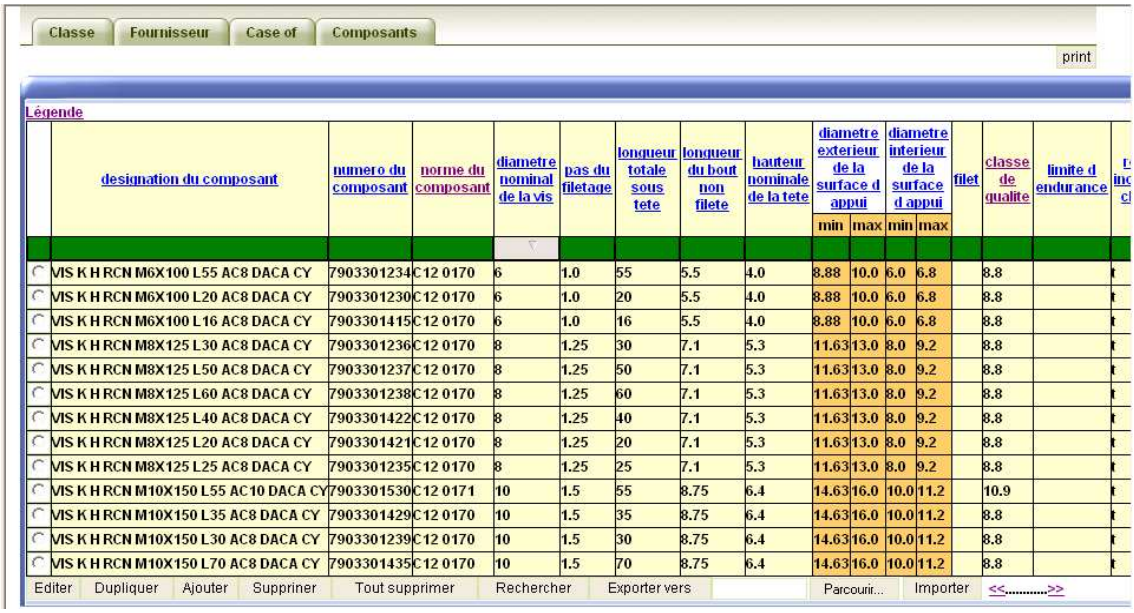


FIG. 4.34 – Visualisation de l'extension d'une classe avec l'interface OntoWEB

elle est proche (en terme de valeurs des propriétés) par modification de cette dernière ;
- Ajouter : permet la création d'une nouvelle instance par saisie des valeurs de ses propriétés ;
- Supprimer : permet de supprimer l'instance sélectionnée ;
- Tout supprimer : permet de supprimer toutes les instances de la classe courante.

Enfin, notons que des codes de couleurs ont été utilisés afin de discriminer les types de propriétés. Ces codes de couleurs sont explicités dans une fenêtre accessible via le lien « Légende ».

La figure 4.35 présente la fenêtre de visualisation d'uns instance particulière. Cette fenêtre les fonctions de mise à jour des valeurs des propriétés de l'instance ou encore de suppression de l'instance.

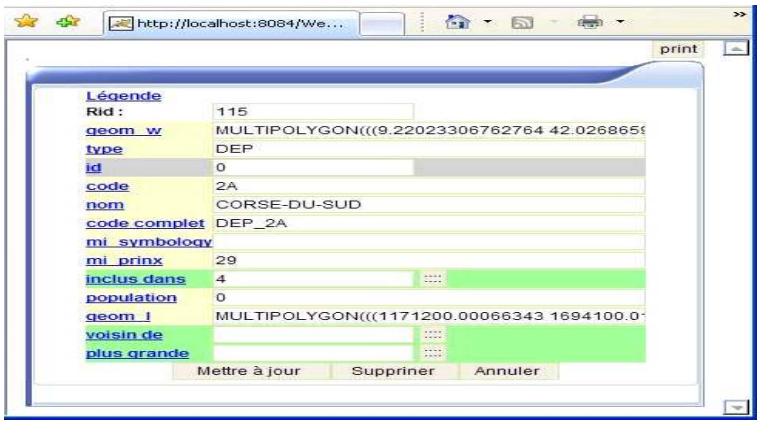


FIG. 4.35 – Visualisation d'une instance particulière d'une classe avec l'interface OntoWEB

6 Synthèse sur l'implémentation d'OntoDB2

La figure 4.36 synthétise l'ensemble des développements effectués pour développer le prototype OntoDB2 pour la gestions des ontologies et des DBO associées et ses différentes API. A la base de ces API, nous avons l'API *OntoLib* constituée des POJO manipulés par Hibernate pour la partie ontologie, l'API *opengisLib* d'extension des types de données spatiaux, et la partie méta-schéma et l'API *Extension* pour la manipulation de la partie données. Ces API sont utilisées à ce jour par l'interface OntoWEB et par la nouvelle version de l'interpréteur de requête ontologique OntoQL en cours de développement sur OntoDB2. La figure 4.36 récapitule l'ensemble des modules qui ont été développés. Dans cette figure, les modules non réalisés et/ou non encore achevées sont grisés.

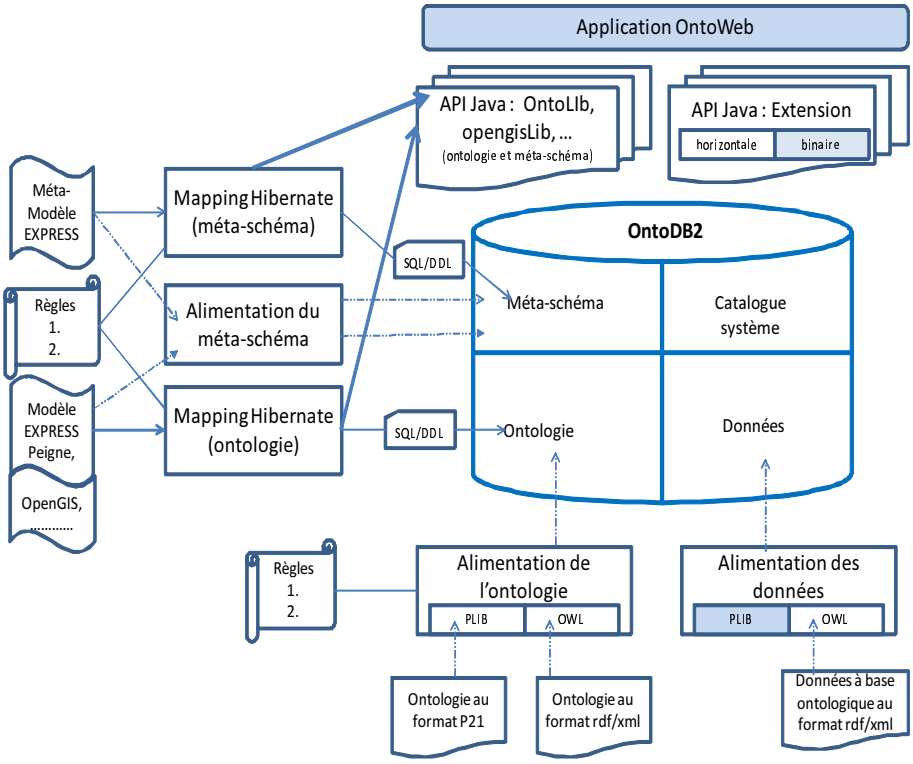


FIG. 4.36 – Récapitulatif des modules du prototype de BDBO OntoDB2

Conclusion

Nous avons présenté, dans ce chapitre, l'implémentation du prototype de BDBO OntoDB2. Ce prototype avait pour objectif de montrer la faisabilité de développer des systèmes de BDBO restant flexible et efficace au niveau de l'ontologie manipulée. Ce prototype met en œuvre des solutions définies lors de la phase de spécification réalisée au chapitre précédent.

Une des caractéristiques principale d'OntoDB2 est d'être fondée sur un formalisme d'ontologie flexible. Pour assurer cet objectif, la solution que nous avons choisi a été d'utiliser des règles de

transformation systématiques associées à des techniques d'IDM, où les développements sont réalisées au niveau méta-schéma pour concevoir des modules indépendants du formalisme d'ontologie qui s'adaptent aux évolutions de ce dernier. Nous avons ainsi pu mettre en œuvre cette solution pour la génération de la structure des tables de la partie méta-schéma et de la partie ontologie, la définition d'une API JAVA de gestion de ces deux parties et, l'alimentation de la partie méta-schéma par le méta-schéma lui-même et le formalisme d'ontologie et enfin, l'alimentation de la partie ontologies par des ontologies existantes.

La seconde caractéristique d'OntoDB2 est la gestion des DBO par des vues et, la représentation dans la partie donnée uniquement des DBO canoniques. Trois difficultés principales ont été rencontrées lors de mise en œuvre de la partie données : (1) la gestion des opérations classiques de manipulation des données sur les vues, (2) la transformation de DBO non canoniques en DBO canoniques, et, (3) la portabilité de la partie données. Concernant la première difficulté, le SGBD PostgreSQL ne prenant pas automatiquement en charges les opérations sur les vues, nous avons défini des règles de gestion ces opérations. Concernant la seconde difficulté, nous importons uniquement les DBO canoniques et les DBO non canoniques des classes OWL Lite, dont la description est complète et dont la classe de base respecte le cadre d'hypothèses que nous avons défini pour OntoDB2. Enfin concernant la troisième difficulté, des travaux en cours au laboratoire étudient l'utilisation d'Hibernate pour mettre à jour dynamiquement à la fois les fichiers de mapping et les POJO d'accès et ceci, sans avoir besoin d'arrêter l'application. Notons que cette solution permettrait également d'étendre dynamiquement le formalisme d'ontologie.

Nous avons également développé une interface graphique de type Web qui offre une ergonomie masquant toute la complexité non seulement du formalisme d'ontologie sous-jacent, mais aussi de son implémentation dans le système de gestion de bases de données cible. Cette interface permet à un utilisateur de gérer les ontologies et des DBO. Cette interface demande toutefois à être complétée. Il n'est par exemple pas possible à partir de cette interface de définir l'expression d'une classe non canonique. De même, cette interface pourrait être étendue pour permettre, par exemple, lorsqu'un un nouvel attribut est ajouté à une entité du formalisme d'ontologie la visualisation de ce dernier. Parmi les développements ultérieurs, il serait intéressant de générer automatiquement les formulaires de l'interface utilisateur à partir des POJOs utilisés par Hibernate sur la partie ontologie. De nombreux développements peuvent encore être effectués pour faciliter la personnalisation du formalisme d'ontologie. Ce qui convient néanmoins de noter est que l'approche d'IDM utilisée permet, pour beaucoup de développements, de les effectuer de façon générique ce que ne permettent, à notre connaissance, aucune des BDBO existantes.

Nous allons dans le chapitre suivant étudier la validation opérationnelle du prototype OntoDB2.

Troisième partie

Validation

Chapitre 5

Application : Raisonnements numériques sur les ensembles partiellement ordonnés

Sommaire

Introduction

Nous avons souligné au chapitre 2, l'existence de besoins spécifiques pour de nombreuses d'applications qui nécessitaient une flexibilité de modélisation à la fois du formalisme d'ontologie

et du système de types accessibles dans les BDBO. Nous avons déjà présenté comme application le cas des données géographiques (cf. section 2.2.3 du chapitre 3).

Nous présentons ici la capacité d'OntoDB2 à répondre à cette exigence en traitant de façon complète le cas de l'utilisation des données géographiques pour indexer des documents. L'approche que nous illustrons consiste à utiliser les capacités des bases de données à traiter efficacement les requêtes numériques et alphanumériques afin d'indexer des opérations de raisonnements qui sinon demanderaient un raisonnement déductif lors de l'exécution d'une requête. Ainsi, notre approche consiste à :

- exploiter la représentation ontologique disponible dans une BDBO pour connaître les caractéristiques des propriétés de type objet (transitive, symétrique, ordre, ...) ;
- enrichir les instances avec de nouvelles valeurs de propriétés jouant le rôle d'index pour remplacer le raisonnement déductif par le traitement de requêtes numériques (ou alphanumériques).

Par exemple, lorsqu'une propriété de type objet π est décrite en utilisant les caractéristiques d'OWL2 *asymmetric* et *transitive*, définissant ainsi un ordre ($\prec$) strict qui, en plus est arborescent (c'est à dire que le graphe de la relation est une forêt), alors cet ordre arborescent peut être représenté par un intervalle numérique [3].

Ainsi, (1) lorsqu'une instance présentant une valeur pour la propriété π est insérée dans la base de données, deux valeurs de propriétés additionnelles (*bound1*, *bound2*) sont automatiquement calculées par le système. Ces valeurs de propriétés reflètent l'ordre arborescent, c'est-à-dire que :

$$\mathrm{x} \prec \mathrm{y} \Leftrightarrow \mathit{bound1}(\mathrm{y}) < \mathit{bound1}(\mathrm{x}) < \mathit{bound2}(\mathrm{x}) < \mathit{bound2}(\mathrm{y}).$$

Ensuite, (2) lorsque des instances d'annotation plus petites qu'une instance donnée sont recherchées, l'interpréteur de requêtes accède à l'ontologie et réécrit la requête récursive utilisant π en une requête numérique utilisant *bound1* et *bound2*. Ce type d'index n'est pas nouveau. En effet, plusieurs approches [3, 14], connues sous le nom *d'étiquetage* (*labeling*), ont proposé de calculer la fermeture transitive de relations en les indexant par des labels numériques ou alphanumériques. Cependant, ces approches sont souvent codées en dur dans le système de gestion de données pour des relations prédéfinies telles que la subsumption de classes.

Nous proposons dans ce chapitre, des extensions aux formalismes d'ontologies qui permettent (1) d'identifier les situations où ce type d'approche peut être suivi et (2) de l'implémenter dynamiquement lorsqu'une ontologie est chargée. La formalisation proposée intègre diverses techniques d'étiquetage permettant de raisonner sur les différents types de relations récursives d'inclusion. Ceci recouvre le raisonnement sur les requêtes taxonomiques, très utilisé dans l'annotation de ressources. De plus, nous montrons que cette formalisation peut également être utilisée efficacement pour les structures de DAG (graphe orienté acyclique) employées dans les applications spatiales et temporelles.

L'organisation de ce chapitre est la suivante. Dans la section 1, nous décrivons les objectifs à atteindre au travers d'un exemple qui illustre le besoin de représentation des objets spatiaux et, présentons les requêtes types auxquelles doivent répondre le système. Nous proposons dans la section 2, une formalisation du problème à résoudre. La section 3 présente en détail notre proposition pour étendre les systèmes de BDBO afin de les doter d'une solution efficace de représentation et de traitement des objets spatiaux. Notre proposition utilise les techniques de

labelling topologiques et géométriques pour permettre la gestion des requêtes portant sur des données spatiales sans avoir à faire appels aux fonctions spécifiques disponibles par les extensions spatiales des SGBD. La section 4 présente la mise en œuvre de notre solution réalisée sur OntoDB2. Nous terminons par une conclusion.

1 Raisonnements Numériques sur des Ensembles Partiellement Ordonnés

1.1 Exemple Motivant

Le but du projet e-Wok Hub[34] est de gérer la mémoire de plusieurs projets d'ingénierie sur la capture et le stockage de CO_2. En particulier, un objectif important est d'améliorer la qualité des recherches de documents sur ce sujet. L'approche suivie consiste à utiliser des annotations de documents définies, autant que possible, de manière automatique. Le système e-Wok Hub vise à permettre de répondre à des questions pratiques sur les sites de stockage du CO_2, en retournant par exemple, une liste de documents pertinents (annotés par certains concepts et fournissant des informations sur tout ou partie d'une zone géographique donnée) pour faciliter la tâche de sélection d'un site.

Dans le cadre du projet e-Wok Hub, nous nous sommes en particulier intéressés à l'aspect géographique des annotations.

- Une ontologie existante appelée COG[35], qui décrit les zones géographiques françaises est utilisée pour annoter les documents. Dans cette ontologie, les zones décrites correspondent au découpage administratif de la France. Ces zones géographiques administratives ont la structure d'une forêt.
- Un autre ontologie, COG+, étend le COG en intégrant en plus des zones géographiques non administratives. Les instances de zones géographiques de cette ontologie décrivent cette fois-ci non plus une forêt, mais un DAG.

Le service d'annotation sémantique du projet utilise l'ontologie COG [36] décrite en OWL, pour identifier, dans un document fourni en entrée, les termes référençant des concepts de l'ontologie COG. Ces termes sont utilisés pour produire en sortie, des annotations au format RDF/XML. La figure 5.1 [37], illustre le processus d'annotation automatique des documents. A partir de l'ontologie COG, et de l'extrait de document fournit en entrée, le terme *Ile-de-France* est identifié dans un segment du document. L'annotation produite en sortie du service d'annotation décrit le fait que la segment d'URI « wl ://myDocument#1inriaGeo_1 » du document en entrée est géolocalisé par la zone géographique d'URI « geo :REG_11 » qui correspond à la région *Ile-de-France* dans l'ontologie COG. Les annotations ainsi produites vont être stockées dans la BDBO OntoDB2. Elles pourront donc être, par la suite, exploitées pour répondre aux différentes interrogations des utilisateurs. Une centaine d'interrogations types ont été identifiées pour l'ensemble du projet e-Wok Hub. Parmi elles, celles relatives aux aspects géographiques doivent permettre, par exemple, de :

- retrouver les informations de localisation géographiques globales de chaque document ;

[34] http ://www-sop.inria.fr/acacia/project/ewok/
[35] Code Officiel Géographique, *http ://rdf.insee.fr/geo/*
[36] http ://rdf.insee.fr/geo/
[37] source : http ://www-sop.inria.fr/acacia/project/ewok/

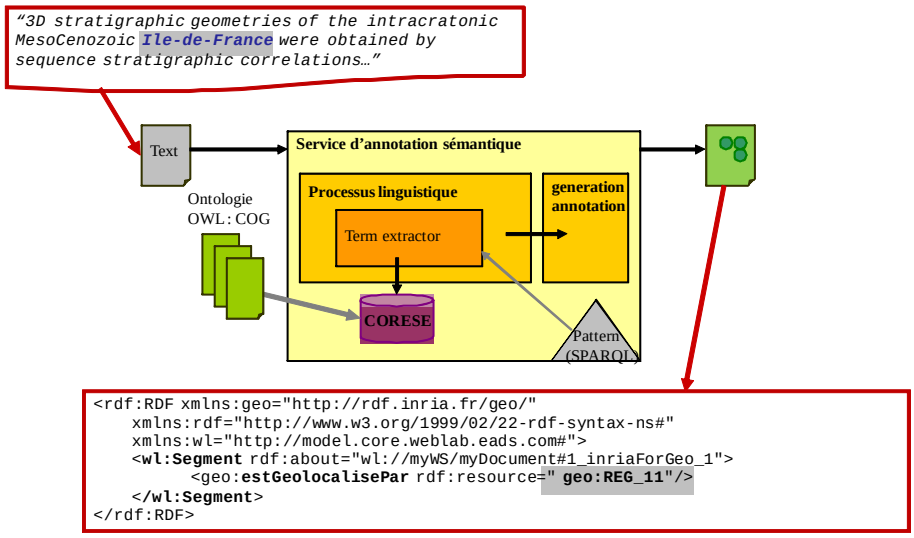


FIG. 5.1 – Illustration du processus d'annotation e-Wok Hub.

– rechercher les documents qui parlent de la zone géographique *Ile-de-France*.

En réponse à ces différentes interrogations, le système doit fournir une liste de documents pertinents qui vont guider la sélection d'un site de stockage.

Le système e-Wok Hub propose deux alternatives d'interrogation comme illustré dans la figure 5.2 :

1. une interrogation indirecte à partir du nom de la zone géographique entrée dans une zone de texte. Ce type d'interrogation est en particulier utilisé lorsque l'interrogation porte sur une zone administrative de l'ontologie COG ;
2. une interrogation directe, à partir du choix sur une carte d'un rectangle délimitant la zone de recherche. Cet type d'interrogation est utilisé avec l'ontologie COG+ qui comporte aussi les zones non administratives de.

Les zones géographiques administratives du COG sont organisées dans une structure d'arbre en utilisant une relation transitive nommée *seSubdiviseEn*. Cette relation a le sens suivant : $(x$ *seSubdiviseEn* $y) \Leftrightarrow (y \subset x)$. Ainsi, elle définit un ordre partiel sur les zones géographiques. De plus, dans l'ontologie COG cet ordre est arborescent, chaque zone n'étant une subdivision que d'une zone de niveau supérieur.

La figure 5.3 présente un extrait de l'arbre des zones géographiques du COG. Chaque nœud représente une zone géographique et chaque arc représente la relation *seSubdiviseEn*. Dans cette figure, la racine de l'arbre est le pays *France* qui est subdivisé en deux régions : *Ile-de-France* et *Poitou Charentes*. Cette dernière région est elle-même subdivisée en deux départements : *Vienne* et *Deux-Sèvres*.

Les documents sont automatiquement annotés en utilisant le COG et le COG+.

Le prédicat d'annotation est nommé *estGeolocalisePar*. L'annotation *(doc contient seg)(seg estGeolocalisePar zone)* (où *doc* représente un *document*, *seg* un *segment* du document et *zone* une *zone géographique* administrative du COG), signifie que le document *doc* contient un segment ayant des informations à propos d'une partie ou de l'ensemble de la zone géographique *zone*.

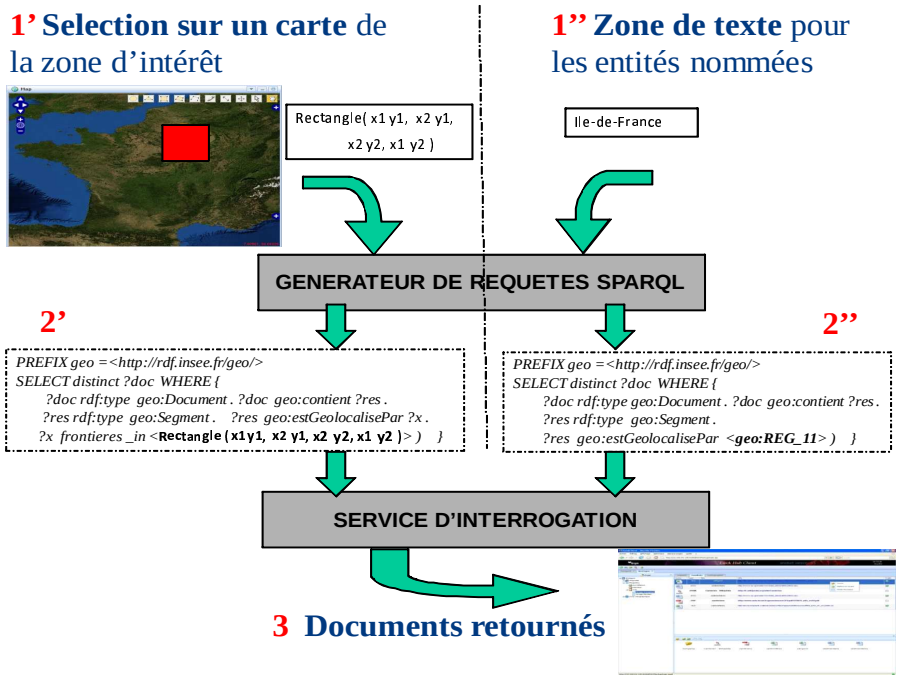


FIG. 5.2 – Illustration du processus d'interrogation e-Wok Hub.

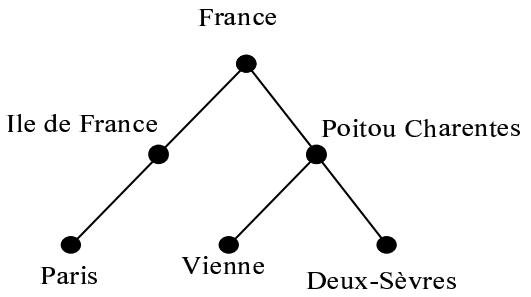


FIG. 5.3 – L'ontologie du COG : exemple de relation d'inclusion entre individus

Nous notons que ce prédicat a un comportement particulier par rapport à l'ordre *seSubdiviseEn*. Si un segment d'un document contient des informations à propos de *Paris*, il contient des informations à propos d'une partie de l' *Ile-de-France*. Ainsi par exemple, *(doc contient seg)(seg estGeolocalisePar Paris)* $\Longrightarrow$ *(doc contient seg)(seg estGeolocalisePar Ile-de-France)*.

Notons que tous les prédicats dont le co-domaine est une zone géographique n'ont pas forcément ce comportement. Par exemple, la personne qui dirige la région *Ile-de-france* ne dirige pas nécessairement le département de *Paris*, ni l'inverse. Ce comportement a une incidence sur les requêtes. En effet, si quelqu'un recherche tous les documents qui ont un rapport avec la zone géographique *zone*, le système doit raisonner sur la relation d'inclusion et retourner tous les documents annotés par les zones géographiques incluses dans *zone*.

Le modèle utilisée dans ce projet, est illustrée par la figure 5.4. Il comporte trois classes :

1. La classe *zone_ geographique*. Une *zone géographique* est caractérisée par un *nom*, un *code*, un *type* (par exemple, pays, département ou commune) et une *uri*. Une zone géographique

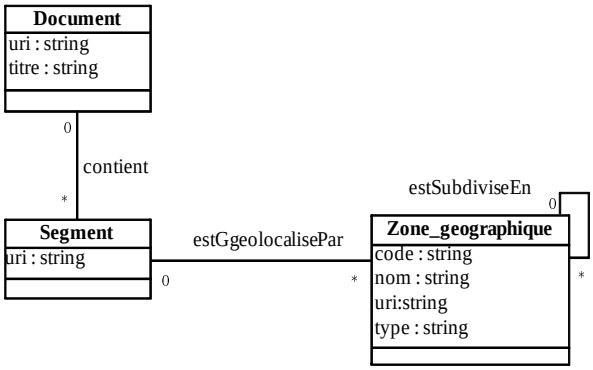


FIG. 5.4 – Indexation géographique de document.

administrative peut se subdiviser (*seSubdiviseEn*) en plusieurs autres zones géographiques. La propriété *seSubdiviseEn*, de type objet, définit sur cette classe un ordre arborescent sur l'ensemble de ses instances.

2. La classe *document*. Un document est caractérisé par une *uri* et un *titre* et, *contient* des segments.
3. La classe *segment* représente une portion d'un Document. Un segment, identifié par une URI, peut référencer (*estGeolocalisePar*) zéro ou plusieurs instances de la classe *zone_ geographique*. La propriété *estGeolocalisePar*, de type objet, liant la classe *segment* à la classe *zone_ geographique* est propagée par l'ordre *seSubdiviseEn*.

Dans le COG+, les zones géographiques sont en plus caractérisées par leur frontières. Le concept *Bassin parisien* par exemple, est une zone non administrative qui peut être décrite par un polygone représentant sa frontière. Les frontières sont représentées par des objets géométriques [38] qui définissent un ordre implicite sur les instances de la classe *zone_ geographique* dans le COG+.

Ainsi, de même que pour les zones géographiques administratives, une recherche des documents ayant des segments géolocalisés par la zone non administrative de frontière correspondante au *Bassin Parisien* du COG+, doit retourner non seulement de tels documents, mais aussi les documents ayant des segments géolocalisés par la région *Ile-de-France*, le département du *Loiret* ou toute autre zone dont la frontière est incluse dans celle de la zone *Basin Parisien*.

Le système de BDBO OntoDB2, support pour la gestion des données du COG et du COG+ doit donc offrir tous les mécanismes pour permettre à la fois de représenter les données géographiques, mais aussi de répondre efficacement aux interrogations qui peuvent être faites sur ces dernières.

1.2 Représentation de données géographiques

Afin de permettre la représentation et la manipulation efficace des données géographiques, OntoDB2 doit permettre à la fois :

– la représentation des données de domaines de valeurs des types spatiaux ;
– la représentation des données d'indexation pour faciliter leur interrogation.

[38] Les *frontières* nous ont été fournies par le BRGM (Bureau de recherches géologiques et minières (*http ://www.brgm.fr/*)) dans le cadre du projet.

1.2.1 Représentation des types spatiaux

Comme nous l'avons vu lors de la spécification d'OntoDB2 au chapitre 3, le formalisme ontologique d'OntoDB2 peut être étendu. En particulier, son système de types de données a été étendu afin de permettre de la prise en compte des types de donnés non conventionnels. Dans le cadre du projet e-Wok Hub, nous avons étendu le système de type d'OntoDB2 en intégrant le support de types de données spatiaux issus de la spécification OpenGIS [39] et, permettre ainsi la représentation de données comme les frontières des zones géographiques.

1.2.2 Représentation des données d'indexation

Les entités de la classe *zone_ geographique* étant ordonnées par la propriété *seSubdiviseEn*, les attributs nécessaires doivent être rajoutés à chaque instance pour permettre de leur associer un (ou plusieurs) label(s). Enfin, les valeurs de ces labels doivent être automatiquement calculées avant de lancer les requêtes.

1.3 Traitement efficace des requêtes

Le traitement des requêtes formulées doit prendre en compte :

- *pour l'ontologie* COG :
  1. le fait que la propriété *seSubdiviseEn*, définie sur les instances de la classe *zone_ géographique*, est une relation d'ordre ;
  2. le fait que la propriété *estGeolocaliseePar* liant les segments de documents aux instances de la classe *zone_ géographique*, soit propagée par la relation d'ordre *seSubdiviseEn* (dans l'ordre inverse).
- *pour l'ontologie* COG+
  - le fait que les frontières des zones géographiques définissent un ordre implicite sur les instances de la classe *zone_ géographique* ;
  - le fait que la propriété *estGeolocaliseePar* liant les segments de documents aux instances de la classe *zone_ géographique*, soit propagée par l'ordre implicite (dans l'ordre inverse).

Nous proposons dans la suite de ce chapitre, une formalisation permettant de sélectionner automatiquement la technique d'étiquetage à utiliser suivant le problème considéré et nous introduisons de nouvelles techniques d'étiquetage pour raisonner sur les zones spatiales et les périodes temporelles.

2 Formalisation Proposée

Pour commencer, nous caractérisons formellement le comportement des relations présentées sur l'exemple : *estGeolocaliseePar* et *seSubdiviseEn*.

Soient E et F deux ensembles, $\mathcal{R} \subset \mathrm{E} \times F$ et $\prec \subset \mathrm{F} \times \mathrm{F}$ deux relations binaires, avec $\prec$ étant une relation d'ordre, c'est-à-dire, réflexive, antisymétrique et transitive.

[39] http ://www.opengeospatial.org/

Nous disons que : $\mathcal{R}$ est *propagée par* l'ordre $\prec$ si et seulement si :
$\forall\ x \in E, \forall\ y, z \in F\ ,\ \ x\ \ \mathcal{R}\ \ y \wedge y \prec z\ \ \Rightarrow x\ \ \mathcal{R}\ \ z$
et nous appelons *fermeture transitive propagée* (FTP) de $\mathcal{R}$ par $\prec$, notée $\mathcal{R}^+_\prec$:
$\mathcal{R}^+_\prec = \{\ (x,z) \in E \times F\ \ /\ \ \exists\ y \in F\ ,\ \ x\ \ \mathcal{R}\ \ y \wedge y \prec z\}$

Si $\mathcal{R}$=*estGeolocalisePar* et $\prec$ = $seSubdiviseEn^{-1}$, (a $\prec$ b $\Longleftrightarrow$ b *seSubdiviseEn* a) pour une zone géographique donnée, l'ensemble des documents dont au moins un segment est en relation avec cette zone dans $\mathcal{R}^+_\prec$ contient tous les documents contenant des segments qui sont annotés soit par cette zone, soit par une de ses subdivisions (récursives).

2.1 Raisonnement sur les Fermetures Transitives Propagées.

Nous notons que $\mathcal{R}^+_\prec$ est la composition de la fermeture transitive de $\prec$, notée $\prec^*$, avec $\mathcal{R}$: $\mathcal{R}^+_\prec = \prec^* \circ\ \ \mathcal{R}$. Et donc, une représentation efficace de la fermeture transitive de $\prec$ permettrait d'obtenir une représentation efficace de la fermeture transitive propagée de $\mathcal{R}$. Pour cela, nous utilisons une technique d'étiquetage. Une technique d'étiquetage $\mathcal{L}$ sur *(F,*$\prec$*)* est un triplet : $\mathcal{L}$ = (D, label, less_or_eq) où :

- D est un domaine concret ordonné ($\leq$) ;
- *label* : $F \rightarrow D$ est un morphisme d'ensembles ordonnés :
  $\forall\ x,y \in F\ ,\ \ x \prec y \Rightarrow label(x) \leq label(y)$
- less_or_eq : D × D → Boolean est une fonction qui compare en temps constant deux valeurs de D :
  $\forall\ a,b \in D\ ,\ \ less\_or\_eq(a,b) \Leftrightarrow a \leq b$

Ainsi, si pour tout $y \in F$, *label(y)* est pré-calculée et stockée dans la base de données et si *less_or_eq* peut être calculée en temps constant (par exemple, par comparaison numérique ou alphanumérique), le calcul de la fermeture transitive propagée de $\mathcal{R}$ peut être fait en temps linéaire par un seul parcours de la relation $\mathcal{R}$.

2.2 Techniques d'Étiquetage Topologiques et Géométriques.

Nous présentons ci-dessous les techniques d'étiquetage topologiques et géométriques qui permettent d'indexer efficacement les relations d'ordre afin de faciliter les traitements sur ces dernières.

2.2.1 Techniques d'étiquetages topologiques

Les techniques de marquage (ou « labelling ») des classes [14] des hiérarchies ont déjà été utilisées dans d'autres environnements (bases de données XML, représentation de connaissances, langages de modélisation objets, etc.) où, elles permettent d'aboutir à des bonnes performances lors des traitements [2, 43]. La technique de marquage consiste à associer à chaque classe d'une hiérarchie des valeurs de propriétés additionnelles appelées *labels* et, reflétant l'ordre arborescent et qui permettront ainsi en une seule requête (moins coûteuse) de calculer les sous-classes ou superclasses d'une classe donnée [14]. Deux techniques d'étiquetage topologiques existent pour la définition du *label* des classes des hiérarchies arborescente.

2.2.1.1 Techniques d'étiquetages statiques

Les labels statiques couramment utilisées [43, 2] dans les systèmes actuels sont des variantes de l'étiquetage statique proposé par Dietz [21]. La technique de Dietz est bâtie sur le constat que le nœud x d'un arbre $\mathcal{A}$ est un ancêtre du nœud y ssi x apparaît avant y dans le parcours en pré ordre et après y dans le parcours en post ordre de $\mathcal{A}$. Ainsi, à chaque nœud de $\mathcal{A}$ est associé un couple de nombre (noté ici *bound1* et *bound2*) correspondant respectivement à la position du nœud dans le parcours pré ordre et post ordre. Ainsi, le nœud C_1 est un sous-nœud de C si et seulement si $\Leftrightarrow C_1$.bound1 $<$ C.bound1 $<$ C.bound2 $<$ C_1.bound2.

Agrawal [3] par exemple, a suggéré une amélioration de la technique de Dietz qui consiste à utiliser au lieu de la position pré ordre du nœud dans l'arbre, la plus petite valeur post ordre de ses descendants. La technique d'Agrawal utilise ainsi moins de chiffres que celle de Dietz. La figure 5.5 montre un exemple où les nœuds sont annotés suivant l'étiquetage de Dietz (figure 5.5-a) et l'étiquetage d'Agrawal (figure 5.5-b). Dans la figure 5.5-b, on peut voir que le nœud C_1 est un sous-nœud du nœud A car : $C_1.bound1\ (=1) \leq C.bound1\ (=1) \leq C_1.bound2\ (=1) \leq C.bound2\ (=2)$.

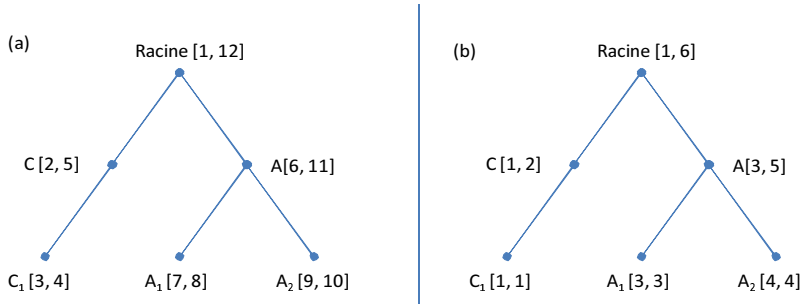


FIG. 5.5 – Exemples d'étiquetage statique

Les labels générés par les techniques statiques sont courts mais manquent de flexibilité. En effet, les valeurs pré ordre et post ordre de nombreux nœuds (voir de l'ensemble des nœuds) doivent être recalculées à l'insertion d'un nouveau nœud. Lorsque la relation d'ordre n'est pas arborescente, mais plutôt un DAG, plusieurs intervalles deviennent nécessaires. Agrawal a proposé une approche qui permet, en se basant sur l'arbre recouvrant optimal du graphe (« spanning tree »), de réduire le nombre de labels associés à un nœud.

2.2.1.2 Techniques d'étiquetages dynamiques

Les labels dynamiques [64, 42] ont été introduits afin de répondre au besoin des environnements dans lesquels les mises à jour sont fréquentes. D'une manière générale, à chaque nœud est associé un code et, le label du nœud est obtenu par concaténation de ce dernier avec les codes de tous les nœuds se situant sur son chemin absolu à partir de la racine de l'arbre. Les techniques d'étiquetage dynamique ne nécessitent pas de recalcul de labels à l'insertion de nouveaux nœuds, et donc ils présentent de meilleures performances lors des mises à jour que les techniques d'étiquetage statique. Cependant, ils génèrent des labels de grande taille (la taille des labels varie et augmente avec la profondeur du nœud) ; l'indexation par une technique étiquetage dynamique peut s'avérer

inefficace si l'index ne tient pas en mémoire. Par exemple, dans la figure 5.6 où les nœuds sont annotés suivant le codage de Dewey [64], on peut voir que le nœud C inclut dans sa subdivision le nœud C_1 car le label de C (*1.1*) préfixe le label de C_1 (*1.1*.1).

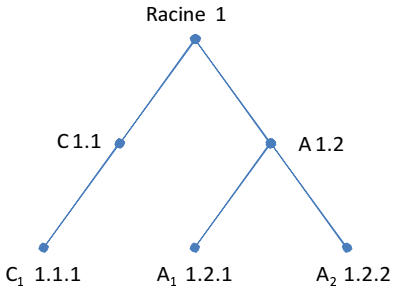


FIG. 5.6 – Exemple d'etiquetage dynamique

2.2.2 Technique d'étiquetage géométrique

La plupart des techniques de labelling qui ont été proposées utilisent la structure topologique du treillis, qui représente l'ordre sur l'espace F, pour définir les labels.

En fait, lorsque l'on raisonne sur des domaines spatiaux ou temporels, l'espace sous-jacent a non seulement une structure topologique où sont définis voisinages et connexions, mais aussi une structure géométrique : un espace vectoriel normé dans lequel sont définies positions et distances. Ainsi, la position d'une entité dans l'espace peut être utilisée pour définir son label. Par exemple, dans la figure 5.3, les rectangles englobant, tout comme les cercles englobant, pourraient être utilisés pour associer des labels aux zones géographiques. Lorsque l'on raisonne sur des périodes géologiques, la datation géologique (approximation de temps géologique), par exemple en millions d'années avant notre ère, peut être utilisée pour associer un label à cette période. Notons que, contrairement aux labels basés sur la topologie :

1. ces labels sont absolus. Ils représentent un savoir additionnel qui ne peut pas être automatiquement dérivé des instances connues de F ou des relations ; mais,
2. ils n'ont pas besoin d'être changés quand le contenu de F est mis à jour.

Tous ces divers labels peuvent être représentés comme des techniques d'étiquetage dans les BDBO permettant un raisonnement efficace sur les fermetures transitives.

Ainsi, les propriétés spatiales et temporelles sont des propriétés ontologiques primitives d'objets spatiaux ou temporels. De ce point de vue, il est raisonnable de considérer que leurs valeurs, pour une instance donnée, peuvent être soit disponibles à un endroit donné (par exemple, par un service web), soit échangées avec la description de l'instance. Mon année de naissance tout comme la géolocalisation de *Paris* sont toutes les deux des propriétés ontologiques qui sont disponibles quelque part et qui peuvent être gérées dans des sources de données à base ontologique. Une difficulté est que les descriptions géométriques peuvent impliquer des structures de données complexes disponibles seulement dans des systèmes spécifiques (par exemple, un système d'information géographique (SIG)). En fait, d'importants raisonnements géométriques nécessitent

seulement des données très simples. L'inclusion spatiale d'objets convexes peut être évaluée en utilisant les rectangles ou cercles englobant. La précédence temporelle nécessite seulement de comparer deux valeurs réelles ou deux intervalles. Ainsi, il est à la fois possible de restreindre l'ensemble des représentations géométriques autorisées et de supporter un large éventail de raisonnements (approximatifs) spatiaux et temporels. Notre suggestion est de supporter seulement des intervalles (une dimension (1D)), des rectangles et des cercles (deux dimensions (2D)).

3 Conception et Implémentation

Cette section présente comment notre approche a été implémentée dans OntoDB2 (et donc peut être implémentée dans différentes architectures de BDBO) pour permettre l'automatisation du mécanisme de propagation de propriétés.

Nous faisons maintenant l'hypothèse que E et F sont deux classes ontologiques. Pour représenter le fait qu'une propriété $\mathcal{R} : E \times F$ est propagée par un ordre partiel $\prec$ sur F, nous devons représenter :

1. le fait que $\prec$ est un ordre,
2. la technique d'étiquetage de cette ordre $\mathcal{L}$ = *(D, label, less_or_eq)*, et
3. le fait que $\mathcal{R}$ doit être propagée par $\mathcal{L}$.

Les formalismes d'ontologies existants ne fournissent pas de primitive pour représenter ces trois informations. En conséquence, les langages d'ontologies ainsi que les BDBO doivent être étendus. Dans une base de données OWL par exemple, la première information nécessite d'ajouter une nouvelle valeur (nommée *orderProperty*) à l'ensemble des caractéristiques de propriétés OWL (*transitiveProperty*, *symmetricProperty*, etc.) puisque *antisymmetric* n'est disponible ni dans OWL1 ni dans OWL2. La troisième information nécessite d'ajouter une nouvelle valeur (nommée *propagatedBy*) à l'unique relation entre propriétés existante en OWL1 (*inverseOf*). Par ailleurs, pour la seconde information, nous devons créer deux (méta-)tables additionnelles dans la BDBO. La première décrit les techniques d'étiquetage disponibles dans la BDBO. La seconde définit à quelle technique d'étiquetage est assignée un propriété d'ordre donnée. Ces deux tables sont des extensions des BDBO requises par notre modèle d'étiquetage. Dans la suite, nous présentons les grandes lignes du processus d'implémentation de notre approche et discutons des problèmes de représentation. Les différentes étapes de cette implémentation peuvent être réalisées à la fois sur les architectures de BDBO de types 2 et 3.

3.1 Extension de la partie formalisme d'ontologies des BDBO

Dans cette section, nous présentons les tables n-aires requises pour enregistrer les informations nécessaires. Les tables *property_characteristic* (table 5.1) et *property_to_property* (table 5.2) représentent les informations que nous proposons d'ajouter aux formalismes d'ontologies. Les tables *labeling_scheme* (table 5.3) et *property_schemes* (table 5.5) sont des tables systèmes.

La table 5.1 *property_characteristic* contient les caractéristiques des propriétés. L'extension des formalismes d'ontologie consiste à représenter *orderProperty* comme une caractéristique.

TAB. 5.1 – colonnes de la table *property_ characteristic*

Colonne	**Description**
propertyId	référence l'unique identifiant de la propriété dans la table des propriétés.
characteristic	la caractéristique de la propriété (par exemple *orderProperty, symmetricProperty*, etc.)

La table 5.2 *property_ to_ property* contient les relations entre deux propriétés. L'extension des formalismes d'ontologie consiste à permettre de représenter la relation *propagatedBy* entre une propriété et une autre qui définit un ordre.

TAB. 5.2 – colonnes de la table *property_ to_ property*

Colonne	**Description**
propertyId	l'identifiant unique de la propriété propagée.
orderId	l'identifiant unique de la propriété dans la table *property_ characteristic*
relationName	le nom de la relation sémantique liant deux propriétés ; par exemple, *propagatedBy by, inverseOf*
direction	la direction de propagation par rapport à la relation d'ordre. Les valeurs autorisées sont *direct* et *reverse*.

Lorsqu'une technique d'étiquetage géométrique est utilisée et qu'un label est indiqué avec une donnée d'instance, par exemple comme un rectangle englobant, la relation d'inclusion peut souvent être implicite : elle doit être calculée par l'inclusion de formes géométriques définies par des labels géométriques. Afin de pouvoir représenter la propagation par cet ordre implicite, une propriété virtuelle définissant cet ordre implicite doit être introduite dans l'ontologie. Le formalisme d'ontologie doit donc être étendu afin de représenter le fait qu'une propriété soit virtuelle. Cette extension se traduit par l'ajout d'un attribut de type booléen (*virtual*) à la classe des propriétés de type objet du formalisme d'ontologie. Dans ce cas, la propriété virtuelle est tout d'abord associée à la caractéristique *orderProperty* dans la table *property_ characteristic* puis, le *orderId* de la propriété propagée est associé à la propriété virtuelle. Cela permet donc de spécifier que la propriété identifiée par *propertyId* est propagée par l'ordre d'inclusion des formes géométriques associé à la propriété virtuelle. Les colonnes qui contiennent les labels géométriques sont nommées comme spécifié dans la table 5.3. Pour une relation *propagatedBy*, la colonne *direction* indique si la propagation est faite de manière *directe* (la même direction que la propriété d'ordre) ou de manière *inverse*. Par exemple, les lois applicables en *Ile-de-France* incluent celles définies dans des zones qui englobent l'*Ile-de-France* ; ceci implique une propagation directe par rapport à l'ordre *seSubdiviseEn*, contrairement à la propriété *estGeolocaliseePar* où la propagation est faite de manière inverse. Par convention, l'ordre direct des relations géométriques est l'ordre croissant.

La table 5.3 *labeling_ scheme* contient les informations à propos des différentes techniques d'étiquetage disponibles dans la BDBO. Cette table est supposée être définie par l'administrateur de la base de données (DBA). Les quatre premiers attributs indiquent comment est représenté physiquement chaque étiquetage. Par ailleurs, les étiquetages géométriques doivent avoir une représentation prédéfinie pour être reconnus lors de la lecture des données. Ils sont donc définis

dans la table 5.4. Ceci permet donc à notre BDBO de supporter les ontologies géographiques comme définies par Cullot et al. [16] puisque cela permet de représenter des types géométriques et leurs fonctions de manipulation (et en particulier les types rectangle et cercle et les fonctions *less_ or_ eq* les concernant), de localiser les objets dans l'espace et de représenter des objets spatiaux. Notre proposition ne nécessite néanmoins pas du tout de disposer de toute la puissance d'un SIG pour mettre en œuvre les mécanismes que nous proposons. En effet, notre solution permet de fournir des calculs approximatifs en réalisant des opérations simples sur les rectangles ou les cercles englobant des objets spatiaux.

TAB. 5.3 – Colonnes de la table *labeling_ scheme*

Colonne	**Description**
schemeId	référence l'identifiant unique associée à la technique d'étiquetage
numberOfColumns	le nombre de colonnes utilisées pour représenter le domaine D (par exemple, 2 pour la technique d'étiquetage par intervalles)
listColumnsSuffixes	une liste de suffixes de colonnes utilisées pour représenter D (par exemple, { *bound1*, *bound2*})
listColumnsTypes	une liste de types de colonnes associés aux noms de colonnes dans *listColumns-Suffixes* (par exemple, {*int*, *int*})
label	le nom optionnel de la fonction SQL/PSM à utiliser pour calculer le label associé aux instances dont le label vaut NULL. Cette fonction est appelée sur F chaque fois qu'une ou plusieurs nouvelles instances de F sont ajoutées dans la base de données au sein d'une même transaction. Ce label n'existe pas (NULL) lorsqu'il doit être fourni de l'extérieur pour chaque instance (par exemple, pour des techniques d'étiquetage géométriques).
less_ or_ eq	le nom de la fonction SQL/PSM à utiliser pour évaluer si une instance est inférieure ou égale à un autre pour l'ordre défini par *propertyId*. Si $\mathcal{L}$ est la technique d'étiquetage par intervalles sur l'espace F, *i1* et *i2* sont deux instances, $i2 \prec i1$ alors l'appel de fonction *less_ or_ eq(i2.bound1, i2.bound2,i1.bound1, i1.bound2)* retourne *true*.
defaultScheme	une valeur booléenne. La technique d'étiquetage par défaut associée à une nouvelle propriété définissant un ordre.

TAB. 5.4 – Techniques d'étiquetage prédéfinies dans la table *labeling-scheme*

schemeId	**numberOf-Columns**	**listColumns-Suffixes**	**listColumns-Types**	...
geo_ rectangle	4	{xmin, xmax, ymin, ymax}	{float, float, float, float}	...
geo_ circle	3	{xcenter, ycenter, radius}	{float, float, float}	...
geo_ interval	2	{bound_1, bound_2}	{float, float}	...

La table 5.5, *property_ schemes*, contient des informations à propos des différentes techniques d'étiquetage associées à chaque propriété. Cette table est automatiquement générée par le sys-

tème. Lorsqu'une propriété d'ordre est introduite dans la table 5.1, la technique d'étiquetage par défaut définie dans la table 5.3 est automatiquement implémentée et une ligne dans la table 5.5 est aussi ajoutée. Le DBA peut changer la technique d'étiquetage par défaut si nécessaire.

TAB. 5.5 – Colonnes de la table *property_ schemes*

Colonne	**Description**
propertyId	l'identifiant unique de la propriété d'ordre dans la table *property_ characteristic*.
schemeId	l'identifiant unique de la technique d'étiquetage
listProperties	la liste des identifiants des propriétés associées à *listColumnsSuffixes* dans la classe F
activeScheme	une valeur booléenne. *true* si la technique d'étiquetage est activée.

3.2 Représentation des Instances

Les instances des classes de l'ontologie seront représentées selon la structure de données utilisée dans chaque BDBO. Pour automatiser la génération de propriétés spécifiques utilisées pour stocker les labels, elles sont gérées comme les autres propriétés et elles sont initialisées à NULL si aucune valeur n'est disponible.

3.3 Traitement des Requêtes

Notre but est de réaliser automatiquement un raisonnement numérique pour les requêtes utilisant des propriétés qui ont pour caractéristique d'être des relations d'ordre ou d'être propagées par un ordre. Chaque requête doit être traitée par l'interpréteur de requête de la BDBO de la manière suivante :

- **Identification de la catégorie de la requête** : chaque requête doit être analysée au niveau ontologique pour déterminer si elle implique une propriété présentant une caractéristique particulière. Ceci peut être réalisé en utilisant les tables *property_ characteristic* et *property_ to_ property* ;
- **Interprétation de la requête** : si la requête n'implique pas de propriétés particulières, elle subira le traitement habituel ; sinon, la requête sera d'abord transformée en une requête numérique en utilisant les informations contenues dans les tables *labeling_ scheme* et *property_ schemes*. Cette traduction dépend également de la représentation des annotations. La requête réécrite résultante sera ensuite exécutée automatiquement par la base de données.

La figure 5.7 illustre les différentes étapes suivies pour le traitement des requêtes. Dans ce qui suit, nous décrivons l'implémentation de notre approche réalisée sur la BDBO OntoDB2.

4 Application à l'ontologie du COG dans la BDBO OntoDB2

Cette section décrit comment notre approche a été effectivement implémentée sur la BDBO OntoDB2 avec les ontologies du COG et COG+ [40] qui étend le COG en définissant les frontières des zones géographiques dont le domaine de valeurs sont des types géométriques.

[40] Les frontières des zones géographiques non disponibles, nous ont été fournies par le BRGM

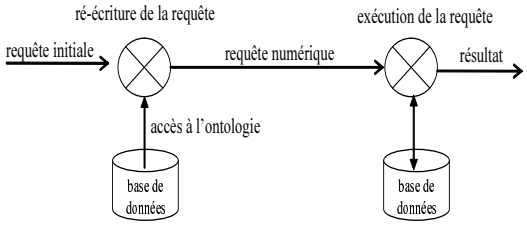


FIG. 5.7 – Étapes de traitement des requêtes.

4.1 Ontologie

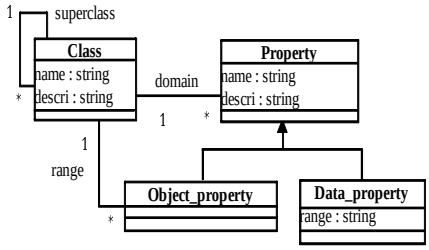
L'ontologie est stockée dans la partie ontologie d'OntoDB2. Après le chargement de l'ontologie dans OntoDB2, cette dernière a été modifiée en spécifiant (1) l'attribut *is_order* à vrai pour la propriété d'ordre *seSubdiviseEn* de la classe *zone_geographique*, (2) l'attribut *propagatedBy* à la propriété d'ordre *seSubdiviseEn* pour la propriété propagée *estGeolocalisePar* de la classe *Segment*, (3) l'attribut *direction* à '*reverse*' et, (4) l'ajout des propriétés de types géographiques pour représenter les frontières des instances de la classe *zone_geographique*. La figure 5.8-b présente le contenu des tables de la partie ontologie en utilisant un formalisme d'ontologie simplifié. Pour des raisons de lisibilité, nous n'utilisons pas les identifants internes lors des références (clés étrangères) mais plutôt, les noms des classes et propriétés. Rappelons également que dans OntoDB2, la structure de représentation des ontologies est une structure plate (où toute hiérarchie est représentée par une table). De même, comme nous l'avons vu dans le chapitre 3, OntoDB2 utilise un attribut booléen pour renseigner les caractéristiques de propriété. Ainsi, dans OntoDB2, les attributs *is_order* et *direction* permettent respectivement de spécifier si une propriété définie une relation d'ordre et le sens de propagation d'une relation par rapport à une relation d'ordre et, dans ce dernier cas, l'attribut *propagatedBy* permet de référencer la propriété d'ordre associée. Cependant, pour des raisons de simplicité, nous n'utilisons pas ici cette représentation plate. Ce formalisme d'ontologie ainsi que son extension par les tables *property_characteristic* et *property_to_property* est stocké dans la partie méta-schéma (figure 5.8-a) (cette représentation n'apparaît pas ici pour éviter de complexifier la figure). Ceci permet d'automatiser la génération de la structure de la partie ontologie en utilisant une transformation de modèles. La figure 5.8-c présente le contenu des tables systèmes *labeling_scheme* et *property_schemes*.

La table *property_characteristics* montre que la propriété *seSubdiviseEn* (de propertyId op3) défini un ordre.

La table et *property_to_property* montre également que la propriété *estGeolocalisePar* (ayant pour propertyId op4) est propagée par la propriété *seSubdiviseEn*.

La table système *labeling_scheme* définit deux techniques d'étiquetage.

1. La première technique d'étiquetage, de schemeId *sch5*, adaptée pour les instances du COG, est un étiquetage topologique ; elle nécessite deux colonnes (*bound1* et *bound2*) définies dans la table *listColumnsSuffixes*. La function *interval* permet de calculer automatiquement les valeurs des colonnes *bound1* et *bound2* des instances de la classe *zone_geographique* pour cette technique d'étiquetage et, la function *include* permet de déterminer la relation d'ordre entre deux individus. Cette première technique est définie comme la technique par défaut du système.



(a) Modèle d'ontologies

Class		
id	name	its_superclass
c1	Zone_geographique	
c2	Segment	
c3	Document	

Object_property				
id	name	domain	range	virtual
op3	seSubdiviseEn	Zone_geographique	Zone_geographique	FALSE
op4	estGeolocalisePar	Segment	Zone_geographique	FALSE
op5	contient	Document	Segment	FALSE

PROPERTY_TO_PROPERTY			
propertyId	orderId	relationName	direction
estGeolocalisePar	seSubdiviseEn	propagated by	direct

PROPERTY_CHARACTERISTIC	
propertyId	characteristic
seSubdiviseEn	orderProperty

Data_property			
id	name	domain	range
dp5	code	Zone_geographique	string
dp6	uri	Zone_geographique	string
dp7	*seSubdiviseEn_bound1*	Zone_geographique	int
dp8	*seSubdiviseEn _bound2*	Zone_geographique	int
dp9	*seSubdiviseEn _xmin*	Zone_geographique	float
dp10	*seSubdiviseEn _xmax*	Zone_geographique	float
dp11	*seSubdiviseEn _ymin*	Zone_geographique	float
dp12	*seSubdiviseEn _ymax*	Zone_geographique	float
dp13	nom	Zone_geographique	string
dp14	type	Zone_geographique	string
dp15	uri	Document	string
dp16	titre	Document	string
dp17	uri	Segment	string

(b) Partie ontologie

PROPERTY_SCHEMES		
propId	schemeId	activeScheme
seSubdiviseEn	sch5	TRUE
seSubdiviseEn	*geo_rectangle*	FALSE

LABELING_SCHEME				
schemeId	numberOfColumns	label	less_or_eq	defaultScheme
sch5	2	interval	include	TRUE
geo_rectangle	4	NULL	MBR_Contains	FALSE

LISTPROPERTIES		
propId	Properties	order
seSubdiviseEn	seSubdiviseEn_bound1	0
seSubdiviseEn	seSubdiviseEn_bound2	1
estGeolocalisePar	seSubdiviseEn_xmin	0
estGeolocalisePar	seSubdiviseEn_xmax	1
estGeolocalisePar	seSubdiviseEn_ymin	2
estGeolocalisePar	seSubdiviseEn_ymax	3

LISTCOLUMNSTYPES		
schemeId	ColumnsTypes	order
sch5	int	0
sch5	int	1
geo_rectangle	float	0
geo_rectangle	float	1
geo_rectangle	float	2
geo_rectangle	float	3

LISTCOLUMNSSUFFIXES		
schemeId	ColumnsSuffixes	order
sch5	bound1	0
sch5	bound2	1
geo_rectangle	xmin	0
geo_rectangle	xmax	1
geo_rectangle	ymin	2
geo_rectangle	ymax	3

(c) Tables systèmes

FIG. 5.8 – COG - ONTODB2 : Partie ontologie

2. La seconde technique d'étiquetage, de *schemeId *geo_ rectangle**, adaptée pour les instances du COG+, est un étiquetage géométrique ; elle nécessite quatre colonnes (*xmin, ymin, xmax et ymax*) définies dans la tables *listColumnsSuffixes*. Pour cette seconde technique, les valeurs de labels doivent être fournies car aucune fonction de calcul automatique ne peut être réalisée par le système : il s'agit de propriétés natives de l'ontologie COG. La fonction *mbr_ contains* permet de déterminer la relation d'ordre entre deux individus pour cette technique d'étiquetage.

La table système *property_ schemes* permet d'associer à la propriété *seSubdiviseEn* (1) la technique d'étiquetage topologique de *schemeId sch5* ; cette dernière est définie ici comme la technique active dans le système. Et, (2) la technique d'étiquetage géométrique de *schemeId *geo_ rectangle**.

La table *Data_ Property* (figure 5.8-b) montre que les propriétés *seSubdiviseEn_ bound1, seSubdiviseEn_ bound2* (respectivement *seSubdiviseEn_ xmin, seSubdiviseEn_ xmax, seSubdiviseEn_ ymin, seSubdiviseEn_ ymax*) nécessaires pour la technique d'étiquetage de schemeId *sch5* (respectivement *geo_ rectangle**) sont associées au co-domaine *zone_ geographique* de la propriété propagée *estGeolocalisePar*. Ces nouvelles propriétés seront utilisées à la fois pour permettre au système (1) de calculer automatiquement les valeurs des étiquettes pour chaque instance en faisant appel à la fonction *interval* pour la technique d'étiquetage de *schemeId sch5* (pour la technique d'étiquetage de *schemeId *geo_ rectangle**, les valeurs des coordonnées des rectangles englobants ont été récupérées d'une autre source et entrées dans OntoDB2). Et, (2) de déterminer en faisant appel à la fonction *include* (respectivement MBR_ *contains*) la relation d'ordre entre deux individus de la classe *zone_ geographique* pour la technique d'étiquetage de *schemeId sch5* (respectivement *geo_ rectangle**).

4.2 Données

Une fois que l'ontologie COG et les caractéristiques des propriétés et des relations ont été représentées, les instances doivent également être représentées dans la partie données. Les figures 5.9-a et 5.9-b illustrent la représentation des instances et la représentation des annotations. Les instances des classes de l'ontologie sont représentées selon la structure horizontale (figure 5.9-a). Pour la représentation des annotations, nous avons utilisée une approche par vue matérialisée. Ainsi, comme le montre la figure 5.9-b, la vue table *Segment_ _ estGeolocalisePar* contient les colonnes *rid* et *estGeolocalisePar_ ref* et également, les colonnes *seSubdiviseEn_ bound1* et *seSubdiviseEn_ bound2* qui permettent de représenter la technique d'étiquetage active dans le système. Les labels pour l'indexation de la relation d'ordre *seSubdiviseEn* ont été automatiquement calculés en appelant la fonction *interval*.

4.3 Traitement des requêtes

Nous nous intéressons maintenant à la transformation de requêtes pour qu'elles puissent être exécutées efficacement sur OntoDB2 en utilisant les capacités de raisonnement numérique de PostgreSQL.

Zone_geographique

id	nom	type	...	seSubdiviseEn_bound1	seSubdiviseEn_bound2	seSubdiviseEn_xmin	seSubdiviseEn_ymin	seSubdiviseEn_xmax	seSubdiviseEn_ymax
z1	France	country		1	12	...	...	...	...
z2	Ile de France	department		2	5	...	...	...	...
z3	Paris	town		3	4	...	...	...	...
z4	Poitou-Charentes	department		6	11	...	...	...	...
z5	Poitiers	town		7	8	...	...	...	...
z6	La Rochelle	town		9	10	...	...	...	...

Document

id	uri	titre
d1	...	...
d2	...	...
d3	...	...

Segment

id	uri
s1	...
s2	...
s3	...
s4	...
s5	...
s6	...

Document__contient

rid	Contient_ref	...
d1	s1	...
d1	s2	...
d2	s3	...
d2	s4	...
d2	s5	...
d3	s6	...

Zone_geographique__seSubdiviseEn

rid	seSudbiviseEn_ref	...
France	Ile de france	...
France	Poitou-Charentes	...
Ile de france	Paris	...
Poitou-Charentes	La rochelle	...
Poitou-Charentes	Poitiers	...

(a)instances

Segment__estGeolocalisePar

rid	estGeolocalisePar_ref	seSubdiviseEn_bound1	seSubdiviseEn_bound2	...
s1	Ile de France	1	2	...
s2	Poitiers	3	3	...
s3	La Rochelle	4	4	...
s4	Poitou Charentes	3	5	...

(b)annotations

FIG. 5.9 – COG - OntoDB2 : la partie données

Requête SPARQL

```
PREFIX geo: <http://rdf.insee.fr/geo/>
SELECT ?doc
WHERE {
(?doc rdf:type geo:Document)
(?doc geo:contient ?seg)
(?seg rdf:type geo:Segment)
(?seg geo:estGeolocalisePar ?x)
(?x rdf:type geo : zone_geographique)
FILTER (? x = <geo:REG_11>) }
```

Requête re-écrite

```
SELECT doc.uri
FROM document doc, doc.contient as seg,
seg.estGeolocalisePar as geo,
WHERE geo.uri= ''REG_11''
USING NAMESPACE "http://rdf.insee.fr/geo/"
```

Requête SQL

```
SELECT distinct doc.uri
FROM document doc, document__contient as cont,
segment__estGeolocalisePar as anno, zone_geographique as x
WHERE (doc.rid = cont.rid)
AND (cont.contient_ref = anno.rid)
AND (x.uri= 'REG_11')
AND (include(anno.seSubdiviseEn_bound1, anno.seSubdiviseEn_bound2,
x.seSubdiviseEn_bound1, x.seSubdiviseEn_bound2));
```

FIG. 5.10 – COG - OntoDB2 : exemple de réécriture d'une requête

La figure 5.10 montre le traitement d'une requête SPARQL relativement simple utilisée dans le projet e-Wok Hub. Cette requête permet de rechercher tous les documents dont les segments sont géolocalisés par la zone géographique *Ile-de-France*. La requête à traiter est d'abord réécrite en utilisant l'interpréteur de requête ontologique de OntoQL. Ensuite, elle est traduite en SQL et réécrite suivant qu'elle nécessite ou non un traitement spécial. Dans le cas présent, la propriété *estGeolocalisePar* est propagée de manière inverse par l'*object_property seSubdiviseEn*. En conséquence, la requête doit être réécrite pour qu'elle utilise la technique d'étiquetage active (technique de *schemeId sch5* associée à la propriété *estGeolocalisePar* dans la table *property_schemes*). Cette réécriture est présentée sur la figure 5.10. Suivant les informations contenues dans la table *labeling_schemes*, la réécriture utilisent la fonction *include* sur les colonnes suffixées par *bound1* et *bound2* pour déterminer si un document contient des segments annotés par une zone géographique incluse dans l'*Ile-de-France*.

Notons que la même cette démarche s'appliquerait à l'identique dans le cas où la technique d'étiquetage **geo_rectangle** serait la technique active dans le système.

Conclusion

La réalisation de la vision du Web Sémantique requiert des outils de gestion d'ontologies qui passent à l'échelle en terme de quantité de données gérées et qui réalisent des opérations de raisonnement sur des ontologies en un temps de réponse acceptable. Le premier problème a été résolu par l'utilisation des bases de données comme structure de persistance. Le second problème reste ouvert dans bien des cas. Dans ce chapitre, nous proposons une approche qui consiste à remplacer un raisonnement déductif par des raisonnements numériques ou alphanumériques portant sur des indexes adaptés. Nous avons d'abord décrit, au travers d'un exemple, le problème de la propagation d'une propriété par une autre définissant une relation d'ordre. Les capacités de raisonnement fournies par les architectures de BDBO usuelles sont principalement basées sur la représentation explicite de tous les faits qui peuvent être déduits par un raisonneur. Cette approche rend difficile, ou même impossible, la mise à jour de la base de données. Comme alternative nous avons formalisé le problème de propagation de propriété et, proposé une approche qui consiste à enrichir les instances d'ontologie utilisées comme annotations avec de nouvelles valeurs de propriétés de manière à pouvoir remplacer un raisonnement déductif par un traitement de requêtes numériques, ou alphanumériques.

Nous avons considéré deux types de raisonnement : les raisonnements par transitivité sur des ensembles partiellement ordonnés et des raisonnements sur la composition de deux propriétés, la seconde étant transitive. Ces cas englobent l'évaluation de propriétés liées à une taxonomie, les raisonnements de subsumption et les raisonnements d'inclusions spatiales et temporelles. Nous avons proposé une formalisation qui permet de caractériser ces cas au niveau des ontologies. Ceci permet au système d'une BDBO d'implémenter dynamiquement le raisonnement numérique lorsque de telles ontologies sont chargées. Cette formalisation nécessite trois types d'informations, chacune d'entre elles correspondant à une extension des modèles d'ontologies existants par :

- le fait qu'une propriété définit un *ordre*, c'est-à-dire qu'elle soit transitive, réflexive et antisymétrique ou un *ordre arborescent*, c'est-à-dire qu'elle implique également l'unicité du majorant (respectivement du minorant) direct ;
- le fait qu'une propriété puisse être *propagée par* une autre propriété transitive ;
- une technique d'étiquetage permettant de spécifier le type d'étiquetage qui doit être utilisé pour remplacer le raisonnement déductif par un raisonnement numérique ou alphanumérique.

Deux types de techniques d'étiquetage existent. Les techniques d'étiquetage topologiques qui correspondent à différentes techniques proposées pour des structures d'arbre. Ce type de label peut être calculé par le système de la BDBO. Notre approche permet ainsi de spécifier de manière déclarative la technique d'étiquetage qui doit être automatiquement utilisée pour une propriété donnée et de définir une technique d'étiquetage par défaut. Les techniques d'étiquetage géométriques sont utilisées pour le raisonnement spatial ou temporel. Les labels doivent être fournis au système par l'intermédiaire de propriétés ontologiques. Notre approche permet également de décrire où et comment traiter ces propriétés. Dans les deux cas, l'interpréteur de requêtes de la BDBO peut réécrire automatiquement les requêtes en utilisant les labels. Cette approche a été implémentée sur la BDBO OntoDB2. Nous avons présenté les approches qui peuvent être utilisées pour réécrire les requêtes.

L'approche que nous proposons permet d'éviter la matérialisation de tous les faits déduits.

Ceci est particulièrement important dans le cas d'utilisation que nous avons présenté. En effet, dans ce cas, la matérialisation des faits déduits consisterait à multiplier la taille de la table d'annotations, déjà très importante compte tenu du volume de documents à traiter, par un facteur proche de la profondeur de la structure hiérarchique considérée (ce qui est important si on envisage la gestion des zones géographiques du monde).

OntoDB2 propose d'autres mécanismes de gestion des inférences en exploitant les capacités des bases de données et, l'interpréteur de requête ontologique. Dans le chapitre suivant, nous présentons le validation d'OntoDB2.

Chapitre 6

Validation d'OntoDB2

Sommaire

Introduction

Nous présentons dans ce chapitre la validation expérimentale de la BDBO OntoDB2 implantée sur le SGBD PostgreSQL. Nous avons réalisé la validation d'OntoDB2 sur trois bancs d'essai.

Comme nous l'avons vu au chapitre 3, OntoDB2 visent à répondre à trois besoins :

1. la flexibilité et l'efficience du formalisme d'ontologie ;

2. la flexibilité du système de types de données ;

3. l'accès efficace aux données canoniques et non canoniques par migration d'instances.

Concernant le premier point, nous avons déjà expliqué au chapitre 4 (cf. section 1.1) comment les techniques d'IDM permettaient d'étendre simplement le formalisme, tant de l'ontologie que du système de type de données. Nous souhaitons donc valider ici l'efficacité, pour le traitement d'ontologies PLIB, du choix de la structure plate de représentation que nous avons présenté au chapitre 4 pour réduire la complexité du niveau ontologique. Les essais que nous avons réalisés ont porté sur le parcours de la relation de subsumption entre les classes de l'ontologie. Nous avons comparé les temps obtenus entre la structure de représentation plate d'OntoDB2 et l'approche mise en œuvre dans OntoDB où le modèle d'ontologie PLIB est traduit à l'identique dans le SGBD PostgreSQL. Nous avons pour ces tests utilisé trois ontologies PLIB utilisés effectivement dans le contexte industriel.

Nous avons également réalisé la validation de l'approche de raisonnement par indexation automatique des relations d'ordre que nous avons présentée au chapitre 5. Nous avons utilisé pour cela, les données de l'ontologie COG. Les requêtes utilisées dans ce banc d'essai ont été définies afin de permettre de comparer l'efficacité temporelle de la méthode proposée. Nous avons comparé donc en terme de temps de réponse, les performances d'OntoDB2, du système SOR et du système d'Oracle.

Enfin, nous avons réalisé la validation de l'approche de représentation des données au sein d'OntoDB2. Pour cela nous nous sommes intéressés à (1) accéder aux données d'une classe canonique dont tout ou partie des instances reçues étaient classifiées comme des instances de classes définies exprimées à partir de cette classe canonique et, (2) accéder aux instances d'une classe définie à partir de la vue qui lui est associée. Nous avons utilisé, pour ce dernier cas, l'ontologie UOB (Oniversity Ontology Benchmark). Nous avons comparé en termes de temps de réponse et de complétude, les performances d'OntoDB2 avec le système SOR qui adopte une approche par saturation. Les requêtes utilisées dans ce banc d'essai ont été définies afin de permettre la validation des choix adoptées dans OntoDB2 pour la gestion des données des classes canoniques et non canoniques des points de vue complétude et temporel.

Ce chapitre s'organise comme suit. Dans la section 1, nous rappelons les mécanismes de flexibilité mis en œuvre dans OntoDB2. Puis nous comparons le temps de navigation dans les classes et propriétés. Dans la section 2, nous présentons la validation des mécanismes de flexibilité du système des types de données d'OntoDB2. Puis nous présentons les résultats numériques obtenus sur l'ontologie COG du second banc d'essai. La section 3, présente l'évaluation de l'approche de gestion des données dans OntoDB2 par rapport au système SOR sur l'ontologie UOB du troisième banc d'essai. Dans cette section, nous commençons par (1) une description du banc d'essai que nous avons utilisé, (2) les requêtes que nous avons exécutées et (3) les résultats obtenus. Enfin nous terminons ce chapitre par une conclusion.

1 Flexibilité et efficience de la représentation ontologique

1.1 Flexibilité de représentation

Du point de vue flexibilité du formalisme d'ontologie, nous avons pu introduire par extension du formalisme noyau que nous avons défini au chapitre 3, de nouvelles constructions spécifiques du formalisme OWL Lite par exemple. Nous avons, en instanciant la méta-représentation du formalisme d'ontologie, introduit des constructions d'utilité générales telles que les constructeurs d'équivalence conceptuelle et les caractéristiques de propriétés. Ensuite, nous avons utilisé les programmes génériques que nous avons définis au chapitre 5 pour générer automatiquement les tables de représentation des constructions du formalisme d'ontologie ainsi que les API de manipulation par les applications. L'approche d'IDM que nous avons utilisé permet d'atteindre l'objectif de flexibilité du système. En effet, la programmation des différents modules étant réalisée au niveau du méta-modèle EXPRESS du formalisme d'ontologie, l'évolution du formalisme d'ontologie n'a pas d'impact sur les modules développés.

1.2 Efficience de représentation

Du point de vue efficience de représentation, afin de supporter en particulier les ontologies PLIB de façon efficace, nous défini une structure de représentation plus simple du formalisme d'ontologie tout en gardant à l'identique son pouvoir d'expression. Cette transformation à fait diminuer le nombre d'entités de 217 à 43 et le nombre de tables de représentation de 568 à 147, réduisant ainsi considérablement la complexité du schéma de représentation des ontologies. Également le schéma d'accès pour les applications que nous avons défini possède 117 classes au lieu de 217 initialement.

Afin de valider la solution que nous avons adoptée, nous avons réalisé des expérimentations sur de véritables ontologies de domaine développées en PLIB. Les tests que nous avons réalisés sur la partie ontologie consistent à mesurer l'incidence de la mise à plat de la structure de représentation du formalisme d'ontologie. Pour cela, nous avons mesuré le temps nécessaire pour l'affichage de la hiérarchie des classes et le temps d'accès à l'ensemble des propriétés visibles d'une classe. Nous avons comparé les résultats obtenus avec OntoDB2 et la BDBO OntoDB. Avant de présenter les résultats obtenus, nous rappelons brièvement la structure de représentation des ontologies dans les BDBO OntoDB2 et OntoDB.

1.2.1 Rappel sur le schéma des ontologies d'OntoDB2 et OntoDB

Pour illustrer la représentation des ontologies dans les BDBO OntoDB2 et OntoDB, nous utilisons le modèle d'ontologie simplifié de la figure 6.1-a, et l'ontologie de la figure 6.1-b définie suivant ce modèle d'ontologie. Cette ontologie comporte trois classes : *C1*, *C2* et *F1*. *C2* est une sous-classe de *C1* et *F1* est un vue de *C1*. Cette ontologie comporte également les propriétés *P1* et *P2* qui s'appliquent respectivement aux classes *C1* et *F1*.

1.2.1.1 Structure des ontologies d'OntoDB2

OntoDB2 adopte, pour la représentation des ontologies, une structure de représentation plate. Comme le montre la figure 6.2, chaque hiérarchie est représentée par une seule table avec une

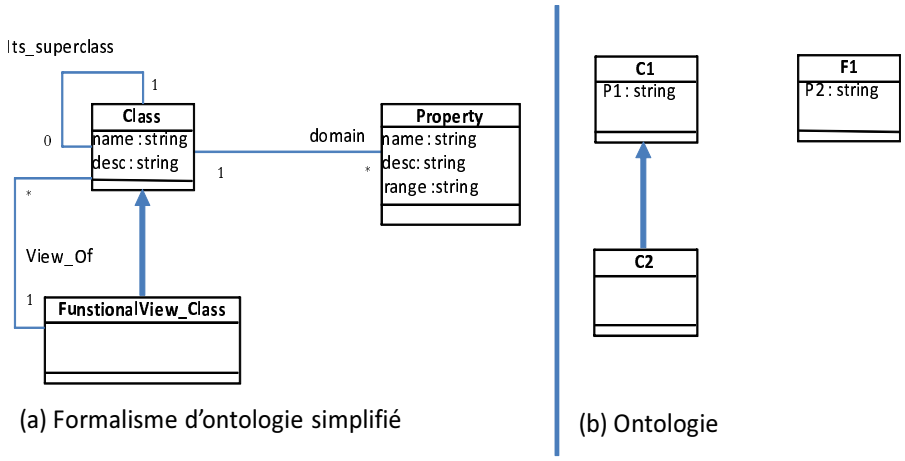


FIG. 6.1 – Formalisme d'ontologie simplifié

colonne discriminante (*rtype*). Rappelons cette transformation a fait diminuer le nombre d'entités de 217 à 43 et le nombre de tables de 568 à 147. De plus, afin d'optimiser les traitements sur les classes et les propriétés, la relation d'héritage simple (*its_superclass*) est indexée par une technique d'étiquetage topologique et, les valeurs des colonnes étiquettes (*bound1*, *bound2*) de chaque classe sont recopiées au niveau des propriétés qui leur sont applicables.

Class

Rid	rtype	name	desc	view_Of	its_superclass	bound1	bound2
C1	'Class'	'C1'				1	2
C2	'Class'	'C2'			C1	1	1
F1	'FuctionalView_Class'	'F1'		C1		3	3

Property

Rid	rtype	name	desc	domain	range	bound1	bound2
P1	'Property'	'P1'		C1	'String'	1	2
P2	'Property'	'P2'		F1	'String'	3	3

FIG. 6.2 – Structure de la partie *Ontologie* d'OntoDB2

1.2.1.2 Structure des ontologies d'OntoDB

Pour la partie ontologie, OntoDB représente à l'identique la structure objet du formalisme d'ontologie dans la base de données. Ainsi, une table est définie pour chaque entité du formalisme d'ontologie avec héritage de tables. En plus, pour la représentation des associations polymorphes, OntoDB défini des tables intermédiaires ou tables d'aiguillage.

Nous remarquons par exemple que les associations du modèle objet du formalisme de la figure 6.1-a, sont traduites dans OntoDB par des tables d'aiguillage comme le montre la figure 6.3. Chaque table d'aiguillage est composée de cinq colonnes :

1. *rid* : de type entier est l'identifiant de la table d'association, utilisé pour l'aiguillage. C'est cette colonne qui est référencée par les tables des entités

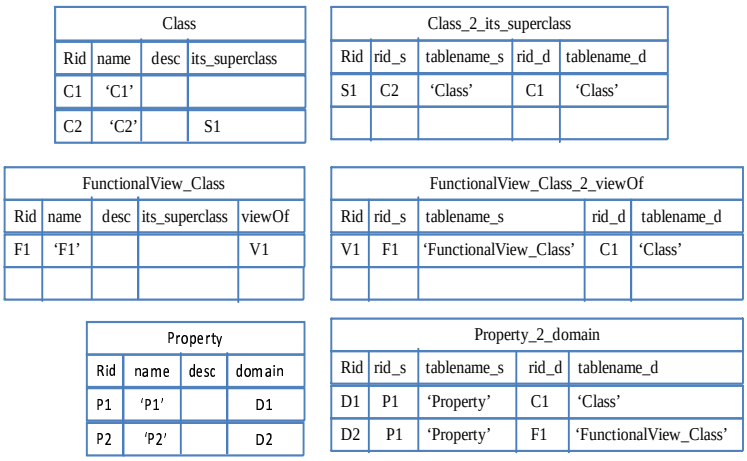


FIG. 6.3 – Structure de la partie *Ontologie* d'OntoDB

2. *rid_ s* : (rid Source) de type entier est le *rid* de l'entité d'origine (dont l'attribut fait référence)
3. *tableName_ s* : (TableName Source) de type chaîne est le nom de la table associée à l'entité dont l'attribut fait référence.
4. *rid_ d* : (rid Destination) de type entier est le *rid* de l'entité de référence.
5. *tableName_ d* : (TableName Destination) de type chaîne est le nom de la table associée à l'entité cible.

Toutes ces tables ont disparue dans OntoDB2 puisque le polymorphisme est traité à l'intérieur de chaque table. Ceci explique la simplification du nombre total de tables.

Avant de présenter les résultats obtenus, nous présentons ci-dessus les ontologies utilisées et les caractéristiques de la machine de test.

1.2.2 Description du banc d'essai

S'agissant des tests d'exploitation d'ontologies PLIB, nous avons pris comme jeux d'essais des ontologies industrielles définies en utilisant ce formalisme. Nous avons effectué nos tests sur trois ontologies de différentes tailles (cf. table 6.1) :

1. l'ontologie normalisée *Roulement* (ISO 23768) composée de 48 classes et 123 propriétés. La moyenne de la profondeur de la hiérarchie des classes est de 3 ;
2. l'ontologie E*cals* définie par JEITA (Japan Electronics and Information Technology industries Association) pour sa base de composants. Cette ontologie est composée de 766 classes et 3086 propriétés. La moyenne de la profondeur de la hiérarchie des classes est de 7 ;
3. l'ontologie normalisée IEC 61360-4 composée de 190 classes et 1026 propriétés. La moyenne de la profondeur de la hiérarchie des classes est de 5.

1.2.3 Machine de test

– Système d'exploitation Windows XP Professionnel

	Ecals	IEC	Roulement
Nombre de classes	766	190	48
Nombre de propriétés	3086	1026	123
Profondeur moyenne de la hiérarchie	7	5	3

TAB. 6.1 – Caractéristique des ontologies PLIB utilisées

- 2 Go de RAM
- 3 Ghz de fréquence de processeur
- 180 Go de disque dur

1.2.4 Résultats obtenus

La figure 6.4 montre les résultats obtenus pour le parcours complet de la hiérarchie simple des classes sur ces différentes bases de données. Nous remarquons que OntoDB2 présente de meilleures performances que OntoDB. Ce résultat est dû au fait que lors de la construction de la structure arborescente, il est nécessaire d'accéder pour chaque nœud à l'ensemble de ses sous-classes directes. Dans OntoDB2 cette requête porte sur la seule table de l'entité *Class* et est réalisée ne manière non récursive car la relation *its_superclass* est indexée. A contrario, pour calculer l'arborescente dans OntoDB, il est nécessaire de réaliser la jointure entre la table *class* et la table d'aiguillage *class_2_its_superclass* pour chaque classe. Le temps de construction de l'arborescence croit donc fortement avec le nombre de classes dans l'ontologie.

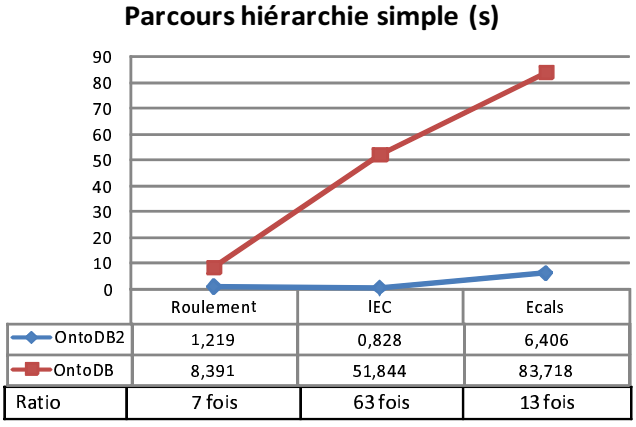
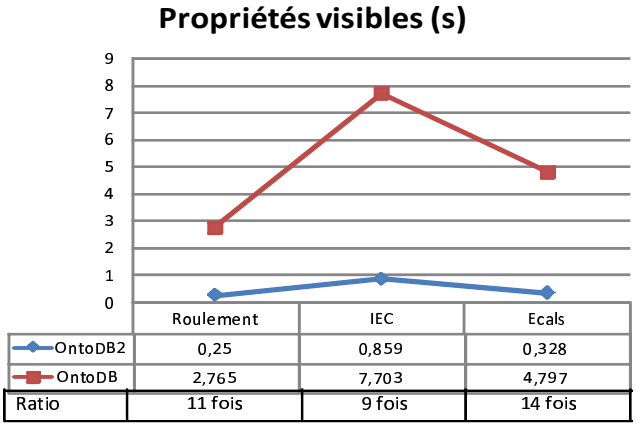


FIG. 6.4 – Temps de construction de la hiérarchie des classes

La figure 6.5 montre les résultats obtenus pour le calcul de l'ensemble des propriétés visibles d'une classe donnée. Dans ce cas également, OntoDB2 présente de meilleurs performances que OntoDB. Ce résultat s'explique par le fait qu'OntoDB2 matérialise dans la table de l'entité *Property*, les valeurs des colonnes d'étiquette du domaine de chaque classe. De ce fait, un seul parcours de la table *property* est nécessaire pour récupérer l'ensemble des propriétés visibles (par transitivité) d'une classe. A la différence d'OntoDB2, OntoDB réalise pour retourner l'ensemble des propriétés définies au niveau d'une classe, deux jointures entre les tables *class*, *property* et la table d'aiguillage *property_2_domain*. Ainsi, pour retourner l'ensemble des propriétés visibles

d'une classe. il répète ce traitement sur toutes les superclasses de cette dernière. L'accès aux superclasses est réalisé de manière récursive.



	Roulement	IEC	Ecals
OntoDB2	0,25	0,859	0,328
OntoDB	2,765	7,703	4,797
Ratio	11 fois	9 fois	14 fois

FIG. 6.5 – Temps d'accès aux propriétés visibles d'une classe

Au total, la nouvelle modélisation d'OntoDB2 est entre 7 fois et 63 fois plus rapide que celle d'OntoDB. Ces évaluations montrent bien que la mise à plat de la structure du niveau ontologique, permet de réduire le temps de calcul de l'arborescence et de parcours des propriétés des classes de l'ontologie. La mise à plat a amélioré les performances des traitements de niveau ontologiques. C'était bien l'objectif cherché et qui a été atteint. Nous présentons ci-dessous les évaluations effectuées sur la partie données.

2 Flexibilité des types de données

Le second objectif d'OntoDB2 était de permettre le support de types de données non standards. Nous avons par exemple dans le cadre du projet e-Wok Hub, étendu les types de données d'OntoDB2 avec les types de données spatiaux. Pour cela, nous avons tout d'abord défini la représentation EXPRESS des types géométriques issus de la spécification OpenGIS. Les entités EXPRESS correspondantes étendent par sous-typage, l'entité *Data_ type* du formalisme noyau d'OntoDB2. Nous avons ensuite grâce aux programmes génériques que nous avons développés en utilisant des techniques d'IDM, généré la structure de représentation en base de données et l'API de niveau ontologique nécessaire à l'instanciation de ces nouveaux types. La méta-représentation de ces types a également été représentée au sein de la partie méta-schéma d'OntoDB2. OntoDB2 étant une architecture de type 3 donc la structure est similaire à celle du MOF, la méta-représentation des types de données permet de modifier aisément ces derniers. La figure 6.6 illustre le processus d'extension du système de type d'OntoDB2 par les types géométriques. Tout d'abord, le système de type OpenGIS est modélisé en EXPRESS dans un schéma que nous appelons OPENGIS_TYPES. Ce schéma utilise le modèle EXPRESS du formalisme d'ontologie (*Peigne*), et ses entités sous-typent l'entité *Data_ type* du Peigne. Ensuite, les modules génériques que nous avons développés sont utilisés pour (1) définir les structures de représentation pour la partie ontologie du schéma de type OpenGIS, (2) représenter ce dernier dans la partie méta-schéma et (3) générer l'API de

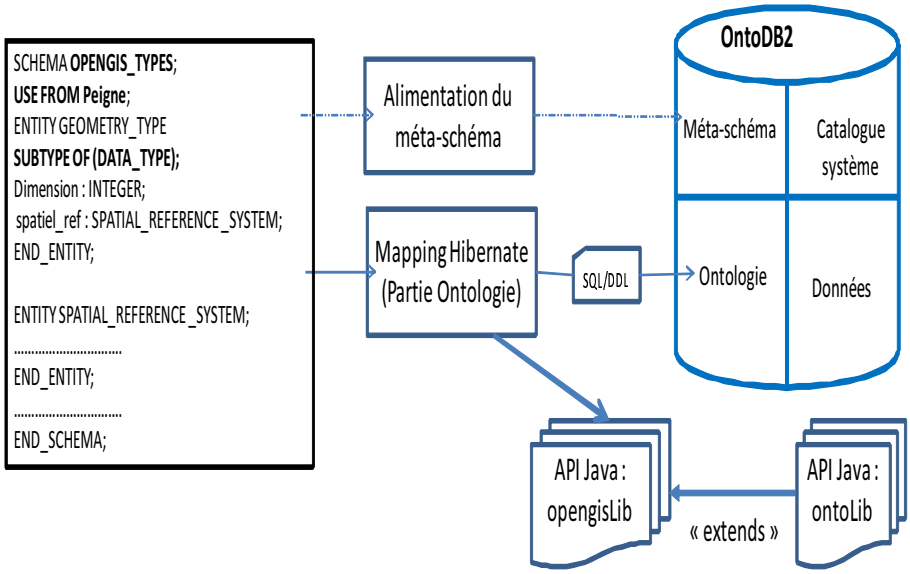


FIG. 6.6 – Processus d'extension du système de types de données

manipulation des types géométriques. Cette API, que nous appelons *opengisLib*, étend l'API *ontoLib* du formalisme d'ontologie. Les propriétés de domaine de valeur un type géométrique peuvent donc être manipulés via cette API et être représentées dans la partie ontologie d'OntoDB2.

Ainsi, les programmes génériques que nous avons développés permettent bien de réaliser facilement l'extension de la bdbo OntoDB2.

Les mécanismes d'indexation des types géométriques que nous avons étudiés au chapitre précédent ont également été intégrés au formalisme d'ontologie afin d'automatiser et de faciliter les interrogations sur les données associées aux types géométriques.

Nous présentons dans le section 2.2 l'évaluation des traitements sur ces données et nous comparons les résultats obtenus entre l'approche par indexation que nous avons choisi et l'approche par saturation sur le système SOR et par raisonnement sur le système d'Oracle. Nous commençons par rappeler succinctement, l'approche de représentation des données adoptée par les systèmes de BDBO OntoDB2, SOR et Oracle.

2.1 Rappel sur le schéma des données d'OntoDB2, SOR et Oracle

Nous rappelons succinctement dans cette section, l'approche de représentation des données des systèmes OntoDB2, SOR et Oracle, ainsi que les solutions adoptées par chacun de ces systèmes pour assurer le raisonnement sur les données. Ces informations vont nous servir par la suite lors de l'interprétation les résultats obtenus.

2.1.1 Schéma des données d'OntoDB2 : approche horizontale

Lors de nos essais sur OntoDB2, nous avons travaillé avec l'approche de représentation horizontale. Dans cette approche, une table est créée pour chaque classe canonique de l'ontologie qui possède une extension. Les propriétés de type collection sont représentées par une table d'association. Un index est crée sur toute colonne représentant l'identifiant d'une table ainsi que sur

toute colonne représentant une clé étrangère.

Rappelons également, qu'une vue est définie pour chaque table de la partie donnée. Dans OntoDB2, trois solutions sont mises à œuvre pour assurer le raisonnement efficace au sein de la BDBO :

1. l'indexation par des techniques d'étiquetage permettant de substituer les raisonnements sur les relations d'ordre par des raisonnements numériques,
2. l'utilisation de l'interpréteur de requête ontologique qui permet par exemple d'accéder aux données des classes non canoniques au travers des vues associées ou encore, d'interpréter les caractéristiques tels que l'inverse pour réaliser l'accès adéquat à la base de données.
3. la saturation des données pour les propriétés transitives et symétriques par exemple.

2.1.2 Schéma des données de SOR

Le système SOR [44] proposé par IBM est une BDBO de type 2 fondée sur le formalisme d'ontologie OWL. Le système SOR définit une table pour chacune des constructions du formalisme d'ontologie OWL. Pour la représentation des instances, SOR utilise une approche verticale avec ID. Les principales tables définies dans SOR et qui sont impliquées dans les requêtes que nous avons utilisées sont :

- la table *individual* permettant de stocker tous les individus des classes de l'ontologie. Cette table comporte deux colonnes : l'identifiant interne et l'*uri* de chaque individu ;
- la table *typeOf* permettant d'associer à chaque individu, la ou les classes auxquelles il appartient. Cette table saturée, comporte une colonne de type booléen qui précise si le type d'un individu a été inféré ou non ;
- la table *relationship* représente la table principale de triples. Cette table permet de stocker pour tous les individus des classes de l'ontologie, les valeurs associées aux différentes propriétés qui le caractérisent. Comme la table *typeOf*, cette table est également saturée et elle comporte une colonne de type booléen qui permet de préciser si la relation a été inférée.

Pour la gestion des raisonnements, SOR utilise une approche par saturation où toutes les inférences (sur les axiomes OWL tels que *subclassOf*, *typeOf*, *subpropertyOf*, *transitive*, etc.) sont réalisées au chargement des données et à chaque nouvelle insertion. Toutes les nouvelles données inférées sont rajoutées dans la base de données. Ainsi, le système SOR, est capable lors de l'interrogation des données, de retourner les résultats en effectuant des requêtes SQL sans inférences supplémentaires.

2.1.3 Schéma des données d'Oracle

Le système proposé par Oracle est une infrastructure pour la gestion des ontologies. Dans les essais que nous avons réalisé, nous avons utilisé cette infrastructure comme une BDBO de type 1 où les ontologies et les données sont représentées dans les mêmes tables dans une approche verticale avec ID comme dans SOR. Oracle [13] réalise à la demande les inférence associées à des bases règles prédéfinies (OWLPrime pour les ontologies OWL), ou à des bases de règles définies par les utilisateurs lors de l'exécution des requêtes.

2.2 Évaluation de l'approche d'indexation

Compte tenu de l'absence de jeux d'essais standard permettant de tester nos propositions sur la gestion des relations d'ordre dans les BDBO, nous avons expérimenté notre approche sur des données réelles fournies par les partenaires du projet e-Wok Hub.

2.2.1 Description du banc d'essai COG

Rappelons que l'ontologie du projet e-Wok Hub (voir figure 5.4, chapitre 5) comporte trois classes : *zone_geographique*, *document* et *segment*. Dans les essais élaborés dans le projet e-Wok Hub, l'arbre de la relation d'ordre *estSubdiviseEn* portant sur les instances de la classe *zone_geographique*, a une profondeur de cinq ; son sommet est un pays, le niveau un est constitué des régions, le niveau deux des départements, le niveau trois des arrondissements, le niveau quatre des cantons et enfin le niveau cinq des communes. La classe *zone_geographique* comporte au total 40938 instances, la classe *document* 8543 instances (en vraie grandeur, elle pourrait comporter quelques centaines de milliers de documents), et la classe *segment* 2181 instances. La propriété *estGeolocalisePar* de la classe *segment* est propagée par la relation d'ordre *seSubdiviseEn*.

2.2.2 Métriques utilisées

Pour valider notre approche, nous avons effectué une comparaison entre OntoDB2 et les BDBO SOR et Oracle qui adoptent un raisonnement par saturation. Nous nous sommes intéressés lors de nos essais à deux aspects :

1. le temps de réponse obtenu à l'exécution des requêtes. Les requêtes utilisées dans cette évaluation sont énumérées dans le tableau 6.2 et sont représentatives de l'approche que nous proposons. Elles permettent les évaluations suivantes :
   (a) Comparer les performances, lors de la substitution, du raisonnement déductif sur les relations d'ordre avec un calcul numérique dans la base de données en utilisant des techniques d'étiquetage. Pour cela, deux requêtes duales R0 et R1, portant sur la propriété *seSubdiviseEn*, ont été définies. La requête R0 réalise le parcours de la relation d'ordre dans le sens direct. La requête R1 est duale de la requête R0, elle réalise le parcours la relation d'ordre dans le sens inverse de la relation *seSubdiviseEn*.
   (b) Comparer l'évaluation de la caractéristique de propagation. Pour cela, nous avons défini la requête R2 portant sur la propriété *estGeolocalisePar* .
2. L'impact des mises à jour. Pour cela, nous avons mesuré le coût lié à l'insertion d'une nouvelle association portant sur la relation d'ordre *seSubdiviseEn*.

TAB. 6.2 – Description des Requêtes

Traitement sur la relation d'ordre
R0 : liste des zones géographiques incluses dans une zone donnée.
R1 : liste des zones géographiques incluant une zone donnée
Traitement sur la relation de propagation
R2 : liste des documents ayant une référence vers une zone donnée.

2.2.3 Expression des requêtes

Nous présentons dans cette section les différentes évaluations que nous avons réalisées. Sur OntoDB2 les requêtes sont exprimées en SQL, sur le système SOR elles sont exprimées en SPARQL, enfin sur le système proposé par Oracle elles sont exprimées en utilisant le langage fournit par celui-ci. Nous ne disposons pas de traducteur automatique entre ces différents langages ; aussi les requêtes ont été traduites manuellement pour chaque système.

2.2.3.1 Expression des requêtes R0, R1 et R2 sur OntoDB2

Les expressions des requêtes R0, R1 et R2 exécutées sur OntoDB2 sont présentées ci-dessous.

- R0 :

```
SELECT zone.URI
FROM zone_geographique zone, zone_geographique x
WHERE (x.URI = 'http ://rdf.insee.fr/geo/REG_11')
AND (zone.seSubdiviseEn_bound1 >= x.seSubdiviseEn_bound1)
AND (zone.seSubdiviseEn_bound2 <= x.seSubdiviseEn_bound2) ;
```

- R1 :

```
SELECT zone.URI
FROM zone_geographique zone, zone_geographique x
WHERE (x.URI = 'http ://rdf.insee.fr/geo/COM_32210')
AND (x.seSubdiviseEn_bound1 <= zone.seSubdiviseEn_bound1)
AND (x.seSubdiviseEn_bound2 >= zone.seSubdiviseEn_bound2) ;
```

- R2 :

```
SELECT doc.URI
FROM document doc, document__contient cont,
     segment__estGeolocalisePar anno, zone_geographique x
WHERE (doc.rid = cont.rid)
AND (cont.contient_ref = anno.rid)
AND (x.URI = 'http ://rdf.insee.fr/geo/PAYS_FR')
AND (anno.seSubdiviseEn_bound1 >= x.seSubdiviseEn_bound1)
AND (anno.seSubdiviseEn_bound2 <= x.seSubdiviseEn_bound2) ;
```

Notons que dans les expressions de requêtes ci-dessus, nous avons remplacé l'appel à la fonction de comparaison *include* par l'expression correspondance (en gras), ceci afin de faciliter l'analyse des résultats effectuée à la section 2.2.4.

2.2.3.2 Expression des requêtes R0, R1 et R2 sur SOR

Les expressions des requêtes R0, R1 et R2 exécutées sur SOR sont présentées ci-dessous.

- R0 :

```
PREFIX geo = <http ://rdf.insee.fr/geo/>
SELECT ?zone
WHERE ( ?zone rdf :type geo :zone_geographique)
      (geo :REG_11 geo :seSubdiviseEn ?zone)
```

– R1 :

```
PREFIX geo = <http ://rdf.insee.fr/geo/>
SELECT ?zone
WHERE ( ?zone rdf :type geo :zone_geographique)
      ( ?zone geo :seSubdiviseEn geo : COM_32210)
```

– R2 :

```
PREFIX geo = <http ://rdf.insee.fr/geo/>
SELECT DISTINCT ?doc
WHERE ( ?doc geo :contient ?seg )
      ( ?doc rdf :type geo :Document)
      ( ?seg rdf :type geo :Segment)
      ( ?seg geo :estGeolocalisePar geo :REG_11)
UNION
      ( ?seg geo :estGeolocalisePar ?zone)
      ( ?zone geo :seSubdiviseEn geo :PAYS_FR)
```

2.2.3.3 Expression des requêtes R0, R1 et R2 sur Oracle

Nous avons défini sur le système Oracle, une règle utilisateur nommée *propagation_rulebase* pour la caractéristique de propagation. Cette base de règle exprime le fait que si *« un segment Seg est géolocalisé par une zone géographique donnée zonegeo »*, alors : *« le segment Seg est également géolocalisé par toute les zones géographiques dont la zone géographique zonegeo est une subdivision »*. Pour la relation d'ordre *seSubdiviseEn*, nous avons utilisée la base de règle prédéfinie OWLP*rime* qui intègre déjà une règle équivalente. Ces deux bases de règles sont appelées lors de l'expression des requêtes R0, R1 et R2 comme présentées ci-dessous.

– R0 :

```
SELECT zone
FROM TABLE(SEM_MATCH('(geo :REG_11 geo :seSubdiviseEn ?zone)',
            SEM_MODELS('COG_md'), SEM_RULEBASES('OWLprime'),
             SEM_ALIASES(SEM_ALIAS('geo', 'http ://rdf.insee.fr/geo/'))), null))
```

– R1 :

```
SELECT zone
FROM TABLE(SEM_MATCH('( ?zone geo :seSubdiviseEn geo :COM_32210)',
            SEM_MODELS('COG_md'), SEM_RULEBASES('OWLprime'),
             SEM_ALIASES(SEM_ALIAS('geo', 'http ://rdf.insee.fr/geo/')), null))
```

– R2 :

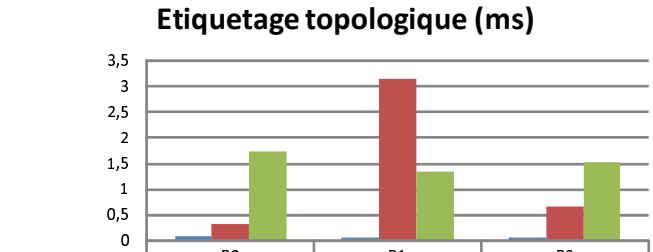
```
SELECT doc
FROM TABLE(SEM_MATCH('( ?doc geo :contient ?seg)( ?seg geo :estGeolocalisePar
geo :PAYS_FR',
        SEM_MODELS('COG_md'), SEM_RULEBASES('propagate_rulebase'),
        SEM_ALIASES(SEM_ALIAS('geo', 'http ://rdf.insee.fr/geo/')), null))
```

Dans les expressions ci-dessus, la primitive SEM_MATCH permet de préciser le pattern de recherche, la primitive SEM_MODEL référence le schéma sur lequel porte la recherche, la primitive SEM_RULEBASES permet de préciser les bases de règles à prendre en compte dans l'évaluation du pattern de recherche et enfin, la primitive SEM_ALIASES permet de définir des alias pour les espaces de noms.

2.2.4 Résultats obtenus

Nous analysons dans cette section, les résultats obtenus pour les requêtes R0, R1 et R2.

2.2.4.1 Temps de réponse



	R0	R1	R2
OntoDB2	0,078	0,047	0,047
SOR	0,313	3,14	0,64
Oracle	1,73	1,34	1,51
Ratio x sor	4 fois	67 fois	17 fois
Ratio x oracle	22 fois	28 fois	32 fois

FIG. 6.7 – Temps de réponse (en s) des requêtes R0, R1 et R2 : Étiquetage topologique

La figure 6.7 montre les résultats obtenus pour les requêtes R0, R1 et R2 sur les systèmes OntoDB2, SOR et Oracle. Nous remarquons que dans tous les cas, (requêtes R0, R1, et R2) OntoDB2 présente dans tous les cas de meilleures performances que SOR et Oracle.

Pour les requêtes R0 et R1 OntoDB2 suit le même plan d'exécution. Ainsi, pour la requête R0 par exemple, OntoDB2 réalise un accès par index pour accéder à l'instance donc l'URI est spécifiée dans la condition de sélection (« http ://www.rdf.insee/fr/geo/REG_11 »). Ensuite, il réalise un second parcours d'index sur la table zone_géographique pour retourner les instances vérifiant le premier membre de la condition d'inclusion *(...>=x.bsup)*. Enfin, une jointure entre les instances de ces deux ensembles suivant le second membre de la condition d'inclusion *(...<=x.binf)* permet d'obtenir les instances correspondantes à la requête R0.

Comme OntoDB2, SOR suit également le même plan d'éxécution pour les requêtes R0 et R1. A la différence d'OntoDB2 qui n'accède qu'a une seule table pour les requêtes R0 et R1 et ne réalise

qu'une seule jointure, SOR réalise au total cinq jointures. Pour la requête R0 par exemple, la jointure entre la table *property* et la table de triples *relationShip* permet de filter les triples ayant pour prédicat la propriété d'URI « http ://www.rdf.insee/fr/geo/seSubdiviseEn ». Une seconde jointure est réalisée avec la table *individual* pour accéder aux URI des instances. Ensuite, ces instances sont filtrées afin de ne retenir que celles dont l'objet correspond à l'individu d'URI « http ://www.rdf.insee/fr/geo/REG_11 ». Enfin, deux autres jointures avec les tables *typeOf* et PrimitiveClass permettent de retourner en résultat final, les instances qui appartiennent à la classe *zone_geographique*. Notons que les tables *relationShip* et *typeOf* étant saturées, les jointures dans SOR portent sur des tables de tailles plus grandes que celles définies dans OntoDB2, ce qui explique les temps de réponse obtenus.

Enfin, pour la requête R2, le temps de réponse obtenu avec OntoDB2, est très inférieur à ceux obtenus avec SOR et Oracle. Ce temps s'explique par le fait que, les labels étant matérialisées dans la table *segment__estGéolocalisePar*, OntoDB2, effectue moins de jointure que SOR. De plus, la table *segment__estGéolocaliséPar* est de taille inférieure aux tables *typeOf* et *relationship* qui sont saturée dans SOR.

Pour chacune des requêtes, Oracle réalise principalement une union entre les données de la table de triplets respectant le pattern de recherche, et de la vue associée à la base de règle spécifiée lors de la requête. On note aussi qu'Oracle effectue des appels récursifs. Ces appels récursifs indiquent que Oracle fait appel à des procédures PL/SQL lors de l'exécution de requêtes, ce qui induit un coût supplémentaire.

Nous venons de voir les performances en terme de temps de réponse de notre proposition de substitution du raisonnement déductif par les requêtes numériques en utilisant une technique d'étiquetage topologique. Cependant, les techniques d'étiquetage topologiques ne sont efficaces que sur les structures d'arbre.

Lorsque les données ne définissent plus une structure arborescente, mais plutôt un DAG, la technique d'étiquetage appropriée est la technique d'étiquetage géométrique par rectangles englobants que nous avons identifié par le nom prédéfini **geo_rectangle**. Comme nous l'avions mentionné précédemment, la démarche à suivre est la même que pour la technique d'étiquetage topologique.
L'expression des requêtes R0, R1 et R2 en utilisant un étiquetage géométrique est donnée ci-dessous.

– R0 :

```
SELECT zone.URI
FROM zone_geographique zone, zone_geographique x
WHERE (x.URI = 'http ://rdf.insee.fr/geo/REG_11')
      AND (zone.seSubdiviseEn_xmin >= x.seSubdiviseEn_xmin)
      AND (zone.seSubdiviseEn_xmax <= x.seSubdiviseEn_xmax)
      AND (zone.seSubdiviseEn_ymin >= x.seSubdiviseEn_ymin)
      AND (zone.seSubdiviseEn_ymax <= x.seSubdiviseEn_ymax) ;
```

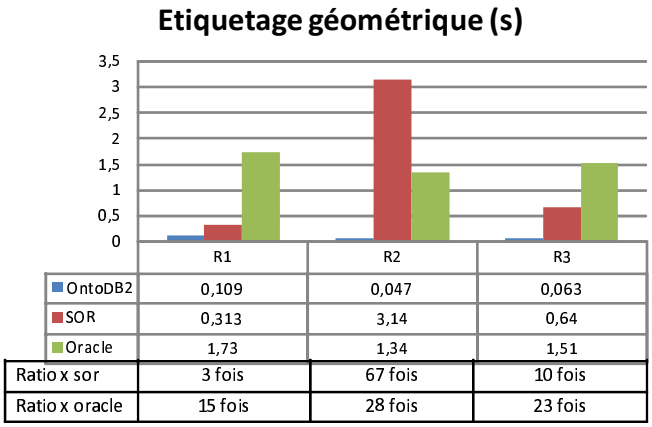
– R1 :

```
SELECT zone.URI
FROM zone_geographique zone, zone_geographique x
WHERE (x.URI = 'http ://rdf.insee.fr/geo/COM_32210')
      AND (zone.seSubdiviseEn_xmin <= x.seSubdiviseEn_xmin)
      AND (zone.seSubdiviseEn_xmax >= x.seSubdiviseEn_xmax)
      AND (zone.seSubdiviseEn_ymin <= x.seSubdiviseEn_ymin)
      AND (zone.seSubdiviseEn_ymax >= x.seSubdiviseEn_ymax) ;
```

– R2 :

```
SELECT doc.URI
FROM document doc, document__contient cont,
      segment__estGeolocalisePar anno, zone_geographique x
WHERE (doc.rid = cont.rid)
      AND (cont.contient_ref = anno.rid)
      AND (x.URI = 'http ://rdf.insee.fr/geo/PAYS_FR')
      AND (anno.seSubdiviseEn_xmin >= x.seSubdiviseEn_xmin)
      AND (anno.seSubdiviseEn_xmax <= x.seSubdiviseEn_xmax)
      AND (anno.seSubdiviseEn_ymin >= x.seSubdiviseEn_ymin)
      AND (anno.seSubdiviseEn_ymax <= x.seSubdiviseEn_ymax) ;
```

Dans les expressions ci-dessus des requêtes R0, R1 et R2, la condition d'inclusion des rectangles englobant des formes géométriques définie par la fonction *mbr_contains* est représentée en gras.



	R1	R2	R3
OntoDB2	0,109	0,047	0,063
SOR	0,313	3,14	0,64
Oracle	1,73	1,34	1,51
Ratio x sor	3 fois	67 fois	10 fois
Ratio x oracle	15 fois	28 fois	23 fois

FIG. 6.8 – Temps de réponse (en s) des requêtes R0, R1 et R2 : Étiquetage géométrique

La figure 6.8 présente les temps de réponses obtenus sur OntoDB2 en utilisant un étiquetage géométrique. Nous notons que dans ce cas également, OntoDB2 présente de meilleures performances que SOR et Oracle. Nous remarquons cependant que les temps réponses obtenus avec les étiquettes géométriques sont pour les requêtes R0 et R2, légèrement supérieurs à ceux obtenus avec les labels topologiques. Cette différence s'explique par le fait que le calcul de la relation d'inclusion d'objets géométriques réalisé avec les rectangles englobant est un calcul approximatif, approché par excès. Les requêtes sont donc susceptibles de retourner dans la plupart des cas un plus grand nombre d'instances qu'initialement escompté.

La table 6.3 montre le ratio du nombre d'instances retournés pour les requêtes R0, R1 et R2 par le traitement avec les étiquettes topologiques (traitement exact) et les étiquettes géométriques (traitement approché), nous observons par exemple que la requête R0 retourne un plus grand nombre d'instances avec la technique d'étiquetage géométrique.

	R0	R1	R2
$\frac{Topologique}{Geometrique}$	$\frac{1513}{1793}$	$\frac{5}{5}$	$\frac{28}{28}$

TAB. 6.3 – Rapport du nombre d'instances entre les étiquetages topologique et géométrique

Au total, les résultats observés montrent que (cf. table 6.4) l'indexation des traitements que nous avons choisi permet d'obtenir avec la BDBO OntoDB2, un ratio de performance (1) de 21 et de 27 fois inférieurs aux temps de réponses observés respectivement sur SOR et Oracle avec la technique d'étiquetage topologique et, (2) de 26 et de 22 fois inférieurs inférieurs aux temps de réponses observés respectivement sur SOR et Oracle avec la technique d'étiquetage géométrique. La technique que nous avons choisi est donc en moyenne 25 fois plus rapide que les approches par saturation et par raisonnement.

	SOR	Oracle
Étiquetage topologique	21 fois	27 fois
Étiquetage géométrique	26 fois	22 fois

TAB. 6.4 – Ratio de performance de l'approche par indexation

2.2.4.2 Mise à jour

En plus des différences observées sur les temps de réponse entre notre proposition et les approches par saturation, un autre aspect important de notre proposition est le coût de mise à jour des données. Les approches de raisonnement par saturation entraînent (1) le calcul de la fermeture transitive à chaque nouvelle insertion afin de matérialiser les nouveaux faits déduits ; (2) une impossibilité de mise à jour des données lorsqu'une instance de relation transitive est supprimée, car il n'est pas possible (dans les systèmes existants) de connaître l'origine d'une relation après fermeture transitive. La saturation doit donc être entièrement ré-effectuée (dans les systèmes SOR et Oracle) pour toute modification nécessitant de supprimer des relations déduites par transitivité. En ce qui concerne l'insertion d'une donnée, la fermeture transitive doit être complétée pour intégrer cette donnée. Ceci est moins coûteux qu'une suppression, mais très coûteux quand même.

Dans les techniques d'étiquetage topologique et géométrique que nous proposons, l'insertion et la mise à jour d'une donnée sont réalisées sans saturation des données. Les opérations de mises à jour sont, de ce fait, plus rapides que dans les approches par saturation.

En effet, l'ajout par exemple de l'instance « http ://rdf.insee.fr.geo/Europe » contenant dans sa subdivision l'instance « http ://rdf.insee.fr.geo/PAYS_FR », se traduit par exemple :

- dans OntoDB2 avec un étiquetage géométrique par deux insertion : (1) l'ajout d'une nouvelle ligne dans la table *zone_geographique* et, (2) l'ajout d'une nouvelle ligne dans la table *zone_géographique__seSubdiviseEn*. A contrario,

– dans le système SOR, cette information entraîne : (1) l'ajout d'une nouvelle ligne dans la table *individual*, (2) l'ajout d'une nouvelle ligne dans la table *typeOf*, (3) le calcul des nouvelles relations possibles par inférence et, (3) l'insertion dans la table *relationShip* des nouvelles relations inférées.

Temps de mise à jour	
OntoDB2	0,014 s
SOR	132 s

TAB. 6.5 – Comparaison des temps de mise à jour d'une relation d'ordre

Quoique les données restant de petite taille, la table 6.5 montre qu'on atteint un ratio de 10000 entre OntoDB2 et SOR lors de l'ajout de cette nouvelle information. Notons également que dans SOR, cela se traduit par l'ajout de 40398 nouvelles relations inférées dans la table *relationShip*.

Enfin, du point de vue occupation mémoire la taille de la table saturée *relationShip* dans SOR croit plus rapidement après chaque nouvelle insertion portant sur le propriété *seSubdiviseEn* que la table *segment_ _ estGeolocaliseePar* dans OntoDB2. Cet avantage de notre approche est d'autant plus important que la profondeur du graphe représentant la relation d'ordre est important.

Les tests que nous avons réalisés montrent que notre proposition présente de bonnes performances en terme de temps de réponse des requêtes. De plus, elle n'entraîne pas de problème de mise à jour des données comme c'est le cas pour les approches par saturation. Après cette évaluation de notre proposition d'indexation de raisonnements déductifs, nous présentons dans la section suivante, les évaluations réalisées pour la validation de notre troisième objectif qui est l'accès aux données canoniques et non canoniques par migration d'instances.

3 Accès efficace aux données canoniques enrichies après migration d'instances

Lorsque une population d'instances ontologiques se réfèrent à une ontologie possédant des classes canoniques et non canonique, les deux méthodes classiques pour permettre de charger et d'interroger ces données consistent soit à saturer toutes les relations, en particulier celles d'appartenance à une classe, ce qui entraîne une redondance des données, soit à raisonner durant les requêtes. Ces deux approches sont contraires à la théorie des bases de données et entraînent dans le premier cas une duplication de données et de grosses difficultés de mise à jour, dans le second cas l'exploitation (interne ou externe) d'un raisonneur. Ces deux approches, en contradiction avec les principes usuels des bases de données, sont incapables de délivrer comme résultat des bases de données classiques (sans redondance et autonomes) dont on connaît la capacité à passer à très grande échelle.

Nous avons, dans cette thèse, proposé une troisième approche qui suppose que les données soient complètes (toute instance appartient à une classe de base explicite, toute propriété est typée) et qui consiste, lors du chargement de la base, à transformer les données représentées comme instances de classe non canonique en instance de classe canonique (migration d'instances), puis à représenter la population des classes non canonique comme des vues sur les classes canoniques.

Pour valider cette approche trois sortes d'évaluations sont nécessaires.

1. Quel est le temps de chargement d'une quantité importante de données à base ontologique d'une part dans OntoDB2 avec migration d'instances, d'autre part dans un système travaillant par saturation pour charger et saturer ces mêmes données ?
2. Peut-on effectivement récupérer, dans une classe canonique, toute la population qui lui appartient, que ses instances soient initialement décrites comme appartenant à cette classe (ou a ses sous-classes) ou que certaines soient décrites comme appartenant à des classes non canoniques définies à partir de cette classe ?
3. Peut-on accéder à toutes les instances d'une classe non canonique par le mécanisme de vue, après migration d'instances lors de son initialisation ?

Ces trois types d'évaluations sont présentés ici. Nous précédons cette présentation de la description du banc d'éssai utilisé.

3.1 Description du banc d'essai

Afin de tester les solutions proposées dans OntoDB2 pour assurer la gestion efficace des données canoniques et non canoniques, nous avons choisi l'ontologie UOB (University Ontology Benchmark) dans sa version OWL Lite.

L'ontologie UOB [46] décrit les classes usuelles dans le domaine universitaire (*université, personne, étudiant, publication, groupes de recherche, assistant, ...*), ainsi que les propriétés qui les caractérisent (*sousOrganisationDe, estInscritA, aPourPublication, estAssistantDe, nom, age, ...*). L'ontologie UOB se décline en deux versions : une version OWL Lite et une version OWL DL qui incluent respectivement toutes les constructions définies dans les deux sous-formalismes d'ontologies OWL Lite et OWL DL. Nous nous somme intéressé ici uniquement à la version OWL Lite car, c'est à ce sous-ensemble du formalisme OWL qu'appartiennent la majorité des constructions supportées par OntoDB2.

Le banc d'essai UOB dispose de trois ensembles de données de une, cinq et dix universités que nous avons utilisé dans nos expérimentations. Le plus grand ensemble de données (de dix universités) comporte deux millions deux cent milles triples de données et, le plus petit ensemble de données comporte deux cent cinquante mille triples de données. Dans la suite, nous utiliserons pour référencer les différentes bases de données, les notations UOB-Lite1, UOB-Lite5 et UOB-Lite10 ; où UOB-Lite*n* correspond à la BDBO avec une ontologie comportant n universités. Notons que les fichiers de données de ces ontologies comportent à la fois des instances de classes canoniques et des instances de classes non canoniques.

Dans les essais que nous avons réalisés avec ce jeu d'essai, nous avons comparé les performances obtenues avec OntoDB2 avec celles obtenues avec le système SOR qui adopte une approche de raisonnement par saturation. Nos évaluations se sont portées sur (1) le temps de chargement des ontologies, (2) la faisabilité de l'approche adoptée et enfin (3) les temps de réponse observés. Nous présentons ci-dessous les différents résultats que nous avons obtenus.

3.2 Temps de chargement des ontologies

Nous avons dans un premier temps, comparé le temps de chargement des différents ensembles de données.

SOR réalise au chargement toutes les inférences possibles aussi bien sur les classes et les propriétés de l'ontologie que sur les instances, puis elle matérialise à la fois les faits explicites et les nouveaux faits déduits. La base de données résultante est donc saturée. Toute nouvelle insertion d'instances nécessite de mettre à jour toute la BDBO en recalculant et insérant dans cette dernière tous les nouvelles relations inférées.

OntoDB2 réalise principalement au chargement :

- le calcul de la hiérarchie de subsumption des classes de l'ontologie,
- le chargement des données canoniques,
- la conversion lorsque cela est possible, des données non canoniques en données canoniques,
- la saturation des caractéristiques symétrique et transitive sur les données ;

Dans l'approche adoptée par OntoDB2, la conversion des données non canonique en données canonique est réalisée *une fois pour toute* au chargement, c'est à dire que le système représente explicitement uniquement des données canoniques. Cette méthodologie assure, à la différence de SOR, que l'insertion par exemple d'une nouvelle instance soit réalisée directement, sans besoin de recalculer les nouvelles relations possibles que cela peut induire.

La figure 6.9 montre le temps de charge en secondes sur les systèmes OntoDB2 et SOR des ontologies UOB-Lite1, UOB-Lite5 et UOB-Lite10. Nous notons que OntoBD2 est en moyenne trois fois plus rapide que SOR.

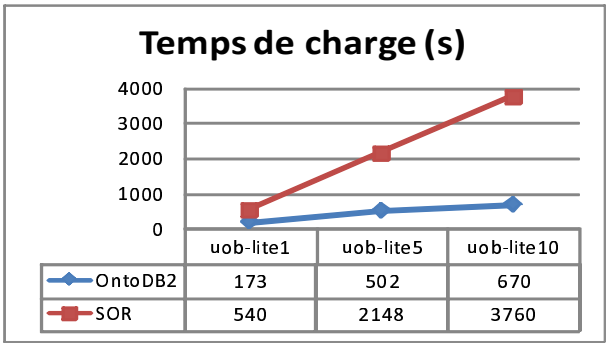


FIG. 6.9 – Comparaison des temps de charge (en s) des ontologies et des données

3.3 Résultat des interrogations

Il convient de remarquer que les instances des ontologies UOB, ne respectent pas toutes les hypothèses du système OntoDB2. En effet, le cadre d'hypothèses (cf section 4 du chapitre 3) que nous avons défini pour OntoDB2 n'est pas entièrement respecté. Il convient donc de modifier les fichiers de données par un raisonneur extérieur pour le rendre complet. Ceci n'a pas été fait de sorte que certaines instances n'ont pas été converties. Nous nous sommes donc intéressés uniquement aux classes dont les instances respectaient ces hypothèses.

3.3.1 Faisabilité de l'approche proposée

Après le chargement des ontologies et des données dans chaque système, nous nous sommes intéressé à validation de l'approche d'accès aux données par migration d'instances adoptée sur

d'OntoDB2. Pour cela, nous avons interrogé des classes permettant de vérifier les différents cas de figures que nous avons énoncés ci-dessus (voir section 3). Nous avons donc défini deux ensembles de requêtes :

1. Dans le premier ensemble, les requêtes Q1 et Q2 interrogent respectivement une classe canonique ayant reçue des instances canoniques, et, une classe canonique ayant reçue des instances converties.

2. Dans le second ensemble de données, les requêtes Q3 et Q4 interrogent des classes non canoniques.

La table 6.6 présente le ratio entre le nombre d'instances retourné par chaque requête et le nombre d'instances attendu. Ce ratio est établi pour l'ensemble de données UOB-Lite1. Ces résultats ont été validé en comparant ces données avec les résultats obtenus avec le système SOR. Les résultats obtenus avec chacune des requêtes correspondent bien à ceux obtenus avec SOR, ce qui nous permet de valider la faisabilité de notre approche : les données converties sont bien accessibles que ce soit pour les classes canoniques, que pour les classes non canoniques.

	Q1	Q2	Q3	Q4
OntoDB2	$\frac{200}{200}$	$\frac{14751}{14751}$	$\frac{640}{640}$	$\frac{13320}{13320}$

TAB. 6.6 – Complétude des requêtes de l'ontologie UOB sur OntoDB2

3.3.2 Temps de réponse des requêtes

Pour chaque évaluation, nous présentons (1) l'expression des requêtes exécutées sur chaque système, (2) les résultats obtenus et enfin, (3) nous analysons ces résultats.

3.3.2.1 Requête sur une classe canonique

Nous considérons par exemple les requêtes Q1 et Q2 suivantes.
Q1 : *« trouver tous les groupes de recherche »*. Ici la classe *researchGroup* est une classe feuille ayant reçue uniquement des données canoniques.
Q2 : *« trouver toutes les personnes »*. Ici, la classe *Person* comporte neuf sous-classes canoniques possédant une extension. C'est une classe canonique ayant reçue à la fois des données canoniques et des données non canoniques converties.
A titre d'exemple, nous donnons ci-dessous l'expression de la requête Q2.

– OntoBD2 :

```
SELECT PostDoc.uri
FROM PostDoc
UNION
SELECT UndergraduateStudent.uri
FROM UndergraduateStudent
. . . . . .
UNION
SELECT FullProfessor.uri
FROM FullProfessor
```

– SOR :

```
PREFIX uob :<http ://uob.iodt.ibm.com/univ-bench-lite.owl#>
SELECT ?x
WHERE ( ?x rdf :type uob :Person) ;
```

La figure 6.10 montre les performances obtenues pour les requêtes Q1 et Q2. On remarque que OntoDB2 présente de meilleures performances que SOR pour la requête Q1, mais est moins rapide pour la requêtes Q2.

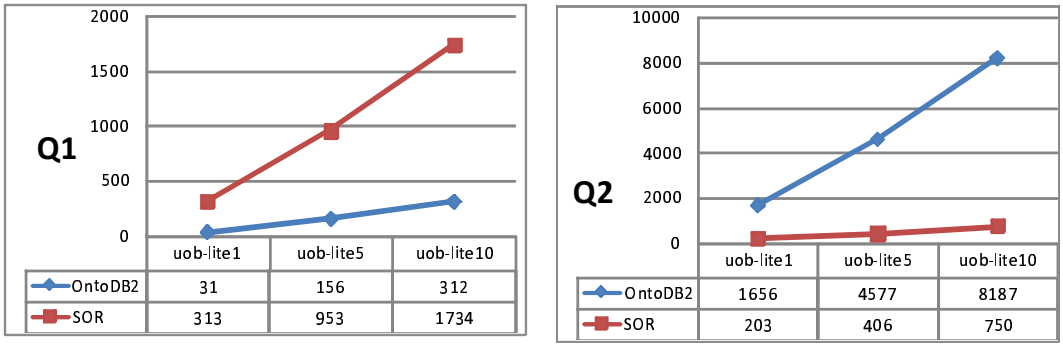


FIG. 6.10 – Temps de réponse (en ms) d'accès aux classes canoniques : requêtes Q1 et Q2

Pour la requête Q1, les résultats obtenus avec OntoDB2 s'explique par le fait que le système accède directement à la table de la classe recherchée (*ResearchGroup*) car cette dernière est une classe canonique feuille. Contrairement à OntoDB2, SOR nécessite de réaliser deux jointures entre les tables *individual*, *typeOf* et *primitiveClass*. Le temps de réponse est donc d'autant plus élevé que la taille des données (le nombre d'instances) dans la base de données augmente. La variation du nombre d'instances a un coût moindre dans OntoDB2 car la seule incidence est dans le temps de parcours de la table alors que dans SOR, les jointures sont réalisées sur des tables beaucoup plus volumineuses.

Sur OntoDB2, une requête sur une classe canonique non feuille s'écrit comme une union de requêtes réalisées sur la classe si celle ci-possède une extension, et sur chacune de ses classes filles canonique qui possède une extension. Le plan d'exécution de chaque sous-requête est le même que celui observé pour la requête Q1.

Sur le système SOR, cette requête est exécutée avec le même plan que la requête Q1. En effet, la table *typeOf* étant saturée, SOR ne réalise pas d'inférence pour déterminer le type des

instances. A contrario, OntoDB2 ne présente pas de bonnes performances pour la requête Q2 qui nécessite de réaliser un grand nombre d'unions.

3.3.2.2 Requête sur une classe non canonique

Nous considérons par exemple les requêtes Q3 et Q4 suivantes.
Q3 : *« trouver tous les professeurs assistants »*. La classe des professeurs assistants (*TeachingAssistant*) est définie ici comme équivalente à l'ensemble des personnes qui sont assistant (*teachingAssistantOf*) d'au moins un enseignement.
Dans ce cas, une seule sous-classe de la classe *Person* utilise la propriété *est assistant de* (*isTeachingAssistantOf*) pour caractériser ses instances. Cette requête portera donc effectivement sur cette seule sous-classe.
Q4 : *« trouver tous les étudiants »*. La classe des étudiants étant définie ici comme équivalente à l'ensemble des personnes qui sont inscrites à une université.
Dans ce cas, un grand nombre de sous-classes de la classe *Person* utilisent la propriété *est inscrit à* (*enrollIn*) pour caractériser leurs instances. A titre d'exemple, l'expression de la requête Q3 sur OntoDB2 et SOR est donnée ci-dessous.

- OntoBD2 : Q3

```
SELECT person.uri
FROM Person, person__isTeachingAssistantOf AS ass
WHERE (person.rid = ass.isTeachingAssistantOf_ref)
```

- SOR : Q3

```
PREFIX uob :<http ://uob.iodt.ibm.com/univ-bench-lite.owl#>
SELECT ?x
WHERE ( ?x rdf :type uob :TeachingAssistant) ;
```

La figure 6.11 montre les performances obtenues pour les requêtes Q3 et Q4.

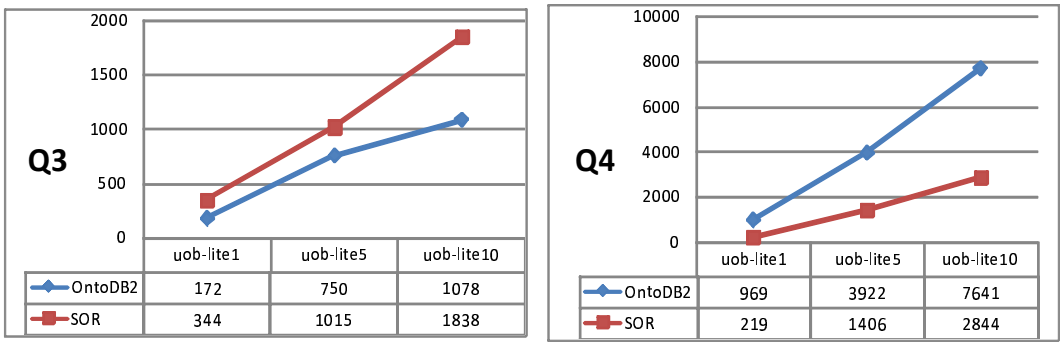


FIG. 6.11 – Temps de réponse (en ms) d'accès aux classes non canoniques : requête Q3 et Q4

La relation de classification étant saturée dans SOR, celui-ci présente le même plan d'exécution avec les requêtes Q3 et Q4 que pour la requête Q1.

Pour la requête Q3, OntoDB2 est plus rapide que SOR ; ceci peut s'expliquer par le fait que OntoDB2 réalise une jointure car une seule sous-classe de *Person* utilise la propriété *teachingAs-*

sistantOf. Par contre, on remarque pour la requête Q4, les temps de réponse sont supérieurs sur OntoDB2. Cela s'explique par le fait que plusieurs sous-classes de *Person* utilise la propriété *enrollIn*, ce qui implique de réaliser des unions.

Les évaluations que nous avons réalisées dans cette section nous ont permis de valider les choix de gestion que nous avons adoptés pour les données. En particulier, ces évaluations montrent que OntoDB2 fournit une bonne qualité de résultat pour les interrogations sur les classes canoniques et les classes définies et ce sans raisonnement ni sans saturation des données. La faisabilité de l'intégration des données par migration d'instances est donc également atteint, mais pour l'instant dans le cadre d'hypothèses assez restrictives.

Nous avons également observés les temps de réponses des requêtes entre OntoDB2 et SOR. Les résultats obtenus montrent que les résultats dépendent essentiellement, pour OntoDB2, de la profondeur de la hiérarchie qui est interrogée et non du fait que l'interrogation porte sur une classe canonique ou non canonique. Si la classe finalement interrogée possède un ensemble significatif de sous-classe, le nombre d'union à réaliser sera plus coûteux que le calcul d'une table saturée.

Notons toutefois que ces résultats doivent être relativisés, car les requêtes réalisées dans notre cas n'accèdent qu'a une seule propriété. Dans la majorité des applications d'ingénierie que nous manipulons, et en particulier dans le domaine technique, les applications accèdent à la description complète des individus c'est à dire à la valeur de l'ensemble (ou d'un grand nombre) de propriétés qui les décrivent. Cela se traduirait, dans SOR, par de nombreuses auto-jointures sur la table principale de triples (saturée) *relationship*. De précédente recherches [20] ont d'ailleurs montré que l'approche de représentation horizontale, que nous avons mise en œuvre, présente de meilleures performances en terme de temps de réponse et de montée en charge que l'approche verticale.

Conclusion

Nous avons dans ce chapitre présenté la validation des différents objectifs fixés pour OntoDB2.

Pour la flexibilité et l'efficience de la représentation ontologique, nous avons montré que les techniques d'IDM utilisées, associées à l'architecture de type MOF de OntoDB2 permettent de prendre en compte de façon simple et de représenter au sein de la partie méta-schéma d'OntoDB2 les différentes évolutions du formalisme d'ontologie. Du point de vue efficience, les schémas de représentation et d'accès que nous avons définis pour l'ontologie dans OntoDB2 ont permis de réduire à la fois la complexité de représentation du formalisme d'ontologie et sa compréhension par les utilisateurs. Enfin, au niveau des traitements de niveau ontologique, nous avons mesuré au moyen d'ontologies PLIB de tailles différentes le temps de navigation dans les concepts de l'ontologie. Ces temps ont montré que la mise à plat de la structure du formalisme d'ontologie a permis non seulement de réduire la complexité du schéma de représentation des ontologies mais aussi d'obtenir un gain de temps très significatif puisque différents parcours de trois ontologies PLIB industrielles sont, en moyenne, réalisés 19 fois plus rapidement avec OntoDB2 qu'avec OntoDB.

Concernant la flexibilité du système de types de données, nous avons monté comment ce dernier peut être, comme pour le formalisme d'ontologie, étendu de façon simple par de nouveaux

types. Nous avons également sur un exemple pratique, montré comment l'extension du système de type de données par des types de données spécifiques d'une application pouvait améliorer l'efficacité de l'application. Ceci a été réalisé avec l'ontologie COG et l'ontologie COG+ associée à l'intégration des types de données géométriques. Nous avons validé, sur ces ontologies, l'approche de raisonnement par indexation que nous avons proposé. Les tests réalisés ont permis de montrer les gains de performance apportés par notre approche dans le traitement des relations d'ordre et de la propagation. Ces tests ont été réalisés sur des données réelles mais encore de taille limitée et, ils ont permis de relever des écarts significatifs entre les temps d'exécution de notre système et les approches usuelles évaluées à la fois sur le système Oracle et sur le système SOR puisque OntoDB2 est, en moyenne 25 fois plus rapide qu'Oracle et 23 fois plus rapide que SOR sur les six requêtes étudiées. Ces écarts devraient probablement encore croître avec la taille des données utilisées, en particulier pour les relations ordonnées.

Enfin, concernant l'objectif d'accès aux données par migration d'instances, nous avons étudié la faisabilité de la conversion des instances non canoniques en instances canoniques. Les tests réalisés dans ce cas ont été menés sur trois sous-ensembles l'ontologie UOB. Nous avons également montré, mais dans le cadre d'hypothèses assez strictes, la faisabilité d'accéder aux instances des classes canoniques enrichies et des classes non canoniques virtuelles par les mécanismes usuels des bases de données. C'est un résultat encourageant, qui ne correspond pas, et de loin, à la totalité d'OWL Lite mais qui est déjà intéressant pour certaines applications, car cela leur permet de décrire les données à la fois en termes de données canoniques et de données non canoniques, et que l'ensemble de celles-ci soient intégrées harmonieusement et interrogeable par des techniques traditionnelles. Nous avons ensuite observé les temps de réponse des requêtes sur OntoDB2 et sur SOR. Les résultats d'OntoDB2 sont environ 2 fois plus lent que ceux de SOR. Ceci est tout à fait acceptable car les requêtes exécutées n'accèdent qu'a une seule propriété, alors que nous savons par des études antérieures [20] que le résultat s'inverse dès que l'on accède à plusieurs propriétés, ce qui est le cas en général dans nos cas d'utilisation. Enfin, les résultats obtenus montrent que les performances observées sur OntoDB2 sont meilleures lorsque les requêtes portent sur des classes feuilles ou des classes ayant un nombre réduit de sous-classes. Ce constat est inversé lorsque les requêtes impliquent des classes possédant de nombreuses sous-classes.

Conclusion et perspectives

Conclusion

Les objectifs de cette thèse étaient de concevoir une architecture de BDBO disposant :

1. d'un formalisme d'ontologie efficient pour PLIB et flexible c'est à dire succeptible d'être étendue avec certaines constructions orthogonales issues d'autres formalismes d'ontologie tels que OWL Lite ;
2. d'un système de types de données flexible, permettant notamment de prendre en compte les types de données non usuels que l'on pouvait rencontrer dans les applications ;
3. d'une approche de gestion des instances canoniques et non canoniques sans saturation ni raisonnement.

Concernant le premier objectif, nous nous sommes basés sur la catégorisation des ontologies selon le modèle en oignon défini au LISI pour :

- définir un noyau constitué des principales constructions canoniques communes aux différents formalismes d'ontologie ;
- définir des mécanismes d'extension génériques de ce noyau par certaines constructions spécifiques et orthogonales des différents autres formalismes d'ontologie.

Des techniques d'IDM ont ensuite été utilisées pour développer des programmes génériques permettant de définir automatiquement les tables de représentation de niveau ontologique et les API d'accès aux ontologies.

En ce qui concerne l'efficience du formalisme d'ontologie, le modèle PLIB initial comportait 217 entités et 118 types définis. Sa représentation en bases de données nécessitait 568 tables. Il n'était donc pas possible de simplifier à la main un tel modèle. Des techniques d'IDM et des règles systématiques de traduction ont été utilisées pour transformer ce modèle en un modèle plus simple, ne possédant en particulier pas de hiérarchie. Ce modèle, appelé *Flatlib* et adapté pour la représentation en base de données, possédait toutefois l'inconvénient de ne pas permettre de contrôler les champs valués et non valués lors de l'accès par programme à la représentation de l'ontologie. Nous avons donc défini, en utilisant également des techniques d'IDM, un autre modèle, à un niveau de hiérarchie et adapté pour la définition des API d'accès par les applications objets. Ce modèle, appelé *Peigne* a également permit de ramener le nombre de tables de 568 à 147 et d'obtenir 117 classes Java au lieu de 217 existant avec le modèle initial. Nous avons ensuite défini, via un programme générique, la correspondance entre le schéma de représentation

en base de données et les schémas d'accès pour les applications. Ce programme se base sur les règles de traduction de la bibliothèque Hibernate.

Grâce à ces transformations, à la fois le modèle de représentation et le modèle d'accès définis sont des modèles simples et facilement compréhensibles par les utilisateurs. Les extensions doivent seulement être spécifiées dans un langage formel pour être générées automatiquement dans la base de données par des programmes génériques développés en utilisant des techniques d'idm. Les extensions pourront donc aisément réaliser des extensions de leurs ontologies pour répondre aux besoins spécifiques des applications qu'ils utilisent.

De plus, les évaluations numériques que nous avons réalisées sur la partie ontologie ont permis d'observer un gain de temps très significatif pour les traitements sur le niveau ontologique, puisque les accès à l'ontologie sont en moyenne 25 fois plus rapide qu'avec le modèle initial.

Concernant le second objectif, les systèmes de BDBO actuels ne permettent pas d'étendre les types de données qu'ils manipulent. Il n'est donc pas possible, dans ces systèmes, de représenter de nouvelles données dont le domaine de valeurs n'a pas été prévu. Nous avons également grâce à l'utilisation des techniques d'IDM et au choix d'une architecture de type 3 similaire au MOF (permettant de représenter le formalisme d'ontologie utilisé dans la base de données sous forme d'instances d'un méta-schéma), de modifier ou étendre le système de type de données, permettant ainsi l'intégration de nouveaux types de données non standards, c'est à dire non prévu dans le formalisme d'ontologie.

Sur l'exemple réel des données géographiques du projet e-Wok Hub, nous avons automatisé, pour le niveau données des BDBO, les techniques d'étiquetage adhoc déjà utilisées pour l'indexation de bases de données XML par exemple. Nous avons également étendu cette approche en implantant non seulement les étiquetages topologiques adaptés lorsque les instances ont la structure d'une forêt, mais aussi les étiquetages géométriques pour les instances qui ont une structure d'un graphe orienté acyclique (DAG). Les résultats numériques obtenus avec cette approche à la fois sur les relations d'ordre et sur les relations propagées, ont montré qu'elle présente des performances largement meilleures que les approches par saturation et par raisonnement.

Nous avons mentionné le fait que ni la solution par saturation, ni la solution par raisonnement ne semblaient satisfaisantes pour la gestion des données non canoniques au sein des BDBO. Concernant le troisième objectif, notre contribution a été de proposer une nouvelle approche de gestion qui consiste à utiliser en plus des capacités des bases de données à assurer la montée en charge, leurs capacités :

- de gestion de la redondance par le langage de requête SQL ;
- de traitement efficace des requêtes numériques et alphanumériques.

La méthodologie que nous avons proposée se base sur quatre hypothèses fondamentales que doivent respecter les ontologies et les données associées pour être intégrées dans OntoDB2 :

1. l'héritage simple pour les classes qu'elles soient canoniques ou non canoniques ;

2. le typage fort des propriétés qui doivent explicitement déclarer un unique domaine et un unique co-domaine.
3. la mono-instanciation des instances ;
4. le typage fort des instances qui doivent chacune définir explicitement son type sous forme d'une classe canonique ou non canonique et n'être décrites que par des propriétés applicables à sa classe.

Cette méthodologie permet alors, dans le cadre de OWL Lite :

1. de convertir les instances de classes non canoniques en instances de classes canoniques, de manière à ne manipuler que des données canoniques dans la BDBO ;
2. d'utiliser le mécanisme de vues des bases de données pour accéder aux instances de classes non canoniques ;

Cette méthodologie permet ainsi (1) de s'affranchir les problèmes de mise à jour rencontrés dans les approches par saturation et (2) d'utiliser les mécanismes usuels des bases de données et non les raisonneurs pour accéder de manière efficace aux instances des classes canoniques complètes (enrichies par les instances définies initialement par leur appartenance à des classes non canoniques), et aux instances de classes non canoniques (virtuelles) par les vues définies sur les classes canoniques.

Une difficulté que nous avons eu pour valider notre approche est que nous n'avons pas trouvé d'ontologie OWL Lite test dont toutes les instances vérifiaient nos hypothèses de typage. Ceci est dû au fait que les ontologies test que nous avons trouvées cherchent essentiellement à vérifier la conformité à toutes les constructions de OWL, ce qui n'était pas notre cas d'études. Nous nous sommes donc limités à l'étude des classes qui les respectaient.

Les résultats observés ont permis de valider la faisabilité de notre approche de gestion des données des classes canoniques et non canoniques par migration d'instances. Le chargement d'instances est d'environ trois fois plus rapide que le chargement dans les systèmes à saturation. Par contre, sur les requêtes expérimentées, notre approche est environ deux fois plus lente que le système SOR. Ce dernier résultat est néanmoins à relativiser très fortement car les requêtes accédaient toutes à une seule propriété, alors que de précédents essais ont montré que dès lors que les requêtes accèdent à plusieurs propriétés (ce qui est le cas usuel de nos applications), l'approche horizontale que nous avons utilisée présente de meilleures performances que les autres approches.

Perspectives

De nombreuses perspectives tant à caractère théorique que pratique peuvent être envisagées. Dans cette section, nous présentons succinctement celles qui nous paraissent être les plus intéressantes.

Extension au support complet du formalisme d'ontologies OWL Lite

Nous nous sommes en particulier intéressé aux formalismes d'ontologie PLIB, RDFS et OWL Lite. Nous avons également défini des hypothèses fondamentales que doivent respecter les ontologies et les données de ces différents formalismes pour être supportées par OntoDB2. Compte

tenu de ces hypothèses, seul un sous-ensemble des ontologies OWL Lite existantes est supporté. Il serait intéressant d'explorer la possibilité de relaxer ces hypothèses pour permettre le support complet d'OWL Lite en réalisant par exemple, un prétraitement par un raisonneur externe. Dans le même ordre de travaux, il serait intéressant de définir, dans le cadre d'OWL2, les profils auxquels peuvent s'appliquer la méthode de migration d'instances que nous avons définie.

Support de l'héritage multiple et de la multi-instanciation

L'architecture actuellement retenue pour les données ne permet ni multi-instanciation, ni héritage multiple. Néanmoins des travaux sont en cours pour implanter la gestion binaire des données qui permettrait d'implanter ces deux mécanismes, si nous en éprouvons le besoin. Il serait intéressant d'étudier de façon plus approfondie comment ces mécanismes pourraient collaborer avec (ou remplacer) les mécanismes d'héritage simple et d'agrégation d'instances de PLIB, puis d'évaluer la faisabilité d'implanter ces mécanismes soit comme compléments soit comme alternatives aux mécanismes PLIB.

Développement et documentation d'un environnement de personnalisation

Un aspect important d'OntoDB2 est qu'il permet de personnaliser son modèle d'ontologie, tant au niveau de la structure de l'ontologie, qu'au niveau des types de données utilisables. Pour rendre ces capacités facilement exploitables, il serait intéressant de définir et de documenter des interfaces Homme-Machine facilitant la définition des extensions souhaitées et automatisant la suite des tâches nécessaires pour valider puis réaliser l'extension.

Intégration des aspects comportementaux au niveau donnée

Dans cette thèse, nous avons proposé des mécanismes d'extension du formalisme d'ontologie et du système de types de données supportés au niveau ontologique. Il serait intéressant d'étendre cette solution de façon à permettre l'intégration automatique au niveau donnée, des comportements que l'on peut associer aux instances. Par exemple pour exprimer qu'une propriété est *dérivée*, c'est à dire qu'elle peut se calculer à partir des autres propriétés, ou pour exprimer les contraintes d'intégrité, locales ou globales, auxquelles doivent se conformer les instances. Ces contraintes d'intégrité vont en effet pouvoir s'exprimer de façon très complète pour la version 2 du formalisme PLIB. Il serait également intéressant d'étudier comment le méta-schéma et le langage de requêtes ontologique peuvent servir de support pour l'intégration dynamique de ces aspects.

Bibliographie

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. 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, 2007.

[2] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *SIGMOD Rec.*, 18(2) :253–262, 1989.

[3] R. Agrawal, A. Somani, and Y. Xu. Storage and Querying of E-Commerce Data. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 149–158, 2001.

[4] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ics-FORTH RDFSuite : Managing voluminous RDF description bases. In *Proceedings of the 2nd International Workshop on the Semantic Web*, pages 1–13, 2001.

[5] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. Owl web ontology language reference. *W3C, http ://www.w3.org/TR/owl-ref/*, 2004.

[6] B.McBride. Jena : Implementing the rdf model and syntax specification. *Proceedings of the 2nd International Workshop on the Semantic Web*, 2001.

[7] A. Borgida. Description logics in data management. *IEEE Transactions on Knowledge and Data Engineering*, 7(5) :671–682, 1995.

[8] A. Borgida and R. J. Brachman. Loading data into description reasoners. *SIGMOD Record*, 22(2) :217–226, 1993.

[9] E. Bozsak, M. Ehrig, S. Handschuh, A. Hotho, A. Maedche, B. Motik, D. Oberle, C. Schmitz, S. Staab, L. Stojanovic, N. Stojanovic, R. Studer, G. Stumme, Y. Sure, J. Tane, R. Volz, and V. Zacharias. 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, 2002. Springer-Verlag.

[10] D. Brickley and R. Guha. Rdf vocabulary description language 1.0 : Rdf schema. *W3C, http ://www.w3.org/TR/rdf-schema/*, 2002.

[11] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame : A generic architecture for storing and querying rdf and rdf schema. In I. Horrocks and J. Hendler, editors, *Proceedings of the 1st International Semantic Web Conference (ISWC'02)*, number 2342 in Lecture Notes in Computer Science, pages 54–68. Springer Verlag, July 2002.

[12] Y. Chen, J. Ou, Y. Jiang, and X. Meng. Hstar - a semantic repository for large scale owl documents. In *First Asian Semantic Web Conference (ASWC'06)*, volume 4185, pages 415–428, 2006.

[13] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *VLDB '05 : Proceedings of the 31st international conference on Very large data bases*, pages 1216–1227. VLDB Endowment, 2005.

[14] V. Christophides, D. Plexousakis, M. Scholl, and S. Tourtounis. On labeling schemes for the semantic web. In *WWW*, pages 544–555, 2003.

[15] D. Connolly, F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Daml+oil reference description. *W3C, http ://www.w3.org/TR/daml+oil-reference*, March 2001.

[16] N. Cullot, C. Parent, S. Spaccapietra, and C. Vangenot. Des SIG aux ontologies géographiques. *Revue internationale de géomatique*, 13(3/2003) :285–306, 2003.

[17] I. H.-D. M. P. P.-S. D. Fensel, F. van Harmelen. Oil : an ontology infrastructure for the semantic web. 16 :38– 45, 2001.

[18] J. de Bruijn, A. Polleres, R. Lara, and D. Fensel. Owl flight. *WSML Delivrable D20.3*

v0.1, http ://www.wsmo.org/TR/d20/d20.3/, august 2004.

[19] H. Dehainsala. Explicitation de la sémantique dans les bases de données : base de données à base ontologique et le modèle ontodb. *Thèse de Doctorat Université de Poitiers - http ://tel.archives-ouvertes.fr/docs/00/15/75/95/PDF/These.pdf*, 2007.

[20] H. Dehainsala, G. Pierra, and L. Bellatreche. Ontodb : An ontology-based database for data intensive applications. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, Lecture Notes in Computer Science, pages 497–508. Springer, 2007.

[21] P. F. Dietz. Maintaining order in a linked list. In *STOC '82 : Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127, New York, NY, USA, 1982. ACM.

[22] C. Fankam, Y. Ait-Ameur, and G. Pierra. Exploitation of ontology languages for both reasoning and persistency purposes : mapping plib, owl and flight ontology models. In J. Filipe, J. Cordeiro, B. Encarnação, and V. Pedrosa, editors, *Third International Conference on Web Information Systems and Technologies(WEBIST'07)*, pages 254–262. INSTICC Press, March 2007.

[23] D. Fensel, S. Decker, M. Erdmann, and R. Studer. Ontobroker : How to make the www intelligent. In *In Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based Systems, Workshop (KAW98*, pages 9–7, 1998.

[24] G.Pierra, E. Sardet, and R. Withier. Modélisation des données : le langage express. Technical report, LISI-ENSMA, Futuroscope, 1995.

[25] S. Grimm and B. Motik. Closed-world reasoning in the semantic web through epistemic operators. In *In CEUR Proceedings of the OWL Experiences and Directions Workshop*, 2005.

[26] T. R. Gruber. *Formal ontology in conceptual analysis and knowledge representation. Chapter : Towards principles for the design of ontologies used for knowledge sharing.* Kluwer Academic Publishers., 1993.

[27] N. Guarino and R. Poli. Formal ontology in conceptual analysis and knowledge representation. *Special issue of the International Journal of Human and Computer Studies*, 43(5/6) :625–640, 1995.

[28] S. Harris and N. Gibbins. 3store : Efficient Bulk RDF Storage. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, pages 1–15, 2003.

[29] A. Harth and S. Decker. Optimized index structures for querying rdf from the web. In *LA-WEB '05 : Proceedings of the Third Latin American Web Congress*, pages 71–80, Washington, DC, USA, 2005. IEEE Computer Society.

[30] I. Horrocks. Using an expressive description logic : Fact or fiction ? *Principles of Knowledge Representation and Reasoning : Proceedings of the Sixth International Conference (KR'98)*, pages 636–647, June 1998.

[31] IS010303.02. Product data representation and exchange - part 2 : Express reference manual. *ISO*, (055), 1994.

[32] ISO 10303-11. Systèmes d'automatisation industrielle et intégration – représentation et échange de données de produits – partie 11 : Méthodes de description : Manuel de référence du langage express. *International Organization for Standardization*, 1994.

[33] ISO-10303-14. Product Data Representation and Exchange - Part 14 :EXPRESS-X Language Reference Manual. Technical report, ISO, 1999.

[34] ISO 10303-21. Systèmes d'automatisation industrielle et intégration – représentation et échange de données de produits – partie 21 : Méthodes de mise en application : Encodage en texte clair des fichiers d'échange. *International Organization for Standardization*, 2002.

[35] ISO-13584-42. Industrial Automation Systems and Integration Parts LIBrary Part 42 : Description methodology : Methodology for Structuring Parts families. Technical report, ISO, 1998.

[36] S. Jean. Ontoql : un langage d'exploitation des bases de données à base ontologique. *Thèse de Doctorat Université de Poitiers - http ://www.lisi.ensma.fr/ftp/pub/documents/thesis/2007-thesis-jean.pdf*, 2007.

[37] S. Jean, G. Pierra, and Y. Ait-Ameur. *Domain Ontologies : a Database-Oriented Analysis*, volume 1 of *Lecture Notes in Business Information Processing*, pages 238–254. Springer Berlin Heidelberg, 2007.

[38] A. Kiryakov, D. Ognyanov, and D. Manov. Owlim : A pragmatic semantic repository for owl. pages 182–192, 2005.

[39] G. Klyne and J. J. Carroll. Resource Description Framework (RDF) : Concepts and Abstract Syntax. *W3C, http ://www.w3.org/TR/rdf-concepts/*, February 2004.

[40] S. D.-K.-D. M. L. Stoimenov, A. Mitrovic. Bridging objects and relations : a mediator for an oo front-end to rdbmss. In *Information and Software Technology*, volume 41, pages 57–66, 1999.

[41] L.Bellatreche, D. N. Xuan, G. Pierra, and H. Dehainsala. Contribution of ontology-based data modeling to automatic integration of electronic catalogues within engineering databases. *Computers in Industry*, pages 711–724, 2006.

[42] C. Li and T. W. Ling. An improved prefix labeling scheme : A binary string approach for dynamic ordered xml. In *Database Systems for Advanced Applications (DASFAA)*, volume Volume 3453, pages 125–137. Springer, 2005.

[43] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB '01 : Proceedings of the 27th International Conference on Very Large Data Bases*, pages 361–370, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[44] J. Lu, L. Ma, L. Zhang, J.-S. Brunner, C. Wang, Y. Pan, and Y. Yu. Sor : a practical system for ontology storage, reasoning and search. In *VLDB '07 : Proceedings of the 33rd international conference on Very large data bases*, pages 1402–1405. VLDB Endowment, 2007.

[45] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar : an RDF Storage and Query System for Enterprise Resource Management. In *Proceedings of the 30th International Conference on Information and Knowledge Management (CIKM'04)*, pages 484–491, 2004.

[46] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu. Towards a complete owl ontology benchmark. In *ESWC*, pages 125–139, 2006.

[47] J. Mei, L. Ma, and Y. Pan. Ontology query answering on databases. In *Proceedings of the 5th International Semantic Web Conference (ISWC'06)*, pages 445–458, 2006.

[48] B. Motik, I. Horrocks, R. Rosati, and U. Sattler. Can OWL and logic programming live together happily ever after ? In *in proceedings of the 2006 International Semantic Web Conference (ISWC'06)*, volume 4273 of *Lecture Notes in Computer Science*, pages 501–514. Springer, 2006.

[49] B. Motik, I. Horrocks, and U. Sattler. Bridging the gap between owl and relational databases. In *WWW '07 : Proceedings of the 16th international conference on World Wide Web*, pages 807–816. ACM, 2007.

[50] C. F.-Z. O. Corby, R. Dieng-Kuntz. Querying the semantic web with the corese search engine. In *Proc. of the 16th European Conference on Artificial Intelligence (ECAI'2004), subconference PAIS'2004*, pages 705–709, Valencia, August 2004.

[51] Z. Pan and J. Heflin. 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, 2003.

[52] E. Pardede, J. W. Rahayu, and D. Taniar. Mapping methods and query for aggregation and association in object-relational database using collection. In *ITCC '04 : Proceedings of the International Conference on Information Technology : Coding and Computing (ITCC'04)*, volume 2, page 539, Washington, DC, USA, 2004. IEEE Computer Society.

[53] M. J. Park, J. H. Lee, C. H. Lee, J. Lin, O. Serres, and C. W. Chung. 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, 2007.

[54] P. F. Patel-Schneider and I. Horrocks. A comparison of two modelling paradigms in the semantic web. In *Proceedings of the Fifteenth International World Wide Web Conference (WWW'06)*, pages 3–12. ACM, 2006.

[55] J. Petrini and T. Risch. SWARD : Semantic Web Abridged Relational Databases. In *Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA'07)*, pages 455–459, 2007.

[56] G. Pierra. Un modèle formel d'ontologie pour l'ingénierie, le commerce électronique et le Web sémantique : Le modèle de dictionnaire

semantique PLIB. *Journées scientifiques Web Sémantique*, oct 2002.

[57] G. Pierra. Context representation in domain ontologies and its use for semantic integration of data. *Journal Of Data Semantics (JODS)*, pages 173–210, 2008.

[58] G. Pierra. Context representation in domain ontologies and its use for semantic integration of data. *Journal Of Data Semantics (JODS)*, 4900 :173–210, 2008.

[59] G. Pierra, H. Dehainsala, Y. Ait-Ameur, and L. Bellatreche. Base de Données à Base Ontologique : principes et mise en œuvre. *Ingénierie des Systèmes d'Information*, 10(2) :91–115, 2005.

[60] G. Pierra, D. Hondjack, N. N. Negue, and M. Bachir. Transposition relationnelle d'un modèle objet par prise en compte des contraintes d'intégrité de niveau instance. In *INFORSID*, pages 455–470, 2005.

[61] D. Price. Standard Data Access Interface. *ISO-CD 10303-22*, 1995.

[62] D. Schenk and P. Wilson. *Information Modelling The EXPRESS Way*. Oxford University Press, 1994.

[63] G. Staub and M. Maier. Ecco tool kit - an environnement for the evaluation of express models and the development of step based it applications. *User Manual*, 1997.

[64] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD '02 : Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215, New York, NY, USA, 2002. ACM.

[65] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of rdf/s stores. In *Proc. of the 4th Int. Semantic Web Conference*, pages 685–701, 2005.

[66] R. M. V. Haarslev. Description of the racer system and its applications. *Intl Workshop on Description Logics (DL-2001)*, August 2001.

[67] R. Volz, S. Staab, and B. Motik. Incrementally Maintaining Materializations of Ontologies Stored in Logic Databases. *Journal of Data Semantics II*, 3360 :1–34, 2005.

[68] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. *HP Laboratories Technical Report HPL-2003-266*, pages 131–150, 2003.

[69] Z. Wu, G. Eadon, S. Das, E. I. Chong, V. Kolovski, M. Annamalai, and J. Srinivasan. Implementing an inference engine for rdfs/owl constructs and user-defined rules in oracle. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008*, pages 1239–1248, Cancun, Mexico, April 2008.

[70] D. N. XUAN. Intégration de base de données hétérogènes par articulation a priori d'ontologies : application aux catalogues de composants industriels. *Thèse de Doctorat Université de Poitiers*, 2006.

Annexe

Le méta-schéma EXPRESS

Cette annexe décrit le méta-schéma du langage EXPRESS que nous avons présenté à la section 1.2 du chapitre 4. Ce dernier est présenté à la fois sous une forme graphique en utilisant la notation EXPRESS-G et sous une forme textuelle en utilisant la représentation source EXPRESS.

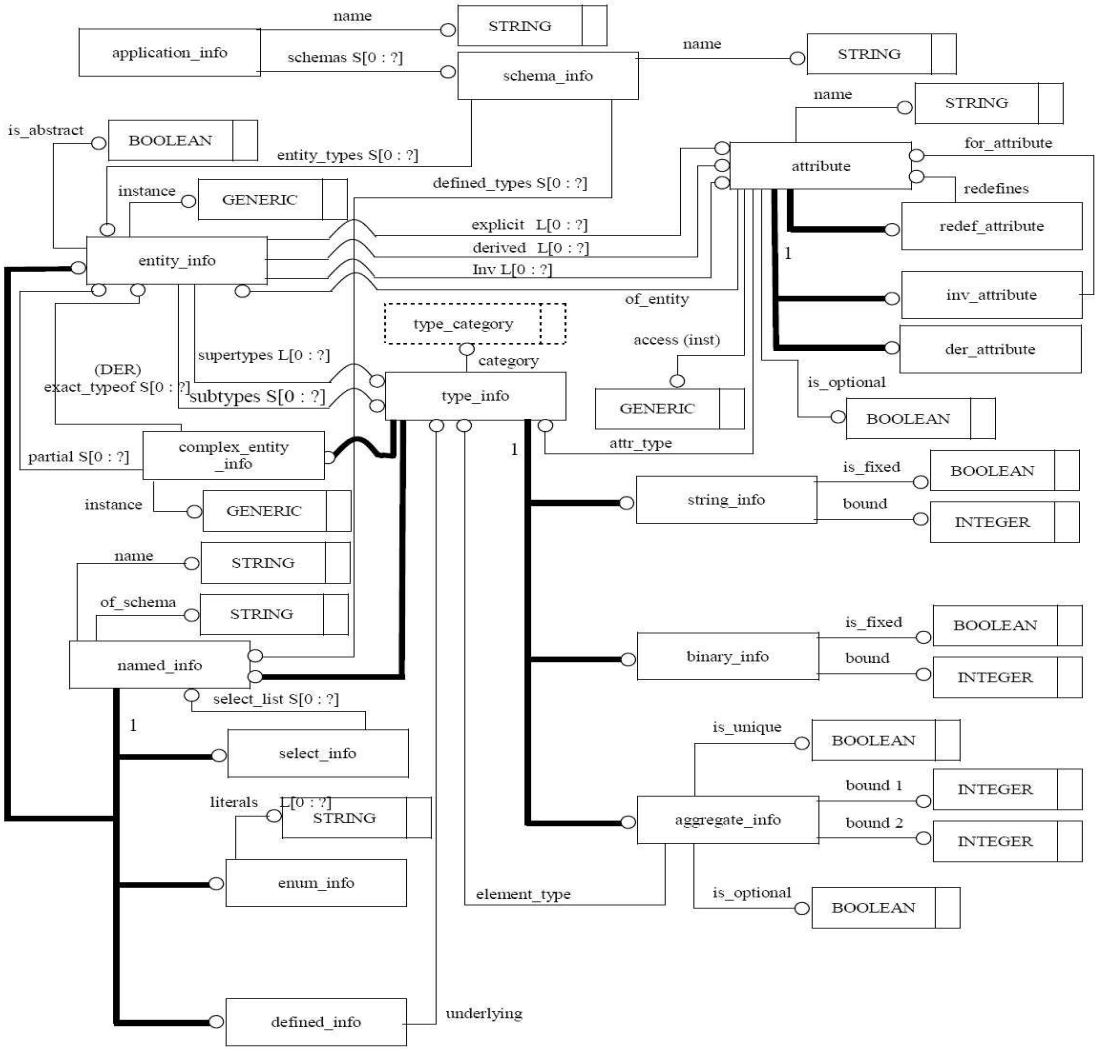


FIG. 1 – Schéma EXPRESS-G du méta-schéma EXPRESS

Représentation source (sous forme textuelle) du méta-schéma EXPRESS

```
SCHEMA meta_schema;

-- basic type categories
TYPE type_category = ENUMERATION OF (
        no_type, number_type, real_type, integer_type, string_type, binary_type,
        logical_type, boolean_type, array_type, list_type, bag_type, set_type,
        aggregate_type, generic_type, enum_type, select_type, entity_type);
END_TYPE;

ENTITY application_info;
OPERATIONS
        name : STRING; -- name of the application
        schemas : SET OF schema_info; -- schemas of the application
END_ENTITY;

ENTITY schema_info;
        name : STRING; -- name of the schema (readonly!)
OPERATIONS
        entity_types : SET OF entity_info; -- entity types of the schema
        defined_types : SET OF named_info; -- defined types of the schema
END_ENTITY;

ENTITY type_info
PRIVATE (obj, auto_extend);
        obj : INTEGER;
DERIVE
        id : INTEGER := obj;
OPERATIONS
        category : type_category; -- the basic category of the type auto_extend;
END_ENTITY;

ENTITY STRING_info
SUBTYPE OF (type_info);
OPERATIONS
        is_fixed : BOOLEAN; -- TRUE if the STRING is FIXED
        bound : INTEGER; -- the bound of the STRING or ? if there is no bound
END_ENTITY;

ENTITY binary_info
SUBTYPE OF (type_info);
OPERATIONS
```

```
        is_fixed : BOOLEAN; – TRUE if the BINARY is FIXED
        bound : INTEGER; – the bound of the BINARY or ? if there is no bound
END_ENTITY;

ENTITY aggregate_info
SUBTYPE OF (type_info);
OPERATIONS
        is_unique : BOOLEAN; – TRUE if the INTEGER has an UNIQUE qualifier
                    – (ARRAY/LIST only)
        is_optional : BOOLEAN; – TRUE if the INTEGER has an OPTIONAL qualifier
                    – (ARRAY only)
        bound_1 : INTEGER; – the lower bound of the INTEGER
        bound_2 : INTEGER; – the upper bound of the INTEGER
        element_type : type_info; – the element type of the INTEGER
END_ENTITY;

ENTITY named_info
SUBTYPE OF (type_info);
OPERATIONS
        name : STRING; – the name of the type (in uppercase letters)
        of_schema : STRING; – the name of the schema (in uppercase letters)
END_ENTITY;

ENTITY defined_info
SUBTYPE OF (named_info);
OPERATIONS
        underlying : type_info; – the type on which the defined type is based
END_ENTITY;

ENTITY select_info
SUBTYPE OF (named_info);
OPERATIONS
        select_list : SET OF named_info; – the types in the SELECT list
END_ENTITY;

ENTITY enum_info
SUBTYPE OF (named_info);
OPERATIONS
        literals : LIST OF UNIQUE STRING; – the ENUMERATION literals (in uppercase letters)
END_ENTITY;

ENTITY entity_info
SUBTYPE OF (named_info)
```

```
PRIVATE (attributes);
OPERATIONS
        is_abstract : BOOLEAN; – return TRUE if the entity is an abstract supertype
        attributes(kind : INTEGER) : LIST OF UNIQUE attribute;
        supertypes : LIST OF UNIQUE type_info; – the supertypes of the type
        subtypes : SET OF  type_info; – restriction : returns only the subtypes
                        – which are declared in the same schema
        explicit : LIST OF UNIQUE attribute; – the explicit attributes of the type
        derived : LIST OF UNIQUE der_attribute; – the derived attributes of the type
        inv : LIST OF UNIQUE inv_attribute; – the inverse attributes of the type
        instance : GENERIC; – create an instance of this type
END_ENTITY;

ENTITY complex_entity_info
SUBTYPE OF (type_info);
        partial : SET OF entity_info;
DERIVE
        exact_typeof : SET OF entity_info := – the most special subtypes
                        QUERY(p <* partial | QUERY(q <* partial |
                        (q :<> : p) AND NOT VALUE_IN(q.subtypes, p)) = []);
OPERATIONS
        instance : GENERIC; – create an instance of that type
END_ENTITY;

ENTITY attribute
SUBTYPE OF (ONEOF(der_attribute, inv_attribute) ANDOR redef_attribute)
PRIVATE (obj, idx, auto_extend);
        obj : INTEGER;
        idx : INTEGER;
OPERATIONS
        name : STRING; – the name of the attribute (in lowercase letters)
        is_optional : BOOLEAN; – TRUE if the attribute is OPTIONAL
        attr_type : type_info; – the attribute type
        of_entity : entity_info;– the entity of which the attribute is part of
        access(inst : GENERIC) : GENERIC; – returns the value of an attribute
                        – access applied on instance inst
        assign(inst, val : GENERIC); – assign the value val to the attribute
                        – of instance inst auto_extend;
END_ENTITY;


    ENTITY der_attribute
SUBTYPE OF (attribute);
END_ENTITY;
```

```
ENTITY redef_attribute
SUBTYPE OF (attribute);
OPERATIONS
        redefines : attribute; – the attribute which is redefined
END_ENTITY;

ENTITY inv_attribute
SUBTYPE OF (attribute);
OPERATIONS
        for_attribute : attribute; – the explicit attribute which is inverted
END_ENTITY;
END_SCHEMA;
```

Liste des tableaux

Table des figures

Glossaire

API	Application Programming Interface
BD	Base de Données
BDBO	Base de Données à Base Ontologique
BSU	Basic Sémantique Unit
CSV	Comma Separated Value
DBO	Données à Base Ontologique
DTD	Document Type Definition
EJB	Enterprise JavaBeans
HQL	Hibernate Query Language
IDM	Ingénierie Dirigée par les Modèles
IEC	International Electrotechnical Commission
ISO	International Organization of Standardization
JDBC	Java DataBase Connectivity
MDA	Model Driven Architecture
MOF	Meta-Object Facility
MCV	Modèle Vue Contrôleur
OL	Ontologie Linguistique
OC	Ontologie Conceptuelle
OCNC	Ontologie Conceptuelle Non Canonique
OMT	Object Modeling Technique
OWL	Web Ontology Language
PLIB	Parts Library (PLIB) - Norme ISO 13584
RDF	Resource Description Framework
RDFS	RDF Schema
SDAI	Standard Data Access Interface - Part 22 (ISO 10303-22 :1998)
SGBD	Système de Gestion de Bases de Données
STEP	Standard for the Exchange of Product Model Data Norme ISO 10303
SQL	Structured query languag
UML	Unified Modeling Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language

OntoDB2 : un système flexible et efficient de Base de Données à Base Ontologique pour le Web sémantique et les données techniques

Présentée par :

Chimène FANKAM

Directeurs de thèse:

Guy PIERRA et Ladjel BELLATRECHE

Résumé. Le besoin d'expliciter la sémantique des données dans différents domaines scientifiques (biologie, médecine, géographie, ingénierie, etc.) s'est traduit par la définition de données faisant référence à des ontologies, encore appelées données à base ontologique. Avec la multiplication des ontologies de domaine, et le volume important de données à manipuler, est apparu le besoin de systèmes susceptibles de gérer des données à base ontologique de grande taille. De tels systèmes sont appelés des systèmes de gestion de Bases de Données à Base Ontologique (BDBO).

Les principales limitations des systèmes de gestion de BDBO existants sont (1) leur rigidité, due à la prise en compte des constructions d'un unique formalisme d'expression d'ontologies, (2) l'absence de support pour les données non standard (spatiales, temporelles, etc.) et, (3) leur manque d'efficacité pour gérer efficacement les données de grande taille. Nous proposons dans cette thèse un nouveau système de gestion de BDBO permettant (1) de supporter des ontologies basées sur différents formalismes d'ontologies, (2) l'extension de son formalisme d'ontologie pour répondre aux besoins spécifiques des applications, et (3) une gestion originale des données facilitant le passage à grande échelle.

Le système que nous proposons dans cette thèse, ontodb2, se fonde sur l'existence d'un ensemble de constructions communes aux différents formalismes d'expression d'ontologies, susceptible de constituer une ontologie noyau, et sur les techniques de gestion des modèles pour permettre l'extension flexible de ce noyau. Nous proposons également une approche originale de gestion des données à base ontologique. Cette approche part du fait que les données à base ontologique peuvent se classifier en données canoniques (instances de classes primitives) et non-canoniques (instances de classes définies). Les instances de classes définies peuvent, sous certaines hypothèses, s'exprimer en termes d'instances de classes primitives. Nous proposons donc de ne représenter que les données canoniques, en transformant sous certaines conditions, toute donnée non-canonique en donnée canonique. Enfin, nous proposons d'exploiter l'interpréteur de requêtes ontologiques pour permettre (1) l'accès aux données non-canoniques ainsi transformées et, (2) d'indexer et pré-calculer les raisonnements en se basant sur les mécanismes du SGBD support. L'ensemble de ces propositions est validé (1) à travers une implémentation sur le SGBD PostgreSQL basée sur les formalismes d'ontologies PLIB, RDFS et OWL Lite, (2) des tests de performances sur des ensembles de données issus de la géographie et du Web.

Mots Clés. Base de données, ontologie, formalisme d'ontologies, bases de données à base ontologique, méta-modélisation, ingénierie dirigée par les modèles, ingénierie des données, interrogation de données, raisonnement, PLIB, OWL.