Explicitation de la sémantique dans lesbases de données : Base de données à base ontologique et le modèle OntoDB

Hondjack Dehainsala

Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Ecole Doctorale des Sciences Pour l'Ingénieur ▪ Université de Poitiers

THESE

pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ DE POITIERS

(Faculté des Sciences Fondamentales et Appliquées)
(Diplôme National — Arrêté du 25 Avril 2002)

Spécialité : INFORMATIQUE

Présentée et soutenue publiquement par

Hondjack DEHAINSALA

Explicitation de la sémantique dans les bases de données :
Base de données à base ontologique et le modèle OntoDB

Directeurs de Thèse : ***Guy PIERRA, Ladjel BELLATRECHE***

Devant la commission d'examen

JURY

Rapporteurs :	Jean CHARLET	Chercheur HDR, INSERM, Université de Paris 6
	Mohand-Said HACID	Professeur, Université Claude Bernard Lyon 1
Examinateurs	Ladjel BELLATRECHE	Maître de Conférences, Université de Poitiers
	Mokrane BOUZEGHOUB	Professeur, Université de Versailles
	Jean-Yves LAFAYE	Professeur, Université de La Rochelle
	Guy PIERRA	Professeur, ENSMA, Futuroscope
	Michel SCHNEIDER	Professeur, Université Blaise Pascal, Clermont Ferrand

Remerciements

Il m'est particulièrement agréable de pouvoir remercier ici le Professeur Guy PIERRA, Directeur de LISI et directeur de cette thèse, pour avoir bien voulu m'accueillir dans le laboratoire, dans son équipe de recherche et m'avoir encadré durant ces quatre années. Je lui exprime ma profonde gratitude. Ses encouragements me furent d'un constant appui sans lequel la persévérance m'aurait manqué.

Je tiens également à remercier Ladjel BELLATRECHE co-direction de cette thèse, pour son encadrement et surtout pour l'aide précieuse qu'il m'a apportée notamment dans la validation de mes travaux et leurs publications.

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

Je remercie Monsieur Jean CHARLET et le professeur Mohand-Said HACID qui ont rapporté sur ma thèse. Je remercie également les professeurs Mokrane BOUZEGHOUB, Jean-Yves LAFAYE et Michel SCHNEIDER qui ont bien voulu accepter de juger mon travail.

Je remercie particulièrement Alphonse DEHAINSALA, mon père, qui a su me convaincre de faire une thèse et du soutient moral qu'il m'a apporté. Je remercie aussi mon frère Patience DEHAINSALA pour la lecture de cette thèse, les remarques et suggestions faites pendant la rédaction de celle-ci.

J'adresse mes remerciements à tout le personnel du LISI, et plus particulièrement, Dung, Loé, Stéphane, Ahmed, Idir, Dago, Manu, Éric, Guillaume, JCP, Frédéric C., Claudine, Hieu, Frédéric R., Nicolas, Chimène, Karim, Sybille, Malo, Youcef, Michaël, Kamel, Dilek, Otzgu et Frédéric D. pour leur présence et leur soutien cordial.

Je suis reconnaissant de l'intérêt fraternel et amical que m'ont témoigné mes ami(e)s, particulièrement à Sylviane, Hung, Patrice, Cédric, Antoine, Sylvain et Anne-lise et son mari Stéphane, Olivier, Yvonne pour leur amitié pour tous les bons moment que nous avons passé ensemble.

Enfin, je remercie la communauté tchadienne, particulièrement Gnomonga KALANDI, à la famille YANAMADJI, Alano, Freddy et sa femme Flora, Ngueto, Tededjim, Talla, Albert, BV, Fabienne, Georges et Oumie pour leur soutien moral et pour tous les bons moments passés à leur compagnie.

A tous les membres de ma famille.
Papa,
Maman,
Patience,
Maïré,
Haiengda,
Maïpa,
Douswé,
et toute la grande famille Waleyam

Table des matières

Table des figures

Résumé

Une ontologie de domaine est une représentation de la sémantique des concepts d'un domaine en termes de classes et de propriétés, ainsi que des relations qui les lient. Avec le développement de modèles d'ontologies stables dans différents domaines, OWL dans le domaine du Web sémantique, PLIB dans le domaine technique, de plus en plus de données (ou de métadonnées) sont décrites par référence à ces ontologies. La taille croissante de telles données rend nécessaire de les gérer au sein de bases de données originales, que nous appelons *bases de données à base ontologique* (BDBO), et qui possèdent la particularité de représenter, outre les données, les ontologies qui en définissent le sens. Plusieurs architectures de BDBO ont ainsi été proposées au cours des dernières années. Les schémas qu'elles utilisent pour la représentation des données sont soit constitués d'une unique table de triplets de type (sujet, prédicat, objet), soit éclatés en des tables unaires et binaires respectivement pour chaque classe et pour chaque propriété. Si de telles représentations permettent une grande flexibilité dans la structure des données représentées, elles ne sont ni susceptibles de passer à grande échelle lorsque chaque instance est décrite par un nombre significatif de propriétés, ni adaptée à la structure des bases de données usuelles, fondée sur les relations n-aires. C'est ce double inconvénient que vise à résoudre le modèle OntoDB. En introduisant des hypothèses de typages qui semblent acceptables dans beaucoup de domaine d'application, nous proposons une architecture de BDBO constituée de quatre parties : les deux premières parties correspondent à la structure usuelle des bases de données : *données* reposant sur un schéma logique de données, et *méta-base* décrivant l'ensemble de la structure de tables. Les deux autres parties, originales, représentent respectivement les ontologies, et le méta-modèle d'ontologie au sein d'un méta-schéma réflexif. Des mécanismes d'abstraction et de nomination permettent respectivement d'associer à chaque donnée le concept ontologique qui en définit le sens, et d'accéder aux données à partir des concepts, sans se préoccuper de la représentation des données. Cette architecture permet à la fois de gérer de façon efficace des données de grande taille définies par référence à des ontologies (données à base ontologique), mais aussi d'indexer des bases de données usuelles au niveau connaissance en leur adjoignant les deux parties : *ontologie* et *méta-schéma*. Le modèle d'architecture que nous proposons a été validé par le développement d'un prototype opérationnel implanté sur le système PostgreSQL avec le modèle d'ontologie PLIB. Nous présentons également une évaluation comparative de nos propositions aux modèles présentés antérieurement.

Introduction générale

Contexte

Cette thèse se situe au confluent de deux domaines de l'informatique : les bases de données et la représentation de connaissances.

Basées, depuis les années 70, sur la dualité entité/association (E/A) pour la modélisation conceptuelle et modèle relationnel pour la gestion des données persistantes, la conception des bases de données a été profondément remise en cause dans les années 90, d'abord par l'émergence de l'objet au niveau conceptuel puis par son apparition au niveau physique avec les bases de données orientées objets. Le reflux de celles-ci, qui semblent désormais surtout utilisées dans des secteurs de niche, amène à une situation où des modèles conceptuels, le plus souvent objets, sont utilisés pour représenter des connaissances matérialisées sous forme de données essentiellement relationnelles, malgré l'intégration progressive et souvent disparate de quelques concepts objets dans les SGBDs dits relationnels-objets. Cette situation entraîne deux difficultés. D'une part, la distance entre le modèle conceptuel objet et le modèle relationnel, qui est seul représenté dans la base de données impose, pratiquement de recoder le modèle conceptuel à l'extérieur de la base de données, que ce soit dans les applicatifs, ou dans des couches logicielles intermédiaires. Ceci est à la fois coûteux et difficile à maintenir. D'autre part, l'intégration sémantique de bases de données reste toujours un sujet aussi difficile. Si deux bases de données sont développées par deux équipes différentes pour représenter essentiellement la même connaissance, il reste impossible d'identifier aisément et en particulier automatiquement, celles des informations représentées qui sont communes aux deux bases, et celles qui sont différentes.

Au cours de la même période, les méthodes de représentation de connaissances faisaient également l'objet d'une évolution importante. Essentiellement motivées par des objectifs de résolution de problèmes ou d'inférences de nouveaux faits à partir de faits connus, ces méthodes privilégiaient la représentation de la connaissance procédurale, qu'elle s'exprime en termes de fonctions (LISP, KIF) ou de règles logiques (Prolog, systèmes experts). Les bases de données jouaient alors un rôle secondaire car l'intérêt essentiel portant sur des aspects dynamiques qui devaient se dérouler en mémoire centrale. Les années 90 ont cependant mis en évidence l'importance de la modélisation explicite du domaine sur lequel portait la connaissance représentée, ainsi que l'intérêt de modéliser les aspects structurels et descriptifs de celui-ci à travers des modèles consensuels appelés ontologies. Bien qu'utilisant des technologies différentes de celles-ci utilisées dans la communauté des bases de données, ces modèles étaient des modèles objets. Les objets du domaine,

souvent appelés des faits, étaient caractérisés par leur appartenance à une (ou plusieurs) classe(s) et par des valeurs de propriétés. Les traitements s'effectuaient toujours, pour l'essentiel, en mémoire centrale, avec des "raisonneurs" tels que RACER, FaCT. Au début des années 2000, en particulier avec l'émergence des travaux sur le Web Sémantique, les objets traités sont devenus de plus en plus nombreux. Différents projets ont donc été lancés dans la communauté de représentation de connaissance pour ranger à la fois ces objets et les ontologies qui en définissaient le sens au sein de bases de données. Nous appelons ces bases de données, des bases de données à base ontologique (BDBO). Une caractéristique singulière des BDBOs développées dans tous ces projets, et qui les rend difficilement utilisables dans la communauté bases de données, est qu'il n'y est fait aucune hypothèse de typage. Toute instance est porteuse de sa propre structure : elle peut être associée à des propriétés diverses et peut même appartenir à des ensembles quelconque de classes. Le schéma de telles bases de données ne peut donc être qu'un schéma décomposé dans lequel chaque instance est éclatée en une multitude d'enregistrements : un pour chaque classe à laquelle il appartient et un pour chacune de ses propriétés. Cette approche laisse néanmoins un problème ouvert : *comment faire lorsque, simultanément, le domaine à représenter contient un nombre significatif d'instances) et que chaque instance est décrite par un nombre significatif de propriétés ?* Toute requête sur un schéma ainsi décomposé demande alors un tel nombre de jointures que le temps de traitements devient considérable.

C'est dans ce double contexte que s'inscrit notre travail. Il vise à concevoir un modèle de base de données à base ontologique qui permet à la fois :

- du point de vue des bases de données traditionnelles, de représenter au sein de la base de données, outre les données et la méta-base, d'une part l'ontologie qui définit le sens des données et, d'autre part, le modèle conceptuel qui en définit la structure ;
- du point de vue de la représentation de connaissance, de représenter, au sein de BDBOs efficaces, des domaines de connaissances comportant un grand nombre de faits décrits par un nombre significatif de propriétés.

Le modèle que nous proposons, appelé *OntoDB*, répond effectivement à ces objectifs.

Du point de vue bases de données, deux éléments sont à souligner. D'abord *OntoDB* permet d'offrir un accès aux données directement au niveau du modèle conceptuel, quelle que soit la distance entre celui-ci et la réalisation relationnelle. Par ailleurs, en représentant également au sein de la BDBO les relations existant éventuellement entre son ontologie locale et une ontologie partagée supposée existante, le modèle OntoDB permet, comme cela a été montré par ailleurs, l'intégration complètement automatique de bases de données à base ontologique ayant une hétérogénéité et une autonomie importantes.

Du point de vue de la représentation de connaissances, la représentation des données de grande taille constitue un réel défi. C'est le cas dans le domaine du Web Sémantique. C'est aussi le cas dans notre domaine d'application cible qui porte sur la modélisation des catalogues de composants industriels et leur intégration au sein des bases de données techniques des grands donneurs d'ordres de l'industrie manufacturière. Dans un tel domaine, des ontologies normalisées

existent et l'enjeu effectif est d'être capable de gérer efficacement des univers qui comportent de l'ordre d'un million de composants, chacun décrit par quelques dizaines de propriétés. Les résultats présentés au chapitre cinq sur un exemple réaliste basé sur une ontologie réelle normalisée montrent qu'effectivement le modèle d'architecture OntoDB supporte ce passage à l'échelle et ce, beaucoup mieux que toutes les BDBOs basées sur un schéma éclaté.

Organisation

La thèse s'organise en trois parties :

1. l'état de l'art,
2. les contributions,
3. l'évaluation de performance.

La première partie est composée de deux chapitres.

Dans le chapitre 1, nous présentons brièvement les approches usuelles de conception de bases de données ainsi que les principaux systèmes de représentation de données persistantes. Nous mettons en évidence des limites des approches classiques de conception de base de données tant du point de vue de la distance entre modèle conceptuel et modèle logique que de celui de l'hétérogénéité des bases de données qui rend difficile l'intégration ultérieure de différentes sources de données. Nous illustrons cet état de l'art par la présentation de quelques outils tels que le langage de spécification EXPRESS et l'application de transformation Hibernate, que nous utiliserons tout au long de cette thèse.

Dans le chapitre 2, nous faisons un état de l'art sur la notion d'ontologie. Nous y présentons, d'abord les principaux composants, leurs caractéristiques, leurs différences avec les modèles conceptuels ainsi que quelques langages de définition d'ontologies proposés dans la littérature (RDF, RDF Schéma, OWL, PLIB). L'objectif visé ici est d'identifier les caractéristiques communes et particulières des ontologies qui devront être représentées dans les bases de données. Nous présentons ensuite les architectures des bases de données à base ontologique existantes. Nous discutons les différentes approches utilisées pour la représentation des ontologies et des données à base ontologique ainsi que l'architecture, les caractéristiques et les fonctionnalités des principaux systèmes de gestion de BDBOs proposés jusque là.

La deuxième partie présente le modèle d'architecture que nous proposons pour les bases de données à base ontologique ainsi que la validation de cette proposition à travers un prototype.

Le chapitre 3 est consacré au modèle d'architecture de bases de données à base ontologique que nous avons proposé dans le cadre notre thèse : le modèle *OntoDB* (**Onto**logie **D**ata**B**ase). Nous présentons, dans un premier temps les objectifs et les fonctionnalités que nous nous sommes fixés pour ce modèle, puis le cadre d'hypothèses que avons défini. Nous décrivons chacune des

quatre parties qui composent notre architecture *OntoDB* : *méta-base*, *données*, *meta-schema*, et *ontologie*. Enfin, nous présentons les différentes APIs d'accès aux ontologies et aux données que nous proposons et qui permettent de masquer la représentation interne de l'information en fournissant des accès au niveau conceptuel. Ces APIs sont de deux types : les APIs à liaison tardive (*late binding*) et les APIs à liaison préalable (*early binding*).

Le chapitre 4 est consacré à l'implémentation du prototype du système de gestion de bases de données à base ontologique que nous avons développé selon le modèle d'architecture *OntoDB* sur le SGBDRO PostgreSQL et qui nous permet de valider l'ensemble des propositions faites dans le chapitre précédent. Nous décrivons successivement comment chacune des parties qui composent notre architecture a été implémentée. Nous avons largement utilisé les techniques d'ingénierie dirigée par les modèles (IDM). L'IDM est une des applications récentes des méthodes de modélisation formelle qui vise à se baser sur des modèles et/ou des méta-modèles pour la génération de programmes. Nous présentons les différentes règles de correspondances que nous avons définis entre le modèle PLIB et le modèle relationnel objet, puis la façon dont ces correspondances sont utilisées pour la génération des schémas de données de la partie *données* de notre architecture. La génération des différentes APIs par les mêmes techniques est également présentée dans ce chapitre.

La troisième partie est consacrée à une évaluation des performances de notre approche par rapport aux approches existantes. Cette partie constitue le chapitre 5.

Dans le chapitre 5, nous faisons une évaluation des performances de notre architecture *OntoDB* essentiellement du point de vue de la gestion des données. Nous présentons d'abord le banc d'essai que nous avons développé. Celui-ci est basé sur une ontologie réelle et normalisée relevant de notre domaine d'application cible : les bases de données de composants industriels et le commerce électronique. Puis nous comparons notre approche de représentation des données à base ontologique avec les principales approches existantes sur la base d'un ensemble de trois familles de requêtes : des requêtes d'interrogation typées pour lesquelles l'utilisateur connaît la ou les classes interrogées, des requêtes d'interrogation non typées et des requêtes de mise à jour des données. Pour la partie *ontologie*, les différents systèmes étant basés sur des modèles d'ontologies différents, les comparaisons ne sont donc pas significatives. Nous vérifions donc seulement la faisabilité de notre approche de représentation des ontologies avec un ensemble d'ontologies *réalistes* de notre domaine d'application cible. Nous ne mesurerons que le temps de navigation et d'accès aux concepts de ces ontologies dans l'éditeur d'ontologies *PLIBEditor* qui permet de naviguer dans la base de données.

Nous terminons cette thèse par une conclusion générale et les perspectives ouvertes de travaux avenir.

Cette thèse comporte trois annexes.

Dans l'annexe A, nous faisons une présentation du modèle d'ontologie PLIB sur lequel se base l'architecture de base de données à base ontologique que nous proposons. Dans l'annexe B, nous présentons brièvement les caractéristiques du SGBR relationnel objet PostgreSQL sur lequel se base notre prototype de base de données à base ontologique. Enfin dans l'annexe C, nous décrivons les règles de correspondances entre les mécanismes du langage de modélisation EXPRESS, utilisé pour définir le modèle PLIB et les mécanismes du relationnel-objet propre au SGBD PostgreSQL.

Publications

La liste suivante représente les publications concernant le travail dans cette thèse.

1. **Hondjack Dehainsala**, Guy Pierra and Ladjel Bellatreche, *OntoDB : An Ontology-Based Database for Data Intensive Applications*, in the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07), LNCS, pp. 497-508, Thailand, April, 2007.

2. Stéphane Jean, **Hondjack Dehainsala**, Dung Nguyen Xuan, Guy Pierra, Ladjel Bellatreche and Yamine Aït-Ameur, *OntoDB : It is Time to Embed your Domain Ontology in your Database* - Demo Paper- , in the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07), LNCS, pp. 1119-1122, Thailand, April, 2007.

3. **Hondjack Dehainsala**, Guy Pierra and Ladjel Bellatreche, *Benchmarking Data Schemes of Ontology Based Databases*- Poster Paper -, Proceeding of On the Move to Meaningful Internet Systems 2006 (OTM'06) : ODBASE conference, LNCS, pp. 48-49, Montpellier, Novembre, 2006, .

4. Ladjel Bellatreche, Dung Nguyen-Xuan, Guy Pierra, **Hondjack Dehainsala**, *Contribution of Ontology-based Data Modeling to Automatic Integration of Electronic Catalogues within Engineering Databases*, Computers in Industry Journal 57 (8-9), pp. 711-724, 2006.

5. Guy Pierra, **Hondjack Dehainsala**, Yamine Ait-Ameur and Ladjel Bellatreche,*Base de données à base ontologique : principe et mise en œuvre*, Ingénierie des Systèmes d'Information (ISI), vol. 10, 2, pp.91-115, 2005.

6. **Hondjack Dehainsala**, Stéphane Jean, Dung Nguyen-Xuan,Guy Pierra, *Ingénierie dirigée par les modèles en EXPRESS : un exemple d'application*, Actes des premières journées d'Ingénierie dirigée par les modèles, IDM'05, pp. 155-174, Paris, 2005.

7. Guy Pierra, **Hondjack Dehainsala**, Nadège Ngabiapsi Negue and Mounira Bachir., *Transposition relationnelle d'un modèle objet par prise en compte des contraintes d'intégrité de niveau instance*, congrès INFORSID, pp. 455-470, Grenoble, 2005,.

8. Guy Pierra, **Hondjack Dehainsala**, Yamine Ait Ameur, Ladjel Bellatreche, Jérôme Chochon and Mourad El-Hadj Mimoune., *Base de Données à Base Ontologique : le modèle OntoDB*, 20 èmes Journées Bases de Données Avancées (BDA 2004), pp. 263-286, Montpellier, 2004.

9. **Hondjack Dehainsala**, *Base de Données à Base Ontologique* - Poster Paper-, Actes du 23eme congrès Inforsid, pp. 539-540, Biarritz, 2004.

10. Ladjel Bellatreche, Guy Pierra, Dung Nguyen-Xuan, **Hondjack Dehainsala**, Yamine Ait Ameur, *An a Priori Approach for Automatic Integration of Heterogeneous and Autonomous Databases*, in proceedings of the 8th International Conference on Databases and Expert Systems (DEXA'04), LNCS, pp. 475-485, Zaragoza, Aout, 2004.

11. Ladjel Bellatreche, Guy Pierra, Dung Nguyen-Xuan, **Hondjack Dehainsala**, *Integration de sources de données autonomes par articulation a priori d'ontologies*, Actes du Congrès d'Informatique Des Organisations et Systèmes D'Information et de Décision (INFORSID'2004), pp. 283-298, Biarritz, 2004.

12. Ladjel Bellatreche, Guy Pierra, Dung Nguyen-Xuan, **Hondjack Dehainsala**, *An Automated Information Integration Technique using an Ontology-based Database Approach*, in Proceedings of Concurrent Engineering Conference, CE'2003 - Special Track on Data Integration in Engineering (DIE'2003), pp. 217-224, Madera, 2003.

Première partie

État de l'art

Chapitre 1

Approche traditionnelle de conception des bases de données

Introduction

Depuis plus de trente ans, basées sur l'architecture à trois niveaux définie par le modèle ANSI/SPARC, les bases de données ne cessent de s'ouvrir à de nouveaux domaines d'application. Après leur généralisation dans le domaine de la gestion dans lequel les données manipulées sont en général représentées par des types simples (atomiques), elles se généralisent désormais dans les domaines des données techniques (CAO, SGDT, etc.), des données multimédia et des applications Web. Parallèlement, les formalismes de modélisation conceptuels ont évolués. Si le modèle entité/association [35] reste encore assez largement utilisé, les formalismes orientés objets tendent à se généraliser, que se soit à travers UML (Unified Modeling Language) [83], EXPRESS [79, 143] dans le domaine technique, ou les schémas XML (Extensible Markup Language) [26] pour les données du Web.

Cette émergence des objets et des mécanismes qui leur sont associés : héritage, polymorphisme, composition, tant au niveau de la modélisation conceptuelle, qu'au niveau des types d'information qu'un système de gestion de bases de données doit supporter, rendent de plus en plus difficile l'utilisation du classique modèle relationnel au niveau logique [35].

Au cours des quinze dernières années, trois grandes approches ont été mises en œuvre pour réduire cette distance entre les modèles conceptuels et leurs modèles logiques.

- La première a consisté à utiliser, au niveau logique, et à représenter effectivement en bases de données, les mécanismes de base de l'approche objet tels qu'ils étaient utilisés au niveau conceptuel. Les modèles conceptuels et logiques ne faisaient alors plus qu'un. Ce sont les bases de données objets [12], dont beaucoup pensaient dans les années 1990 qu'elles remplaceraient inéluctablement les systèmes relationnels. Malheureusement, pour diverses raisons, et en particulier parce que les performances n'ont pas été au rendez-vous, il est semble admis désormais que leurs domaines d'application devraient se limiter à des niches particulières n'exigeant pas, simultanément, des données de grande taille et un traitement rapide des requêtes [152].

- La deuxième approche, qui triomphe actuellement, est l'approche relationnel-objet [52]. Elle consiste à conserver le concept de base du relationnel (les relations ou tables), qui constituent la clé de l'efficacité du relationnel et à utiliser ces tables pour ranger des objets. Le rapprochement des formalismes conceptuels et logiques se fait ici en injectant un plus ou moins grand nombre de concepts objets au niveau du formalisme logique. Cette approche garde, en théorie, toute l'efficacité de l'approche relationnelle, et elle est si prometteuse que tous les éditeurs actuels de systèmes relationnels (Oracle, IBM, Microsoft et Sybase) ont une offre relationnelle-objet. Le principal problème de cette approche est que, s'il est vrai que tout système relationnel-objet supporte certains mécanismes objets, ce ne sont par contre en général pas les mêmes d'un système à un autre. Ceci rend les nouveaux systèmes de plus en plus incompatibles et donc les applications de moins en moins portables.
- Une troisième approche, de plus en plus utilisées au cours des dix dernières années, consiste à définir des correspondances entre les concepts du paradigme objet et les concepts du relationnel [135, 151, 110]. Bien que largement utilisée, cette approche souffre néanmoins de l'éclatement des concepts objets en nombreuses relations. Pour l'utilisateur final, ou pour le programmeur d'applications, la manipulation de ces modèles logiques est difficile. Les modèles logiques résultant de la traduction des modèles objets (1) perdent de leurs sémantiques, (2) réduisent souvent l'efficacité de traitement des données et (3) augmentent le coût de développement des applications chargées de fournir à l'utilisateur final des accès correspondant au niveau conceptuel. Ce sont précisément ces difficultés que l'approche des bases de données à base ontologique que nous proposons vise à reduire.

Le but dans ce chapitre, est de présenter brièvement un état de l'art sur les méthodes actuelles de conception de bases de données et sur les difficultés sur lesquelles elles débouchent. Nous présenterons également, à cette occasion, certains des outils que nous utiliserons dans ce travail : le langage de spécification EXPRESS et son environnement d'ingénierie dirigée par les modèles et les applicatifs supportant la correspondance objet-relationnel et en particulier l'applicatif *Hibernate*.

Dans la section 1, nous précisons la terminologie que nous utiliserons tout au long de ce travail. Dans la section 2, nous nous intéressons à la conception orientée objet des modèles conceptuels. Nous présentons à cette occasion, le langage de modélisation EXPRESS sur lequel nous nous baserons tout au long de cette thèse pour la description et la représentation des modèles que nous voulons traiter. Dans la section 3, nous nous intéresserons à la représentation des données. Nous montrons d'abord que la tendance dominante est d'utiliser des modèles de représentation de type relationnel (ou relationnel objet), très différents donc des modèles conceptuels, et nous discutons plus en détail les méthodes et outils permettant de passer d'un univers à l'autre. Nous illustrons ce type d'approche par l'applicatif Hibernate que nous utiliserons également pour certains aspects de notre architecture. La section 4 met en évidence deux inconvénients des méthodes usuelles de conception. La divergence des méthodes de modélisation conceptuelle objet et de représentation relationnelle rend peu compréhensible les données résultantes. La conception isolée de chaque modèle conceptuel rend l'intégration ultérieure de deux bases de données très difficiles à réaliser. On montre alors le rôle que pourraient jouer les ontologies de domaine pour

réduire ces inconvénients.

1 Terminologie

Nous définissons ici la terminologie que nous utiliserons dans cette thèse pour discuter des niveaux conceptuels et logiques des bases de données.

1.1 Le niveau conceptuel

Le niveau conceptuel s'intéresse à l'appréhension, par un être humain, d'un domaine de connaissance [62].

Une **information** est une *connaissance* sur un fait, un concept et un processus.

Un modèle dépend du point de vue et son objectif. Ainsi pour Minsky [114], un objet A* est le modèle d'un objet A pour un observateur *B*, s'il aide **B** à répondre aux questions qu'il se pose sur A.

Ainsi, un **modèle d'information** est un objet qui permet de répondre aux questions que l'on se pose sur un ensemble d'information.

Un **formalisme de modélisation conceptuel** (ou sémantique) est un ensemble de notations précises, associé à une signification précise et à des règles de raisonnement et qui permet de représenter un *modèle d'information* [62].

Un modèle conceptuel étant une représentation de la connaissance propre à un domaine, un formalisme de modélisation sera d'autant plus puissant qu'il permet de représenter les différents catégories de *connaissances*.

Une **connaissance** possède trois dimensions [62] :

- **Dimension structurelle** : l'un des modes d'appréhension du monde réel par l'homme est la formulation de *classes* (ou *catégories* ou *types* d'entités) d'objets, leur représentation et leur différentiation. Ces catégories sont organisées au moyen de *relations*. Une relation particulièrement importante est la relation de généralisation/spécialisation qui va du général vers le particulier et définit des hiérarchies ou, plus généralement des graphes acycliques. Cette relation est appelée également relation de *subsomption*.

- **Dimension descriptive** : la connaissance descriptive associe aux catégories des *propriétés* (ou *attribut* ou *slot*) qui permettent de discriminer les instances des catégories.

- **Dimension procédurale** : la connaissance procédurale correspond au comportement et aux règles de raisonnement qui peuvent être appliquées aux différentes instances de chaque catégorie. Ce type de connaissance est représenté par des règles, des fonctions, et des pro-

cédures.

Un **modèle conceptuel** est donc caractérisé par (1) le domaine auquel il s'intéresse, (2) le formalisme qui a permis de le définir, et enfin (3) le contexte ou le point de vue qu'il souhaite représenter, et qui définit les questions auxquelles il vise à répondre.

Concernant ce dernier point, il faut noter que les modèles d'information dépendent très fortement des besoins applicatifs pour lesquels ils ont été définis, car les questions auxquelles il vise à répondre ne sont jamais exactement les mêmes. C'est la raison pour laquelle les modèles de deux bases de données portant, pourtant sur des domaines qui apparaissent identiques, sont *toujours différents*. Nous verrons dans le chapitre 2 que ce point constitue une différence essentielle avec les ontologies (cf. section 1).

Notons également que ce sont essentiellement les modèles objets qui ont proposés de représenter certains éléments de connaissance procédurale au niveau des modèles conceptuels entraînant une divergence croissance entre modèle conceptuel et spécification de données.

1.2 Le niveau logique

Une **donnée** est un symbole qui représente une information, basé sur des règles d'interprétation implicites ou explicites [143].

Une **spécification de données** est une description d'un ensemble de données sous forme automatiquement implémentable et qui est associée à des règles d'interprétation explicite ou implicite.

Distinction conceptuel/logique

La distinction entre le niveau conceptuel et le niveau logique est universellement admise dans la communauté des bases de données. Ceci est complété par l'architecture ANSI/SPARC[11, 8] qui spécifie trois niveaux abstraction pour les SGBDs.

1. Le **niveau conceptuel**, où la connaissance du domaine (ou univers du discours) est formalisé dans un formalisme de modélisation conceptuel (E/A, Merise [154], OMT [141], UML [83, 120], EXPRESS [79, 143], etc.).
2. En dessous, le **modèle logique** est une spécification des données telle qu'elle sera implémentée sur le système de base de données.
3. Au dessus, les modèles externes ou les *vues utilisateurs* qui permettent d'adapter les données fournies aux besoins des différentes catégories d'utilisateurs.

Les modèles conceptuels sont définis dans des formalismes de modélisation. Ces formalismes sont tous plus ou moins différents les uns des autres. Nous décrivons brièvement dans la section qui suit le langage EXPRESS qui représente un des meilleurs exemple de tel formalisme et sera ensuite utilisé tout au long de ce travail et en particulier pour la définition du modèle d'ontologie

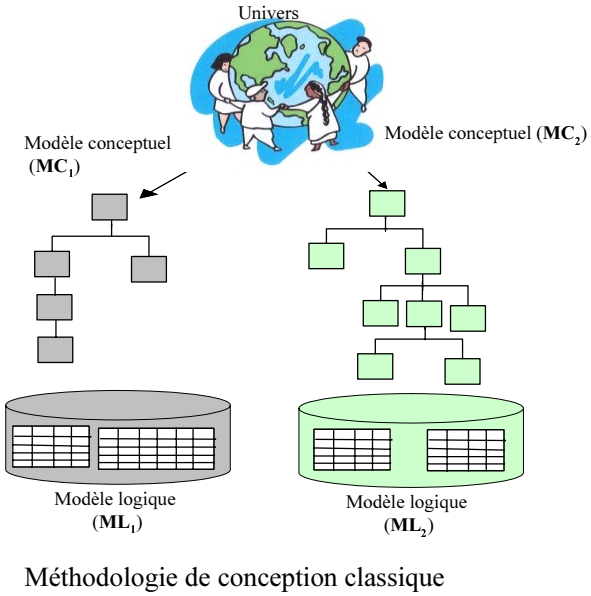


FIG. 1.1 – Approche classique de conception de bases de données

PLIB [80, 81]. Nous faisons ensuite une brève comparaison du langage EXPRESS avec le langage UML pour mieux cerner les particularités de ces deux langages actuels.

2 Conception orientée objet : le Langage EXPRESS

Le langage EXPRESS est un langage orienté objet qui présente la particularité d'avoir été conçu exclusivement pour concevoir des modèles de données. II a été défini dans le cadre d'un projet de normalisation de l'ISO intitulé STEP (STandard for Exchange Product model data) initié à la fin des années 1980. STEP avait pour objectif la normalisation de modèles de données pour les différents produits fabriqués en entreprise. Comme il n'existait pas de langage, à l'époque, pour représenter de façon formelle des modèles d'information de grande taille, le projet STEP[1] a donc commencé par développer à la fois un langage de modélisation et toute une technologie associée. Ceci a abouti à la production d'une série de normes définissant un environnement appelé EXPRESS :

- un langage de modélisation de l'information : EXPRESS (ISO 10303-11 :1994) ;
- un format d'instances qui permet l'échange de données entre systèmes (ISO 10303-21 :1994) ;
- une infrastructure de méta modélisation associée à une interface d'accès normalisée, appelée SDAI (Standard Data Interface). Cette infrastructure permet d'accéder et de manipuler simultanément les données et le modèle de n'importe quel modèle EXPRESS. Cette interface, associée à un méta modèle d'EXPRESS défini en EXPRESS, a d'abord été définie indépendamment de tout langage de programmation (ISO 10303-22 :1998). Puis des implémentations spécifiques ont été spécifiées pour le langage C++ (2000), Java (2000), C (2001) ;

[1] www.tc184-sc4.org/

– un langage déclaratif de transformation de modèles EXPRESS-X(ISO 10303-14 :2002).

Pour contrôler l'intégrité des données, le langage EXPRESS possède un langage procédural complet (analogue au langage PASCAL) pour l'expression de contraintes. Au prix de quelques extensions mineures, essentiellement la possibilité de lancer un programme et de faire des entrées-sorties, ce langage peut également être utilisé comme langage impératif de transformation de modèles. Cette extension est effectuée dans tous les environnements de programmation EXPRESS. Dans l'environnement que nous utiliserons, cette extension porte le nom d'*EXPRESS-C*.

Depuis une dizaine d'années, de nombreux environnements de modélisation EXPRESS sont commercialisés et le langage EXPRESS est utilisé dans de nombreux domaines, que se soit pour modéliser et échanger des descriptions de produits industriels, ou pour spécifier et générer des bases de données dans des domaines divers, ou même pour fabriquer des générateurs de codes dans des ateliers de génie logiciel [130].

Nous présentons succinctement, dans la section qui suit, les concepts généraux du langage EXPRESS et son environnement de modélisation.

2.1 Le langage

EXPRESS est un langage de modélisation formel de l'information. Une particularité très intéressante d'EXPRESS est qu'il permet à la fois de modéliser les concepts d'un domaine (modèle conceptuel) et de spécifier les structures de données permettant de représenter ces concepts sous forme de données traitables par machine (modèle logique), tout ou moins dans un contexte des bases de données orientée objets. C'est un langage orienté objet : il supporte l'héritage et le polymorphisme. Il est également ensembliste : il permet les manipulations de collections. Nous présentons succinctement ses composantes structurelles, descriptives et procédurales.

2.1.1 Connaissance structurelle

Le langage EXPRESS utilise la notion d'*entité* (ENTITY), pour réaliser l'abstraction et la catégorisation des objets du domaine de discours. Une entité est similaire à une classe des langages à objets à la différence qu'elle ne définit pas de méthodes. Les entités sont hiérarchisées par des relations d'héritage qui peuvent relever de l'héritage simple, multiple ou répété.

2.1.2 Connaissance descriptive

Une entité est décrite par des attributs. Les attributs permettent de caractériser les instances d'une entité par des valeurs. Une attribut peut aussi bien représenter une (collection de) valeur(s) qu'une relation avec une (collection d') autre(s) entité(s). Un attribut peut être défini indépendamment de tout autre attribut (attributs libres) ou calculé à partir des valeurs d'autres attributs (attributs dérivés). Un type de données est associé à chaque attribut. Le langage EXPRESS définit quatre familles de types.

– *Les types simples* : ce sont essentiellement les types chaînes de caractères (STRING), numériques (REAL, BINARY, INTEGER) ou logiques (LOGICAL, BOOLEAN).

- *Les types nommés* : ce sont des types construits à partir de types existants auxquels un nom est associé. Un type nommé peut être défini par restriction du domaine d'un type existant. Cette restriction peut être faite par la définition d'un prédicat qui doit être respecté par les valeurs du sous domaine créé. Il peut également être défini par énumération (ENUMERATION) ou par l'union de types (SELECT) qui, dans un contexte particulier, sont alternatifs.
- *Les types agrégats* : ce sont des types qui permettent de modéliser les domaines dont les valeurs sont des collections. Les types de collections disponibles sont les ensembles (SET), les ensembles multivalués (BAG), les listes (LIST) et les tableaux (ARRAY). Un type collection n'a pas a être nommé : il apparaît directement dans la définition de l'attribut qu'il type.
- *Les types entités* : un attribut d'un tel type représente une association.

2.1.3 Connaissance procédurale

Le langage EXPRESS est très expressif au niveau des contraintes. Celles-ci peuvent être classifiées selon deux grandes familles. Les *contraintes locales* s'appliquent individuellement sur chacune des instances de l'entité ou du type sur lequel elles sont définies. Les *contraintes globales* nécessitent une vérification globale sur l'ensemble des instances d'une (ou d'un ensemble d') entité(s) donnée(s).

Les contraintes locales (WHERE) sont définies au travers de prédicats auxquels chaque instance de l'entité (ou chaque valeur du type), sur lequel elles sont déclarées, doit obéir. Ces prédicats permettent par exemple de limiter la valeur d'un attribut en définissant un domaine de valeurs, ou encore de rendre obligatoire la valuation d'un attribut optionnel selon certains critères.

Concernant les contraintes globales :
- La contrainte d'unicité (UNIQUE) spécifique l'unicité de valeur d'un (ensemble d') attribut(s) sur l'ensemble de la population d'instances d'une même entité.
- La contrainte de cardinalité inverse (INVERSE), permet de spécifier la cardinalité de la collection d'entités d'un certain type qui référence une entité donnée dans un certain rôle.
- Les règles globales (RULE) ne sont pas déclarées au sein des entités mais sont définies séparément. Elles permettent d'itérer sur une ou plusieurs population d'entités pour vérifier des prédicats qui doivent s'appliquer à l'ensemble des instances de ces populations.

Des fonctions (FUNCTION) et des procédures (PROCEDURE) peuvent être écrites. En EXPRESS standard, elles ne peuvent être utilisées que dans l'écriture de fonctions de dérivations ou de prédicats servant à définir des contraintes. De plus, le langage EXPRESS propose un ensemble de fonctions prédéfinies : QUERY (requête sur les instances d'une classe), SIZEOF (taille d'une collection), TYPEOF (introspection : donne le type d'un objet), USED_IN (calcul dynamique des associations inverses). L'ensemble de ces constructions permet l'expression de contraintes d'intégrité ou de fonctions de dérivations arbitrairement complexes telles que l'acyclicité d'un graphe ou la régularité d'un système d'équations linéaires.

```
SCHEMA Universitaire ;

TYPE CLASSE = ENUMERATION OF (A1, A2, A3);
END_TYPE;

ENTITY Personne ;
SUPERTYPE OF ONEOF (Etudiant, Salairié);
  noSS : NUMBER;
  nom : STRING ;
  prénom : STRING;
  age : INTEGER;
  conjoint : OPTIONAL Personne;
DERIVE
 nom_prenom : STRING := Nom_complet (SELF);
UNIQUE ur1 : NoSS;
END_ENTITY ;

ENTITY Notes ;
  module_ref : STRING;
  note_40 : REAL ;
WHERE
  wr1: {0 <= note_40 <= 40};
DERIVE
  note_20 : REAL := note_40/2;
END_ENTITY ;

ENTITY Etudiant ;
SUBTYPE OF (Personne);
  sa_classe : CLASSE;
  ses_notes : LIST[0 : ?] OF NOTES;
INVERSE
  appartient_à : SET[0 :?] OF Etudiant FOR ses_notes ;
END_ENTITY ;

ENTITY Salarié;
SUBTYPE OF (Personne)
  salaire : REAL;
END_ENTITY ;

ENTITY Etudiant_Salarié;
SUBTYPE OF( Salarié, Etudiant);
END_ENTITY ;

FUNCTION Nom_complet(per : Personne): STRING;
RETURN (per.nom + ' ' + per.prenom);
END_FUNCTION;

END_SCHEMA ;
```

FIG. 1.2 – Exemple d'un schéma EXPRESS.

Pour illustrer cette présentation d'EXPRESS considérons la figure 1.2. Dans cet exemple, nous définissons un schéma de nom *universitaire*. Ce schéma est constitué de cinq entités. Les entités *étudiant* et *salarié* qui héritent de l'entité *Personne*. L'entité *étudiant_ salarié* qui hérite des entités (héritage multiple et répété) *étudiant* et *salarié*. Enfin l'entité *notes* qui est co-domaine de l'attribut *ses_ notes* de *étudiant*. L'entité *Personne* définit un attribut dérivé (*nom_ prénom*) qui est associé à la fonction EXPRESS (*nom_ complet*) retournant la concaténation de l'attribut *nom* et *prénom* d'une instance de l'entité *Personne*. On peut remarquer la contrainte locale dans l'entité *notes* qui permet de vérifier que les valeurs de l'attribut *note_ 40* sont comprises entre 0 et 40.

2.1.4 Représentation Graphique d'EXPRESS

EXPRESS possède également une représentation graphique (sauf pour les contraintes) appelée EXPRESS-G. Cette représentation a pour objectif de donner une vue synthétique d'un schéma EXPRESS et est adapté aux phases préliminaires d'une conception. La représentation graphique du schéma EXPRESS de la figure 1.2 est présentée dans la figure 1.3.

2.1.5 Modularité

Destiné a concevoir des modèles de tailles importantes, par exemple l'ensemble des entités constituants un avion, EXPRESS intègre des mécanismes de modularité permettant la décomposition d'un modèle complexe en plusieurs sous-modèles. Ceci permet de faciliter la conception, la maintenance et la réutilisation d'un modèle. Un modèle EXPRESS, appelé schéma, peut faire référence à un ou plusieurs autres schémas, soit pour intégrer tout ou partie des entités définies dans ceux-ci (USE), soit uniquement pour exploiter entité et type référencés pour typer ses propres attributs (REFERENCE). Le découpage de la modélisation d'un domaine donné peut se faire selon deux approches qui peuvent être combinées :

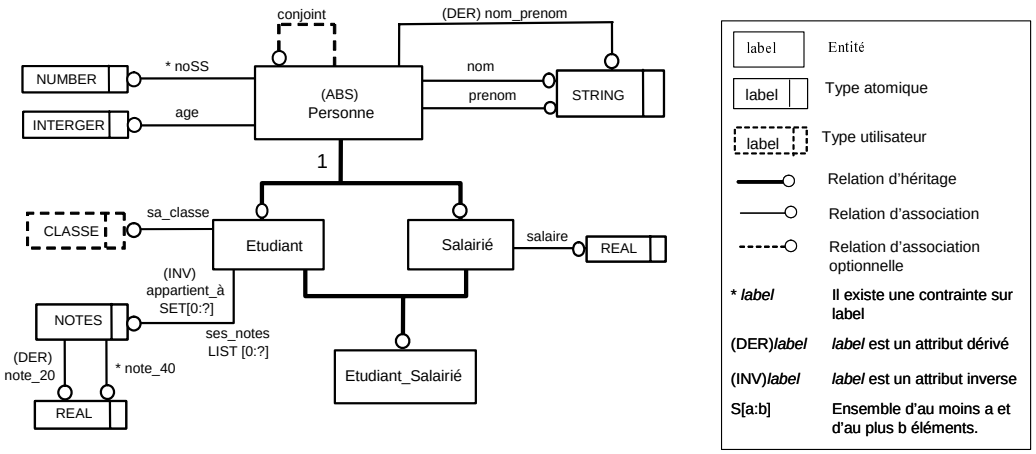


FIG. 1.3 – Exemple d'un Schema EXPRESS en EXPRESS-G.

- horizontal : chaque schéma modélise un sous domaine du domaine considéré ;
- vertical : chaque schéma représente une modélisation du domaine à un différent niveau d'abstraction.

2.1.6 Représentation des instances

A tout modèle EXPRESS, est automatiquement associé un format textuel de représentation d'instances permettant de réaliser l'échange entre systèmes. Ce format est appelé *fichier physique* [ISO10303-21 :1994]. Chaque instance est identifiée pour un oid (Object IDentifier : #i). Elle est caractérisée par le nom de la classe instanciée. Enfin, elle est décrite par la liste des valeurs de ses attributs représentée entre parenthèses. Les attributs optionnels sont désignés par '$', et les types énumérés par une notation encadrant l'identificateur par un point. Les collections d'éléments sont définies entre parenthèses. Les attributs dérivés et inverses ne sont pas représentés. Les instances de la figure 1.4 sont des instances des entités du schéma de la figure 1.3.

```
#2 = SALAIRE(13457, 'Jean', 'Hondjack' , 27, #3, 1000);
#3 = SALAIRE(23215, 'Jean', 'Nathalie', 25, #2, 2500);
...
#100 = ETUDIANT(02489, 'Nguyen', 'Sylviane' , 18, $, 'A3', (#101, #102));
#101 = NOTE('A3_120', 31);
#102 = NOTE('A3_121', 28);
```

FIG. 1.4 – Un exemple de fichier physique correspondant aux entités de la figure 1.3.

2.2 EXPRESS comparé à UML pour la modélisation objet

Comme UML, EXPRESS est un langage de modélisation objet mais il possède quelques différences du point de vue de la représentation de certains concepts objets.

2.2.1 Représentation de relation d'association entre classes (entités)

Une association est une relation structurelle qui permet de lier n ($n > 1$) classes entre elles. Lorsque n est égal à 2 on dit que c'est une *relation binaire*. Contrairement à UML, en EXPRESS, il n'existe pas de "constructeur" pour représenter des associations. Les associations s'expriment au moyen d'attributs dont le co-domaine est une autre entité et d'attributs inverses. Par exemple, pour une association binaire entre les classes A et B, on déclare un attribut *a2b* dans l'entité A et, eventuellement, un attribut inverse *b2a* dans l'entité B si l'on souhaite naviguer aussi de B à A. Les associations n-aires se représentent par une classe avec n attributs dont les co-domaines sont les entités.

2.2.2 Représentation des agrégations, compositions et collections

Une agrégation et une composition correspond à une relation entre deux classes dans laquelle une extrémité de la relation (composite) joue un rôle prépondérant par rapport à l'autre (composant). En UML, il correspond à une association avec un losange du coté de l'agrégat. En EXPRESS, agrégation et composition ne sont pas représentées de façon spécifique. Les collections s'expriment sous forme d'attributs de type agrégat (SET, LIST, BAG, ARRAY) d'entités ayant éventuellement un attribut de type simple ou agrégat. Les spécificités de la relation s'expriment au moyen de cardinalités directes et inverses des attributs. Par exemple, une composition entre A et B s'exprimera par un attribut *a2b* de type B, son attribut inverse *b2a* ayant une cardinalité 0 :1 ou 1 :1 et pas d'autres associations.

2.2.3 Différents types d'héritage en EXPRESS

Dans le langage EXPRESS, comme en UML, une entité déclare ses super-types. Elle peut aussi déclarer ses sous-types en utilisant trois opérateurs (ONEOF, AND, ANDOR) qui imposent ou permettent à une entité d'être instance d'une ou de plusieurs de ses sous-types. Illustrons ces opérateurs sur des exemples :

- Person SUPERTYPE OF ONEOF(male, female) : une instance de l'entité *person* peut être soit une instance de l'entité *male* soit une instance de l'entité *female*
- Person SUPERTYPE OF (employee ANDOR student) : une instance de *person* peut être une instance de l'instance *employee* mais aussi simultanément une entité de l'entité *student*.
- person SUPERTYPE OF (ONEOF(female,male) AND ONEOF(citizen, alien)) : une instance de l'entité *person* peut être une instance de l'entité *male* ou de *female* et une instance de l'entité *citizen* ou de l'entité *alien*.

2.2.4 Méthodes

EXPRESS est un formalisme de modélisation des données. Il permet de représenter les contraintes d'intégrités, fonctions de dérivations, mais à la différence d'UML, il ne permet de représenter aucune autre méthode. Ceci définit clairement des points de vue d'EXPRESS, la frontière entre "base de données" et "application", et ce, même dans un système objet.

Dans la méthode traditionnelle de conception des bases de données, le développement d'un modèle conceptuel constitue la première étape. Une fois celui-ci conçu, le plus souvent désormais dans un langage objet, typiquement EXPRESS ou UML se pose la question de gestion des données. C'est le développement du modèle logique que nous discutons dans la section suivante.

3 Gestion persistante des objets

Dans cette section, nous discutons de l'évolution des modèles spécifications de données qui s'est réalisée en parallèle avec la généralisation de l'approche objet au niveau conceptuel. Nous discuterons successivement des trois modèles proposés [82] : le modèle relationnel, le modèle objet et le modèle relationnel-objet (apparu pour concilier les univers relationnel et objet).

Enfin, nous discutons de façon plus approfondie des méthodes et outils permettant une coexistence entre les applications et le modèle objet d'une part, et, la représentation relationnelle d'autre part, puisque c'est le problème que nous devrons traiter pour la représentation des ontologies dans les bases de données (cf. chapitre 3).

3.1 Le modèle relationnel

Le modèle relationnel, apparu dans les années 70 [40], visait à fournir un modèle simple pour faciliter l'accès aux données. Le modèle possède une seule notion : celle de relation mathématique représentée comme une table. Les opérations (création, suppression, insertion, mise à jour, etc.) sur les données sont faites aux moyens du langage SQL (Structured Query Language) qui est un langage déclaratif. Aujourd'hui encore, le modèle relationnel domine ses concurrents [152] parce qu'il est (1) efficace pour gérer des données de très grande taille, (2) basé sur des fondements mathématiques solides (l'algèbre relationnelle), (3) est basé sur des concepts simples, consistants et faciles d'utilisation [151] et (4) normalisé. Toutefois, il est limité pour les applications nécessitant des structures de données complexes (CAO, multimédia, SIG, IA, etc.).

3.2 Le modèle objet

Une deuxième catégorie de systèmes de bases de données [12, 91] est apparue dans les années 80 après l'émergence des concepts objets. Les bases de données orientées objets (BDOO) offrent la possibilité de représenter directement les instances des classes d'un modèle objet. Les objets sont donc stockés de façon persistante contrairement aux objets des langages de programmation orientés objets. Afin d'assurer la portabilité des interfaces d'un SGBD objet à un autre. L'ODMG a proposé un modèle abstrait de données : OQL-*Object Query Language* qui a été ensuite implémenté pour les langages de programmation orienté objet, tels que C++, Smalltalk et Java. Les avantages étaient les suivants. (1) Le modèle conceptuel et le modèle logique deviennent identiques. Les classes définies au niveau du modèle conceptuel sont directement représentées dans la base de données empêchant tout problème d'"impédance mismatch". (2) Une BDOO offre la possibilité de stocker des données de types diverses (Multimédia, SIG, CAO, etc.). (3) Elle est moins assujettie aux erreurs lors des mises à jour [91]. Malgré ces avantages, force est de constater que le modèle objet n'a pas connu le même succès commercial que le modèle relationnel

(cf. section 3.1). Aujourd'hui les BDOO sont cantonnées à des marchés de niche. Leur modèle n'est plus considéré comme une solution générale pour la définition des modèles logiques.

3.3 Le modèle relationnel objet

Le modèle relationnel objet est un compromis entre le modèle objet et le modèle relationnel [152]. L'idée de base, proposée par Atkinson et al. [12] lors du deuxième manifeste des bases de données objet, était de tirer profit de la maturité et des performances du modèle relationnel et des avantages de la modélisation objets des données. Cela a abouti à la définition d'une extension de la théorie du relationnel pour supporter les données ne respectant pas la première forme normale (1FN). Il fournit des types abstraits de données (ADT), des types complexes de données, les collections, l'encapsulation, l'héritage, des OIDs pour les instances. Un nouveau SQL en a découlé et est normalisé aujourd'hui sous le nom de SQL99 [52]. Plusieurs éditeurs de SGBD relationnels (Oracle, DB2, PostgreSQL) se sont lancés dans cette approche pour étendre leurs produits.

Aujourd'hui, dans le contexte industriel, on peut constater si la modélisation orientée-objet s'impose face aux autres approches de modélisation de l'information [151], au niveau représentation des données, les SGBDRs et les SGBDROs s'imposent dans la plupart des cas par rapport aux SGBDOOs. Néanmoins, les SGBDROs qui implémentent la norme SQL99 [52] sont très récents et ne supportent tous qu'une partie de la norme. Par exemple, PostgreSQL implémente l'héritage de table, ce que ne fait pas Oracle. Par contre, Oracle implémente les associations entre classes avec la possibilité de polymorphisme et le calcul des références inverses des associations, ce qui n'existe pas en PostgreSQL. Cette différence d'implémentation fait que les schémas de bases de données ne sont pas portables d'un SGBDRO à un autre. Pour cette raison, une des approches très fréquente consiste à marier un schéma objet avec un schéma essentiellement relationnel, c'est l'approche hybride que nous présentons dans la section 3.4.

Notons que le prototype de base de données à base ontologique que nous avons implémenté au cours de notre thèse a été réalisé sur le SGBDRO PostgreSQL dont nous présentons les principales caractéristiques dans l'annexe B.

3.4 Approches hybrides de conception de bases de données : les mappings objet/relationnel

Compte tenu des divergences croissantes apparaissant entre les formalismes de modélisation conceptuelle et logique, depuis les années 1990, plusieurs études ont été menées [37, 101, 106, 152] en vue de concilier l'univers objet de l'approche relationnelle. Deux grandes approches ont été proposées :

1. La connexion des applications objets directement aux bases données relationnelles en codant dans les méthodes de classes ("getters", "setters", constructeurs, destructeurs, etc.) l'accès aux données. Cet accès se fait généralement par du "*embedded SQL*" repartit dans tout le code des applications.

2. La définition d'un *gestionnaire d'objets* (ou *object layers*) entre les applications objets et les bases de données relationnelles (cf. figure 1.5). Le gestionnaire d'objets (1) reçoit les requêtes (OQL par exemple) venant des applications objets, (2) les traduit en SQL et les envoie au SGBDR pour exécution ; (3) récupère les résultats et enfin (4) les traduit de nouveau dans le format adapté (OQL par exemple) pour les applications objets. Exemple : Ibatis [94], Hibernate [70], JDO [132], etc.

Les deux approches précédentes partagent le fait que les objets des classes de l'application sont stockés dans des bases de données de type relationnel (ou relationnel-objet). Ces approches nécessitent donc de définir des correspondances entre les concepts objets et les concepts relationnels, puis de gérer de façon la plus simple possible les transformations associées à ces correspondances.

Dans la section 3.4.1, nous discutons les principales règles de correspondances objet/relationnel qu'on trouve dans la littérature. Nous devrons en effet choisir de telles règles pour représenter nos ontologies dans nos BDBOs. Dans la section 3.4.3, nous présentons un exemple d'outils pour la gestion des transformations, à savoir le "framework" *hibernate* qui est un outils de transformation *objet / relationnel* paramétrable très fréquemment utilisé et que nous utiliserons dans notre travail.

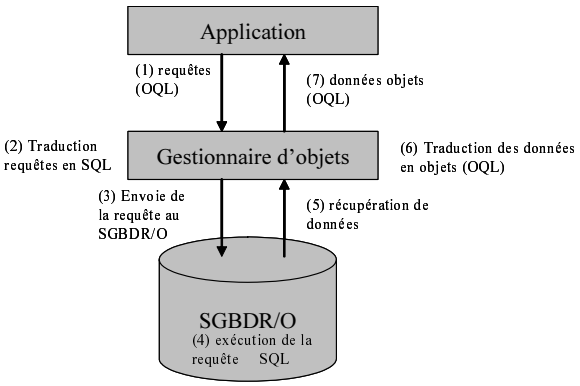


FIG. 1.5 – Approche hybride de représentation d'objets : gestionnaire d'objets

3.4.1 Règles de transposition objet-relationnel

Plusieurs approches ont été proposées pour la représentation des concepts objets dans un environnement relationnel. Nous discuterons essentiellement ici les propositions portant sur la représentation des relations d'héritage [134, 141, 97, 53, 108], d'associations et d'agrégations [54, 145] qui sont les principaux concepts objets qui exigent des règles de transformations particulières pour être représentés en relationnel.

3.4.1.1 Représentation de la relation d'héritage

Les propositions de représentation de la relation d'héritage peuvent être classées en trois catégories :

1. Représentation "à plat",

2. Représentation horizontale,
3. Représentation verticale.

1. *Représentation "à plat"* : une table par arborescence
   Dans cette stratégie, tous les attributs de toutes les classes d'une hiérarchie sont stockés dans une seule et même table. Le nom de la racine de la hiérarchie est le plus souvent utilisé pour nommer cette table. Deux autres attributs sont ajoutés. Le premier représente l'identifiant, ou clé primaire de la table. Le second est généralement un code spécifiant le type effectif de l'objet instancié (cf. (1a) dans la figure 1.6). Une variante de cette approche consiste à remplacer ce second attribut par plusieurs attributs de type booléen (cf. (1b) dans la figure 1.6) ; chacun d'eux correspondants à un des types possibles.

   Cette approche est simple et supporte le polymorphisme car tous les objets polymorphes appartiennent à la même table. Ils peuvent donc être référencés par une même clé étrangère. L'accès aux instances d'un niveau quelconque de la hiérarchie demande seulement la projection d'une sélection car toutes les instances sont rangées dans une seule table. C'est la seule méthode qui sera économe en jointure. L'inconvénient, par contre, se situe au niveau de l'espace de stockage car les tables peuvent être très grandes et contenir de nombreuses valeurs nulles [5]. Ceci est en particulier le cas lorsque des motifs de classes abstraites sont utilisés pour représenter des mécanismes génériques, tels, par exemple, le motif Modèle-Vue-Controleur (MVC) [96]. La mise à plat impose alors de regrouper un très grand nombre de classes disparates, qui n'ont pas d'autres points communs que d'utiliser le même mécanisme.

2. *Représentation horizontale* : une table par classe concrète
   Dans cette approche (cf. (2) dans la figure 1.6), une table est créée pour chaque classe concrète. Tous les attributs de la classe concrète et ceux hérités des super-classes de cette dernière constituent les colonnes de la table. A ces colonnes, s'ajoute une clé primaire. Lorsqu'il n'y a qu'un seul niveau de classes concrètes, l'avantage de cette approche est qu'il est facile d'obtenir et de stocker les informations sur les objets existants car toutes les informations sur un objet se retrouvent dans une seule table.

   Les inconvénients principaux de cette représentation sont (1) qu'une requête générique sur une classe abstraite nécessite des unions de projections des tables des sous-classes et (2) que le polymorphisme n'est pas supporté de façon aisée car un lien vers une classe abstraite se matérialise par une référence vers des tables différentes selon les instances. De complexes mécanismes d'"aiguillage" (cf. section 3.4.1.2) sont alors nécessaires.

3. *Représentation verticale* : une table par classe
   Dans cette approche (cf. (3) dans la figure 1.6), on crée une table pour chaque classe. Chaque table a pour colonnes les attributs définis au niveau de la classe qu'elle représente. Un même identifiant est utilisé comme clé primaire pour toutes les tables. Au niveau des sous-classes, il représente à la fois une clé primaire et une clé étrangère.

L'avantage de cette approche est qu'elle représente de façon très adéquate les concepts orientés objets. Le polymorphisme peut se représenter assez aisément par une clé étrangère unique (avec jointure automatique avec les super-classes) parce que nous avons un enregistrement dans la table correspondant à chaque classe auquel un objet appartient. Il est facile également de modifier les super-classes et d'ajouter de nouvelles sous-classes car cela correspond respectivement à la modification et à l'ajout d'une table. Outre le nombre important de tables dans la base de données (une pour chaque classe), le principal inconvénient de cette approche est que les informations sur un objet d'une classe se retrouvent reparties entre grand nombre de tables. Ainsi, tout accès à un objet nécessite de faire de nombreuses jointures, ce qui peut devenir très coûteux.

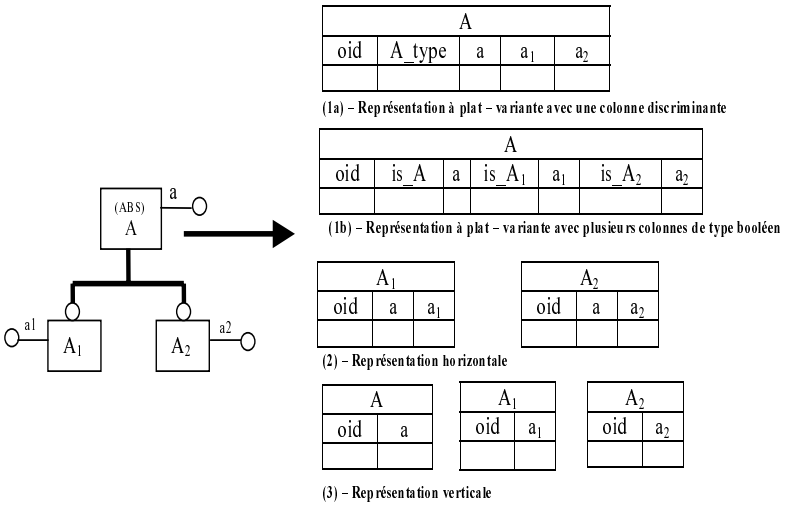


FIG. 1.6 – Différentes approches de représentation de l'héritage

3.4.1.2 Représentation des relations d'association, d'agrégation et de composition

Pour ce qui concerne les associations, agrégations et compositions, on trouve également dans la littérature, plusieurs propositions de représentation [54, 137, 136, 138, 139, 140]. Ces propositions sont définies en fonction des cardinalités des relations ($1:1$, $1:n$, $n:m$) et des propriétés des relations d'agrégations (dépendance, partage, exclusivité, prédominance) [105]. [145] énumère toutes les solutions envisageables pour la représentation de ces relations en relationnel. Ces différentes solutions se résument soit à la création de tables intermédiaires entre les tables des classes participantes à la relation, soit à la déclaration de clés étrangères sur des colonnes de tables. Ces différentes propositions ne s'appliquent que lorsque le choix de représentation de l'héritage est vertical ou à plat. Lorsque seules les classes concrètes sont représentées, il est nécessaire de représenter tout lien par un aiguillage comportant en particulier une colonne qui indique le type effectif de l'instance référencée.

3.4.2 Choix de règles de transposition

Chaque représentation possible de chaque mécanisme objet présente des avantages et des inconvénients. Ceci fait que la meilleure représentation dans chaque cas dépend étroitement des caractéristiques locales du modèle objet à transposer. Trois solutions sont alors envisageables :

- Choisir pour chaque classe la représentation spécifique adaptée et programmer spécifiquement pour chaque classe les accès à ses objets persistants. Cette méthode est extrêmement coûteuse et peut difficilement être envisagée lorsque, comme c'était notre cas, plusieurs centaines de classes doivent être transposées.

- Utiliser des méta-règles permettant dans chaque contexte de choisir les règles à appliquer, et de générer automatiquement les représentations relationnelles et les programmes de transformation. C'est ce que nous avons fait pour représenter le modèle d'ontologie PLIB et lui associer une API adaptée.

- Utiliser un outil paramétrable pour la gestion des transformations permettant une définition déclarative des règles à utiliser pour chaque classe. Nous utiliserons également une telle approche lorsque nous souhaiterons développer une API objet plus simple que le modèle objet initial [128].

Nous présentons ci-dessus un exemple d'un tel outil, qui est précisément celui que nous avons utilisé.

3.4.3 Outil de gestion des transpositions relationnel-objet : l'exemple d'Hibernate

Hibernate est une couche logicielle de transposition objet/relationnel pour les environnements Java et .NET. Hibernate permet de libérer les programmeurs de la plus grosse partie de la programmation liée à la persistance des données sous forme d'objets dans les bases de données. Les classes (Java ou .NET) et les tables sont mises en correspondance grâce à un langage de mapping dont la syntaxe est en XML. Hibernate génère alors l'implémentation des méthodes des différentes classes par accès à la base de données. Il permet également de faire des requêtes dans le langage HQL qui porte directement sur le modèle objet.

Un des avantages d'Hibernate est que le langage de mise en correspondance objet/relationnel qu'il propose est paramétrable. En effet, Hibernate autorise de spécifier pour chacun des principaux concepts objets (classe, héritage, association, composition, etc.) et pour chacune des classes la stratégie la plus adaptée pour la représentation des données dans les bases de données. Un document XML de correspondance est exploité par Hibernate pour (1) connecter les sélecteurs des classes aux colonnes correspondant aux attributs dans la base de données, (2) gérer les accès concurrents aux objets en mémoire en assurant la synchronisation des objets en mémoire avec la base de données et (3) pour générer éventuellement le schéma de la base de données et/ou les classes si elles n'existent pas encore.

L'utilisation de la couche Hibernate est relativement simple. Le programmeur définit dans

```
public class Personne{
Long id;
String nom;
public Personne (){}
public String getNom(){
    return this.nom;
}
public void  setNom(String valeur){
    this.nom= valeur;
}
…
}
```

```
public class Salarie extends Personne{
float salaire;
public float getSalaire(){
    return this.salaire;
}
public void setSalaire(float val){
    this.salaire=val;
}
…
}
```

A

```
<hibernate-mapping>
  <class name="Personne" table="personne">
    <id name="id" column="ID"/>
    <property name="nom" column="nom"/>
  </class>
  <subclass name="salarie" table="salarie" extends="Personne">
    <property name="salaire" column="salaire"/>
  </subclass>
</hibernate-mapping>
```

B

FIG. 1.7 – Exemple de classes Java et de mapping Hibernate

un premier temps en Java les classes de son application. Ensuite, il définit la correspondance entre le schéma de la base de données et les classes qu'il a définies. Hibernate offre la possibilité de générer à la fois les méthodes de classe et le schéma de la base de données à partir de cette correspondance. Nous illustrons l'utilisation d'Hibernate à travers l'exemple des figures 1.7 et 1.8. Le modèle objet de l'exemple est constitué de deux (2) classes (*Personne* et *Salarié*). La classe *Salarié* hérite de *Personne* (cf. figure 1.7A) et définit localement l'attribut *salaire*. On peut constater que les classes ne définissent que des sélecteurs pour chacun de leurs attributs. Hibernate recommande que toutes les classes disposent d'un attribut (de type entier) qui servira d'identifiant pour ces objets de la classe (l'attribut *id* dans notre exemple). La figure 1.7B montre un exemple de fichier XML de configuration pour la connexion des classes et leurs attributs à leurs tables et colonnes respectives dans la base de données. Dans notre exemple, la stratégie de représentation de l'héritage utilisée entre les classes *Personne* et *Salarié* est la "représentation à plat" ou "table par hiérarchie". Enfin la figure 1.8 présente un petit programme utilisant Hibernate.

Nous verrons dans le chapitre 4, comment nous avons utilisé ce système pour la mise en œuvre de notre architecture de base de données à base ontologique.

Notons, pour clore cette section, que la définition des modèles logiques relationnel ou relationnel-objet permettant de représenter des modèles conceptuels objet, nécessite de transformer les modèles. Cette opération de transformation peut se faire soit manuellement, soit automatiquement à partir de programmes. Les techniques utilisées pour réaliser ces transformations sont appelées In-

```
// Créer une session Hibernate
Configuration cfg = new Configuration().addResource("monmapping.xml") ;// lecture du mapping
SessionFactory sessionFactory = cfg.buildSessionFactory();
Session session = sessionFactory.openSession(); // ouverture d'une session Hibernate.
//Créer une instance de classe Salarié
    Salaire unSalarié = new Salarie(); // création d'une instance
    unSalarie.setNom("DEHAINSALA");
    session.flush(unSalarié); // Ajout de l'objet dans la BD
//Récupérer un Salarié stocké dans la base de données
   Salaire unSalarie = (Salarie) session.load(Salarie.class, Salarie  Id);
```

FIG. 1.8 – Exemple d'utilisation d'Hibernate

génierie Dirigé par les Modèles (IDM). Pour la mise en œuvre du prototype de notre architecture de base de données, nous avons largement utilisé les techniques de l'IDM pour l'implémentation de chacune de ses composants [46, 128].

Dans la section suivante, nous présentons brièvement l'IDM que nous illustrons avec le langage EXPRESS.

3.5 Transformation de modèle et l'Ingénierie Dirigée par les Modèles

L'ingénierie des systèmes informatiques se base toujours sur des modèles. Pendant longtemps, les modèles ont seulement constitué une aide pour les utilisateurs humains, permettant de développer manuellement le code final des applications informatiques. L'approche de l'Ingénierie Dirigée par les Modèles (IDM) consiste à représenter les modèles comme instances d'un méta-modèle à traiter par programme les données ainsi constituées, et à les utiliser pour générer le code final des applications. Le MDA [10, 7] (Modèle Driven Architecture) de l'OMG est l'approche la plus connue d'ingénierie dirigée par les modèles. Le MDA est basé sur le standard UML pour la définition des modèles et sur l'environnement de méta-modélisation MOF (Meta-Object Facility) [63] pour la programmation au niveau modèle et la génération de code. Basé sur un ensemble de langages différents (UML, XML, OCL), MDA est une architecture lourde et difficile à maîtriser.

Nous présentons dans la section suivante l'environnement d'IDM EXPRESS. Construit autour du langage EXPRESS [79, 143], cet environnement est au moins aussi ancien et mature que l'environnement basé sur les standards de l'OMG (UML, OCL, MOF, MDA et XMI). Il a un domaine d'application plus restreint, puisqu'il ne traite que les modèles de données et non les modèles de systèmes. Il est, par contre, beaucoup plus simple à mettre en œuvre car le même langage, EXPRESS, peut être utilisé pour les différentes tâches impliquées dans une activité d'IDM : modélisation, méta-modélisation, instanciation et transformation de modèles. Nous exploiterons largement dans le chapitre 4, cet environnement pour le développement, par génération de code, du système de base de données à base ontologique réalisé au cours de notre thèse. Gérant de façon uniforme les données et les modèles, le système ainsi développé permettra également de montrer l'intérêt des concepts clés de l'IDM, de la méta modélisation et des actions au niveau

modèle, lorsqu'ils sont transposés au niveau de l'utilisateur final, qu'il soit utilisateur interactif ou programmeur d'application.

En EXPRESS, deux environnements existent pour faire de l'IDM :
- transposition de modèle : EXPRESS-X,
- transformation de modèles : Programmation EXPRESS.

Nous les présentons dans les deux sections suivantes.

3.5.1 Transposition de modèle : EXPRESS-X

EXPRESS-X est un complément déclaratif du langage EXPRESS (ISO 10303-14) [2] dont l'objectif est de permettre une spécification déclarative des relations de correspondances (*mapping*) existant entre des entités de différents schémas EXPRESS. Ce langage supporte deux types de constructions :

1. SCHEMA_VIEW qui permet de déclarer des *vues* sur les données d'un schéma EXPRESS ;
2. SCHEMA_MAP qui permet de déclarer des règles de transformation (*"mapping"*) entre des entités ou des vues d'un (ou plusieurs) schéma(s) EXPRESS source(s) vers un (ou plusieurs) schéma(s) EXPRESS cible(s).

Dans l'exemple de la figure 1.9, nous déclarons un mapping pour faire migrer les instances du schéma de la figure 1.3 en des instances du schéma *person* (première colonne du tableau). La deuxième colonne du tableau montre la déclaration du mapping. Il transforme les instances de l'entité *salarié* en des instances d'entités *employee* de la manière suivante :
- le *noSS* de *salarié* correspond à l'attribut *id* de l'entité *employee* ;
- l'attribut *name* de l'entité *employee* est la concaténation des attributs *nom* et *prénom* de l'entité *salarié*.
- l'attribut *good_salary* de type BOOLEAN est dépendant de l'attribut *salaire* de l'entité *salarié*. La valeur de l'attribut est VRAI si la valeur de l'attribut *salaire* est supérieur à 1999 et FAUX dans le cas contraire.

La dernière colonne du tableau montre l'application du mapping sur deux instances de l'entité *salarié*. Si ce langage est essentiellement déclaratif, on peut néanmoins utiliser toute la puissance du langage procédural pour calculer des attributs (dérivés) du schéma source destinés a être transposés dans le schéma cible.

3.5.2 Programmation EXPRESS et transformation de modèles

Les environnements de modélisation EXPRESS permettent soit de coder un programme dans un langage classique (C, C++, JAVA) en utilisant des interfaces normalisées pour accéder aux données (niveau modèle et niveau instance) associées à un modèle EXPRESS, soit de coder un programme directement dans le langage procédural associé à EXPRESS. Si la première approche est adaptée pour certains types de programmes, elle présente l'inconvénient de nécessiter la connaissance à la fois d'EXPRESS et d'un langage de programmation. De plus, l'intégration d'EXPRESS dans un langage de programmation pose les problèmes usuels qu'impliquent

Schéma cible	Schéma de mapping	Instances de données

SCHEMA Person ;
ENTITY Employee;
id : **NUMBER**;
name : **STRING** ;
good_salary : **BOOLEAN**;
UNIQUE url : id;
END_ENTITY ;
END_SCHEMA
	SCHEMA_MAP mapping_exemple;
	REFERENCE FROM Universitaire **AS SOURCE**;
	REFERENCE FROM Person **AS TARGET**;
	MAP U_Salarié_map **AS** emp : Employee;
	FROM sal : Salarié;
	SELECT
	emp.id := sal.noSS;
	emp.name := sal.nom_prenom;
	emp.good_salary := is_good(sal.salaire);
	END_MAP;
	FUNCTION is_good (salaire : **REAL**) : **BOOLEAN**;
	IF salaire > 1999 **THEN RETURN** ***TRUE***
	ELSE RETURN ***FALSE***;
	END_IF;
	END_FUNCTION;
	END_SCHEMA;
		(*Instances du schéma de source*)
		#2 = SALAIRE(13457,'Jean', 'Hondjack' , 27, #3, 1000);
		#3 = SALAIRE(23215,'Jean', 'Nathalie', 25, #2, 2500);
		
		(*Résultats de mapping *)
		#502 = EMPLOYEE(13457, 'Jean Hondjack', 'FALSE');
		#503 = EMPLOYEE(23215, 'Jean Nathalie', 'TRUE');

FIG. 1.9 – Exemple de transformation de modèle avec EXPRESS-X.

les conversions de types (impédance mismatch) . La seconde permet, par contre, une prise en main rapide qui permet d'exploiter un modèle EXPRESS dans un environnement homogène de développement. Le langage procédural associé à EXPRESS n'est pas complet. Pour en faire un langage complet de programmation, il ne lui manque que deux fonctions :

- les primitives d'entrée/sortie (read/write) ;
- la primitive ("run") permettant de demander l'exécution d'une procédure.

Tous les environnements existants l'enrichissent de ces deux primitives pour en faire un langage de programmation à part entière permettant de réaliser un environnement simple à mettre en œuvre. Ce langage permet alors de réaliser à la fois :

- le développement d'un modèle conceptuel ;
- la spécification de l'ensemble des contraintes d'intégrités ;
- la manipulation et les transformations de modèles et d'instances.

3.5.3 Environnements d'IDM en EXPRESS

Différents environnements existent. Certains permettent de rendre persistante les instances créées. Par exemple l'environnement ECCO [147] génère à partir de la compilation de modèle EXPRESS une application C++ qui représente automatiquement toutes les instances EXPRESS sous forme d'instances C++, et toutes les contraintes sous forme de méthodes C++. Le schéma est lui-même, représenté sous forme d'instances C++ d'un méta-schéma. Cette application, qui comporte en particulier les fonctions de lecture/écriture de fichiers physiques, peut être exploitée par une interface graphique, par une API utilisable en C, C++, et Java et directement en EXPRESS par une application compilée avec le modèle. Une fois les traitements effectués, les instances doivent être sauvegardées sous forme d'un fichier physique pour être conservées.

D'autres environnements, tel que EXPRESS Data Manager (EPM Technology), sont liés à une base de données. Toute compilation d'un schéma EXPRESS génère automatiquement le schéma de base de données correspondant, les contraintes assurant l'intégrité de la base et l'interface

SDAI permettant d'accéder à la fois aux schémas et aux instances.

4 Limites des méthodes classiques de conception des bases de données

A l'issue du bref état de l'art de l'art présenté jusque là sur les méthodes actuelles de conception de bases de données, il nous parait important de souligner deux difficultés majeures auxquelles elles donnent lieu. Ceci nous permettra alors de fonder la principale proposition que nous développerons dans cette thèse, à savoir représenter explicitement la sémantique des données dans les bases de données.

4.1 Les problèmes

Dans la section 1.2, nous avons vu que la conception de bases de données était un processus qui se déroulait en trois étapes allant (1) de la conception du modèle conceptuel, (2) à l'élaboration du modèle logique et enfin (3) à la définition de *vues* utilisateur. La mise en œuvre de ce processus présente, en particulier dans le contexte d'évolution des formalismes de modélisation de l'information et de gestion des données vus aux sections 1 et 3, deux inconvénients de plus en plus significatifs :

1. **l'écart existant en général entre les modèles conceptuels et les modèles logiques des données**, qui s'accroit avec la divergence des formalismes.
2. **la très forte dépendance des modèles vis-à-vis des concepteurs et des besoins applicatifs particuliers**, ce qui rend difficile les activités d'intégration indispensables dans les sociétés modernes d'information et de communication.

4.1.1 L'écart entre les modèles conceptuels et les modèles logiques des données

L'un des objectifs essentiel de la modélisation conceptuelle est de permettre le développement d'un modèle logique de données qui permettra la création d'une base de données. Le passage du modèle conceptuel au modèle logique, nécessite, comme nous l'avons vu, des opérations de traduction complexes. Le modèle logique résultant de cette traduction est alors très différent du modèle conceptuel initial. Les entités et les associations sont éclatées en de multiples tables (dans le cas des bases de données relationnelles). Le modèle logique résultant peut, très vite devenir incompréhensible (surtout pour des grands modèles conceptuels conséquents), pour un utilisateur et ne permet plus alors d'appréhender le problème traité. Ceci impose, en général, de recoder dans tous les différents applicatifs d'accès, l'ensemble des concepts existants initialement dans le modèle conceptuel. Même si des techniques d'Ingénierie Dirigée par les Modèles, ou les outils tels que PowerAMC de Sybase ou Hibernate [70] peuvent aider dans ce travail, il s'agit toujours de développement très important, coûteux et difficile à maintenir.

4.1.2 Forte dépendance des modèles vis à vis des concepteurs et des objectifs applicatifs

Comme nous l'avons déjà mentionné, le modèle conceptuel d'un univers quelconque est spécifique des questions très précises auxquelles on souhaite que le système puisse répondre. En d'autres termes, le modèle conceptuel dépend étroitement de l'objectif applicatif du système en cours de développement. Cela aboutit donc à des modèles conceptuels qui seront toujours différents, même s'ils adressent essentiellement le même domaine et ce, sans qu'il soit possible d'identifier par des moyens automatiques, ceux des concepts qui se correspondent, et ceux qui sont effectivement différents.

Les bases de données qui résulteront de ces différents modèles conceptuels seront alors forcement hétérogènes et occasionneront de nombreux problèmes lors de toute tentative d'intégration future des données contenues dans ces bases de données [18, 16, 20, 27, 92, 162]. Kim et al. [93] présente les différentes incohérences, sous le nom de conflits, pouvant exister entre sources de données résultant de deux modèles conceptuels différents d'un même domaine. Les principaux conflits sont les suivants.

1. Les *conflits de nommage* interviennent lorsqu'un même concept C de deux modèles conceptuels est nommé différemment. Par exemple : dans un MC, on peut rencontrer le concept de *salarié* et le concept de *employé* dans un autre.
2. Les *conflits de subsomption* intervient lorsqu'une hiérarchie de classes est différente d'un modèle à un autre. Par exemple : dans MC_1, la classe *élève* est subsumée la classe *personne* et dans MC_2, *élève* est subsumé *étudiant*.
3. Les *conflits de structure* interviennent lorsqu'une classe n'est pas décrite par les mêmes propriétés d'un MC à un autre. Par exemple dans MC_1, *personne* est décrite par les propriétés : *id*, *nom*, *prénom*, *adresse* et dans MC_2 par *id*, *nom*, *prénom*, *téléphone*, *télécopie*.
4. Les *conflits de granularité d'un concept* interviennent qu'un concept C est décrit comme une classe dans un modèle conceptuel et comme un attribut dans un deuxième modèle. Par exemple **adresse** dans MC_1 est décrite comme une classe et dans MC_2 comme une propriété de la classe *personne* (*id*, *nom*, **adresse**)
5. Les *conflits de type de propriétés* intervient lorsque une propriété P possède des domaines différents entre deux modèles conceptuels. Par exemple, la propriété *sexe* dans MC_1 a pour type STRING ("F" = féminin et "M" = masculin) et dans MC_2 a pour type ENTIER (0 = féminin et 1 = masculin).
6. Les *conflits de granularité d'attributs* interviennent lorsqu'un attribut est décrit de façon atomique dans un modèle conceptuel et composé dans un autre. Par exemple, dans la classe *personne* de MC_1, on a deux propriétés distinctes *nom* et *prénom* et dans MC_2 une propriété *nom_prénom* qui compose le *nom* et le *prénom*.
7. Les *conflits d'unité de mesures* interviennent lorsque les valeurs des propriétés sont dans des unités différentes (décrites formellement ou de façon informelle suivant la richesse du formalisme de représentation) d'un modèle à un autre. Par exemple la mesure de la propriété *salaire* de la classe *salarié* est en euro et dans MC_2 la mesure est en CFA[2].

[2]franc de la Communauté Financière Africaine

8. Les *conflits d'identificateurs d'objets* interviennent lorsque l'identifiant d'un classe (qui peut être un ensemble de propriétés) est différent d'un modèle à un autre.
9. Les *conflits de contraintes d'intégrités* interviennent lorsqu'une propriété identique dans deux modèles différents est associée à des contraintes d'intégrités différentes. Par exemple le *salaire* d'un salarié dans le modèle MC_1 est inférieur à 10000(implicitement exprimé en euros) et dans le modèle MC_2 est inférieur 1000000.
10. Les *conflits de contextes* interviennent lorsqu'un concept est perçu différemment d'un modèle conceptuel à un autre. Cette différence est souvent dû à des objectifs légèrement différents des systèmes cibles. Par exemple le concept *prix* est considéré dans le modèle MC_1 selon le point de vue TTC et dans le modèle MC_2, il est vue selon l'aspect HT.

Les conflits évoqués ci-dessus résultent du fait qu'un modèle conceptuel est seulement considéré comme une étape intermédiaire dans la définition d'une base de données, elle-même structurée en fonctions des objectifs applicatifs du système cible. Le modèle conceptuel est donc crée de façon isolé. Il contient exclusivement ce qu'il apparaît pertinent au concepteur de représenter effectivement. Cette représentation est effectuée selon une structure qui dépend beaucoup des spécificités du concepteur. Deux modèles conceptuels conçus par deux équipes différentes pour deux systèmes visant à remplir la même fonction dans le même domaine seront donc toujours :
- partiellement différents du point de vue du domaine exact modélisé,
- très différents du point de vue de la structure du modèle résultant.

Les modèles logiques seront alors encore plus différents, et nécessiteront un énorme travail manuel d'intégration sémantique, chaque fois qu'il faudra intégrer deux de ces bases de données.

4.2 Une solution : représenter la sémantique des données

Ces deux problèmes ci-dessus résultent de deux conséquences des méthodes traditionnelles de conception des bases de données.
(1) Les modèles conceptuels ne sont plus accessibles lorsque la base de données est achevée.
(2) Chaque modèle conceptuel est spécifique à un projet de conception d'une base de données.

Pour éviter la première conséquence, il est alors apparu intéressant de donner plus de place et d'autonomie à la notion de modèle conceptuel pour permettre sa représentation effective dans une base de données. Différents travaux menés au cours des années 1990, ont visé à atteindre cet objectif [153, 110]. Cela permettrait de faire abstraction du modèle logique au moyen de programmes qui accédaient directement aux modèles conceptuels. Pour atteindre cet objectif, il était nécessaire de :
- trouver une représentation des modèles conceptuels dans la base de données pour qu'ils puissent être accessibles,
- d'établir une liaison entre le modèle logique et le modèle conceptuel représenté dans la base de données.

Aucune représentation formelle des modèles conceptuels n'étant alors disponible, ces travaux n'ont pas entraîné de réelle évolution des pratiques de conception. Quoi qu'il en soit, ils n'auraient pas permis de résoudre les problèmes d'intégration qui nécessitent qu'en plus, les modèles conceptuels soient aussi, pour leurs parties communes consensuelles.

Ce sont ces deux difficultés qui vont être levées par les ontologies. En effet, une ontologie définit de façon *formelle*, *explicite* et *consensuelle* l'ensemble des concepts partagés d'un domaine.

- *Formel* signifie qu'elle est définie dans un langage traitable par machine. Les ontologies seront donc représentables.
- *Consensuelle* signifie qu'elle est acceptée par les membres d'une certaine communauté. Tous les concepts partagés pourront donc être modélisés à l'identique par tous les membres de cette communauté, qui peut être, avec les ontologies normalisées, l'ensemble d'un secteur professionnel.
- *Explicite* signifie que le domaine modélisé est décrit indépendamment d'un point de vue ou d'un contexte implicite. Plusieurs points de vues, légèrement différents pourront donc partagés les mêmes concepts.

Un modèle conceptuel peut être représenté comme un sous-ensemble (fragment) d'une ontologie. C'est la raison pour laquelle nous proposons dans cette thèse :

1. De promouvoir le développement d'ontologies partagées (OP) de domaine, au moins dans les domaines où les problèmes d'intégration de bases de données peuvent être anticipés (étape 1 figure 1.10).
2. De précéder de développement d'un modèle conceptuel (MC) par le développement d'une ontologie locale (OL) utilisant un formalisme standard de modélisation d'ontologie et conçue, si elle existe, par l'emprunt d'une ontologie (ou plusieurs) ontologies partagées (étape 2 figure 1.10).
3. De représenter un modèle conceptuel comme un sous-ensemble de l'ontologie locale (étape 3 figure 1.10).
4. De représenter l'ontologie locale, le modèle conceptuel et les données au sein de la même base de données (étapes 4, 5, 6 figure 1.10).

Nous appelons ce type de bases de données des *bases de données à base ontologique*, et nous proposons dans ce travail un modèle d'architecture appelé *OntoDB*, qui montre la faisabilité d'une telle implémentation.

5 Conclusion

Nous avons vu dans ce chapitre les différentes approches de modélisation de l'information et de leurs représentations dans des bases de données. Concernant la modélisation de l'information, deux grandes approches sont actuellement utilisées dans la littérature : l'approche entité-association et l'approche objet. L'approche objet, qui se généralise aujourd'hui, consiste à hiérarchiser des catégories par des relations de *subsomption* associées à un mécanisme d'héritage. Les

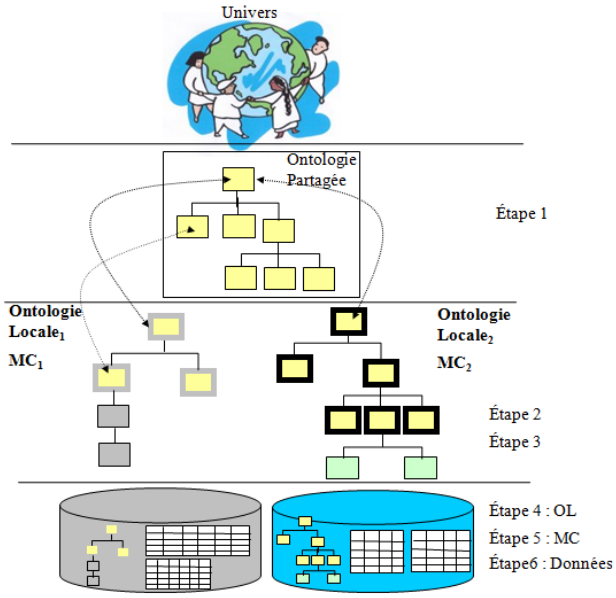


FIG. 1.10 – Methodologie de conception de bases de données à l'aide d'ontologie

classes sont caractérisées par des propriétés qui peuvent être de types complexes. Elles peuvent également être associées à de la connaissance procédurale exprimée sous forme de contraintes, de règles et/ou de méthodes. Ces différents mécanismes permettent de modéliser de façon de plus en plus fine la connaissance d'un domaine.

Concernant la persistance des données, essentiellement deux approches existent : le modèle relationnel et le modèle objet. Le modèle relationnel, basé sur l'algèbre relationnelle, est un système efficace, simple, mature et normalisé. Il bénéficie d'excellentes performances résultant de nombreuses années d'expériences, de fondements mathématiques solides et d'un grand succès commercial. Le modèle de base de données objet, apparu suite au succès de la modélisation objet, permet de rendre persistants des objets. Les systèmes de bases de données objets n'ont pas connu le même succès que la modélisation objet. Les systèmes de bases de données relationnelles semblent avoir repris le dessus grâce à leur performance et à leur maturité industrielle, même s'ils s'ouvrent progressivement, et en ordre dispersé, à l'introduction de quelques concepts objets dans le relationnel à travers l'implantation progressive de la norme SQL99.

L'évolution récente aboutit donc à une divergence entre les méthodes de représentation de modélisation, essentiellement objet et les méthodes de représentation des données persistantes, essentiellement relationnelles. Une méthode très fréquente pour tenir compte de cette divergence consiste alors à définir une ensemble de règles de correspondances entre les concepts de l'orienté objet et les concepts du relationnel, puis à convertir automatiquement les données et les modèles d'un univers à l'autre, soit en utilisant les techniques d'IDM, soit en utilisant des outils de transformation, appelés gestionnaires d'objets.

Cette approche, qui tend à s'imposer actuellement, présente néanmoins deux inconvénients.

- La première difficulté est la distance importante existant entre les modèles conceptuels, objets, et les modèles logiques sur lesquels se base la représentation des données. Seuls ces derniers étant représentés au sein de la base de données, cela entraîne de grosses difficultés pour les programmeurs qui accèdent aux bases de données et pour les utilisateurs qui ne comprennent pas le modèle effectivement implémenté. Il est alors impossible d'offrir des accès au niveau connaissance, c'est-à-dire au niveau du modèle conceptuel, sauf à ré-coder à l'extérieur de la base de données, des applications d'accès qui incorporent les modèles conceptuels.
- La deuxième difficulté est liée au fait que les modèles conceptuels dépendent étroitement du point de vue, ou du contexte applicatif, pour lesquels ils ont été élaborés. Cela est à l'origine de l'hétérogénéité importante entre bases de données portant sur le même domaine et des difficultés considérable d'intégration ultérieure de telles bases de données.

C'est à ces deux problèmes que va s'attaquer la nouvelle approche de modélisation que nous proposons et qui vise à utiliser des ontologies dans le processus de conception de bases de données. En proposant de baser tout modèle conceptuel sur une ontologie formelle locale et de représenter le modèle conceptuel au sein de la base de données, elle permet d'offrir un accès au niveau connaissance tant aux programmeurs d'application qu'aux utilisateurs interactifs. En recommandant de baser les ontologies locales sur les ontologies partagées de domaine, elle permet d'aller vers une intégration automatique des différentes bases de données d'un domaine.

Dans le chapitre suivant, nous discutons de façon plus détaillée de la modélisation à base ontologique et de son utilisation pour la conception de bases de données.

Chapitre 2

Modélisation et gestion de données à base ontologique

Introduction

Les ontologies occupent désormais une place importante dans la recherche en informatique et sont utilisées ou proposées à l'utilisation, dans de nombreux domaines (intégration de données, Web sémantique, Web services, E-commerce, bases de données, etc.). Le caractère formel et consensuel d'une ontologie permet de la diffuser dans une communauté et contribue à pouvoir rendre interopérable les applications en représentant explicitement la sémantique des données.

L'utilisation croissante d'ontologies dans divers domaines (techniques ou documentaires) fait qu'aujourd'hui, certaines données sont déjà représentées comme des instances de classes d'ontologies et ont donc leur sémantique définie à partir d'ontologies. Nous appellerons ces données des *données à base ontologique*. Le besoin de représenter les ontologies et les données à base ontologique dans de vraies bases de données plutôt qu'en mémoire centrale ou dans des fichiers ordinaires se fait de plus en plus sentir. Ces besoins sont à l'origine de l'émergence d'une nouvelle approche de bases de données dites *bases de données à base ontologique (BDBO)*.

L'objectif de ce chapitre est de faire un état de l'art à la fois sur les ontologies et sur la représentation et la gestion des données à base ontologique.

Le chapitre s'organise comme suit. Dans la section suivante, nous discutons sur la notion d'ontologie, son origine et les principales définitions qu'on trouve dans la littérature. Nous y faisons également une comparaison entre ontologie et modèle conceptuel pour préciser comment les ontologies peuvent être utiles dans une perspective de conception de bases de données. Dans la section 2, nous présentons les principaux composants des ontologies, et décrivons brièvement quelques langages actuels de définitions d'ontologies. Dans la section 3, nous introduisons la notion de base de données à base ontologique. Nous décrivons également dans cette section les principales approches de représentation des ontologies et des données à base ontologies proposées dans la littérature puis quelques uns des principaux systèmes mettant en œuvre ces représentations. Cette analyse nous permet, en conclusion, d'identifier les insuffisances des approches

existantes, que le modèle d'architecture proposé dans le cadre de cette thèse visera précisément à combler.

1 Notion d'ontologie

Après avoir rappelé l'origine de la notion d'ontologie, nous discutons, dans la partie 2, de quelques définitions qu'on trouve dans la littérature. Dans la partie 3, nous faisons une comparaison entre ontologies et les modèles conceptuels. Dans la partie 4, nous faisons une comparaison entre ontologies conceptuelles et ontologies linguistiques.

1.1 Origine historique

Le terme "*ontologie*" vient de la philosophie. Étymologiquement en grec ancien, le terme *ontologie* signifie : ontos - être et logos - science, discours ou études. En d'autres termes, l'ontologie est la science ou de la théorie de l'être [111]. Cette discipline a été inventée par les grecs, même si, en fait, ce n'est qu'au XVIIe siècle que le terme ontologie a été créé pour désigner cette approche.

Dans la philosophie d'Aristote, l'ontologie est la science de l'être en tant qu'être, indépendamment de ses manifestations particulières. Il s'agit de capturer l'essence d'un être indépendamment de son existence. Dans les écrits récents, le terme ontologie recouvre deux usages : le premier pour désigner la discipline philosophique et le second pour désigner son utilisation en informatique. Nous nous cantonnerons désormais à l'utilisation informatique.

McCarty fut le précurseur de l'usage du terme en informatique. McCarty affirmait que les concepteurs des systèmes d'intelligence artificielle devaient avant tout modéliser les aspects essentiels des objets du domaine d'études, en d'autres termes de construire une ontologie de leur domaine, et ensuite seulement baser leurs systèmes sur cette ontologie [161]. Les ontologies sont aujourd'hui au cœur de la plupart des systèmes de représentation de connaissances. Elles visent à fournir des représentations à travers lesquelles les machines peuvent représenter explicitement la sémantique des informations.

1.2 Définition des ontologies en informatique

Définition du Net[3]
Le dictionnaire du Net définit une ontologie comme étant : une "organisation hiérarchique de la connaissance sur un ensemble d'objets par leur regroupement en sous catégories suivant leurs caractéristiques essentielles" [1].

Définition de Gruber [64]
La définition admise et utilisée par beaucoup d'auteurs est celle de Gruber : "An ontology is an explicit spécification of a conceptualization".

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

Définition de Guarino [65]
Guarino affine la définition de Gruber en disant qu' "une ontologie est définie comme étant une spécification formelle d'une conceptualisation partagée".

Définition de Fendel [56]
Fensel propose dans Fensel et al., la définition suivante "formal and explicit specifications of certain domains and are shared between large groups of stakeholders".

Pour Gruber, Guarino et Fensel :

1. *spécification formelle* signifie que les ontologies sont basées sur des théories formelles qui permettent à une machine de vérifier automatiquement certaines propriétés de consistance et/ou de faire certains raisonnements automatiques sur les ontologies et leurs instances.
2. *explicite* signifie que les catégories de concepts du domaine et leurs différentes propriétés sont décrites explicitement.
3. *partagée* signifie qu'une ontologie capture une connaissance *consensuelle*, i.e., admise par tous les experts d'une communauté.

Définition de Jean et al. [87]
Jean et al. [87], partant des critères énumérés précédemment, définissent une ontologie de domaine comme étant "a formal and consensual dictionary of categories and properties of entities of a domain and the relationships that hold among them". Cette définition caractérise les ontologies de domaine auxquelles nous nous intéressons dans notre étude. Pour Jean et al., *"entities"* désigne n'importe quel élément ayant une existence (matérielle ou immatérielle) dans le domaine à conceptualiser. Le terme *"dictionary"* souligne le fait que toutes entités et propriétés d'entités décrites dans une ontologie de domaine peuvent être référencées directement, au moyen de symboles identifiants, pour en permettre l'usage dans n'importe quel modèle et dans n'importe quel contexte.

Pour préciser le concept d'ontologie dans une perspective de bases de données et pour préciser ce qu'elles peuvent apporter par rapport aux limites actuelles des modèles conceptuels, nous faisons dans la section suivante une comparaison entre ces deux types de conceptualisation.

1.3 Comparaison entre ontologie et modèle de conceptuel

Les ontologies et les modèles conceptuels présentent à la fois des similitudes et des différences [87, 146].

1.3.1 Similitudes

Comme les modèles conceptuels (MC), les ontologies conceptualisent également l'univers du discours au moyen de classes associées à des propriétés et hiérarchisées par *subsomption*. Les principes de bases de la modélisation sont similaires.

1.3.2 Différences

On peut identifier cinq différences majeures entre ces deux types de modèles.

- Une première différence se situe dans l'objectif de la modélisation. Les MCs *prescrivent* l'information qu'ils représentent dans un système informatique particulier. Par exemple, dans un MC_1 destiné pour un système médical, on trouvera nécessaire de décrire une personne par son groupe sanguin, son poids, etc. et dans un MC_2 pour un autre système médical, on ne devra pas représenter le poids mais l'antécédent familial *devra* être représenté. Au contraire, les ontologies *décrivent* les concepts d'un domaine indépendamment de toutes applications et systèmes informatiques particuliers dans lesquels l'ontologie pourrait être utilisée. Dans l'exemple précédent, mais en restant toujours dans un contexte d'une ontologie des patients médicaux, une personne sera caractérisée (directement ou dans une de ses sous-classes) par toutes les propriétés de MC_1 et MC_2 et probablement bien d'autres. Chaque système particulier pourra alors puiser celles des propriétés qui sont pertinentes.

- Une deuxième différence est que les classes et les propriétés définies dans les ontologies sont associées à des *identifiants*, ce qui leur permet d'être référencées à partir de n'importe quel format ou modèle indépendamment de leur structure. Au contraire, la conceptualisation effectuée dans un MC ne peut pas être réutilisée à l'extérieur et indépendamment de ce MC.

- Une troisième différence est liée au caractère *formel* des ontologies. Ce caractère formel permet d'appliquer des opérations de raisonnement sur les ontologies soit pour vérifier la cohérence des informations, soit pour de déduire de l'information [13]. Par exemple dans la plupart des modèles d'ontologies [9, 126, 103], pour une ontologie et une classe données, on peut calculer (1) toutes ses super-classes (directes ou non), (2) ses sous-classes (directes ou non), (3) ses propriétés caractéristiques (héritées ou locales), (4) toutes ses instances (polymorphes ou locales), etc.

- Une quatrième différence est liée au caractère *consensuel* des ontologies. Ceci permet de représenter de la même façon les mêmes concepts dans tous les systèmes d'une "communauté" (sans empêcher bien sur que chaque système possède, en plus ses concepts propres). Dans notre exemple précédent, en utilisant une ontologie du domaine médical pour définir MC_1 et MC_2, tous deux auraient sans aucun doute importé un important sous-ensemble commun de classes et de propriétés.

- Enfin, une cinquième différence (qui résulte de la première), est la souplesse de la relation "instance/ontologie". Toutes les instances des classes d'une ontologie, peuvent ne pas initialiser les mêmes propriétés. Elles n'ont pas forcement la même structure. Cette souplesse dans la description des instances est permise par le fait que les concepts des ontologies soient associés à des identifiants universels. Cela a pour conséquence de rendre les ontologies beaucoup plus simples à utiliser pour des échanges ou intégration de systèmes informatiques.

1.4 Ontologie conceptuelle et ontologies linguistiques

Le terme ontologie étant également utilisé dans le domaine du traitement automatique de la langue naturelle, une première classification des ontologies est proposée dans [126] à partir de la question suivante : l'ontologie décrit-elle des mots ou des concepts ? Les ontologies qui décrivent des mots sont dites des *ontologies linguistiques* (OL). Celles qui décrivent des concepts sont dites *ontologies conceptuelles* (OC). Ces deux types d'ontologies s'adressent à deux différents types de problèmes.

Les OLs sont les ontologies qui visent à représenter le sens des mots utilisés dans un langage particulier. Les OLs sont utilisées pour la recherche de synonymes linguistiques. Une type de problèmes visé peut s'exprimer ainsi : "trouver tous les documents pertinents par rapport à une requête qui s'exprime par un ensemble de mots connectés éventuellement par des opérateurs logiques AND, OR et NOT "*même si ces documents ne contiennent pas exactement ces mots.* Pour la représentation d'ontologies de domaine, ce type d'ontologie utilise, outre des définitions textuelles, un certain nombre de relations linguistiques (*synonyme*, *hyperonyme*, *hyponyme*, etc.) pour capturer de façon approximative et semi-formelle les relations entre les mots. La construction de telles ontologies est également souvent faite de façon semi-formelle par un processus d'extraction de termes dans un ensemble de documents du domaine et qui est ensuite validé et structuré par un expert (humain) du domaine. Un exemple d'OL est WordNet[4].

Même si elles sont parfois utilisées dans les phases préliminaires d'intégrations de bases de données hétérogènes pour identifier de façon semi-automatique les correspondances possible entre deux modèle conceptuels [19], les OLs de part leur caractère à la fois imprécis et fortement contextuel ne sont pas utilisables dans le domaine de la modélisation conceptuelle.

Au contraire, les OCs ont pour but de représenter des concepts généraux, indépendamment de leurs représentations dans un langage particulier. Les ontologies conceptuelles adoptent une approche de structuration de l'information en termes de classes et de propriétés et leur associent des identifiants réutilisable dans différents langages. Ce type d'ontologie vise à répondre aux questions du genre : "*Est-ce que deux instances données appartiennent à une même classe ? Une classe est-elle subsumée par une autre classe (à laquelle appartiennent donc ses instances quelque soit le contexte) ? Quelles sont les instances d'une classe donnée respectant un certain critère ?*"

Les OCs correspondent donc a une conceptualisation formelle au même titre qu'un modèle conceptuel. Voici quelques exemples d'OC : Dublincore[5], IEC-61360-4[6] (International Electronic Commision).

La figure 2.1 tirée de [122] fait une comparaison entre les ontologies linguistiques et les ontologies conceptuelles. Dans la suite de notre thèse, nous nous intéresserons exclusivement aux ontologies conceptuelles qui visent un univers de discours particulier aussi appelée "ontologie de

[4] `http://wordnet.princeton.edu/`
[5] `http://dublincore.org/`
[6] `http://dom2.iec.ch/iec61360?OpenFrameset`

domaine" et le terme ontologie désormais signifiera "ontologie de domaine conceptuelle".

	OL	OC
Elément	Mot	Concept
Identification des éléments	Mot	GUI
Définition des éléments	Phrase	Modèle
Taille de l'ontologie	Large	Minimale
Relations	Formelles + Linguistiques	Algébriques
Contenu	Eléments primitifs + Définitions Conservatives	Eléments primitifs + Définitions Conservatives
Focus	Orienté Classe	Orienté Propriété
Développement	Semi-automatique	Manuel
Usage de l'ontologie	Assistée par ordinateur	Automatique

FIG. 2.1 – Comparaison ontologie linguistique et ontologie conceptuelle [122]

Nous proposons dans la section suivante les formalismes usuels d'expressions des ontologies conceptuelles.

2 Les langages de définitions des ontologies

Nous présentons dans cette section quelques uns des formalismes usuels de descriptions d'ontologies. Dans la première partie de cette section, nous discutons les différents composants utilisés pour modéliser une ontologie en mettant en évidence ce qui est commun et ce qui est différent. Nous discutons ensuite dans une deuxième partie des extensions proposées au noyau commun par les modèles d'ontologies qui visent à faciliter le partage et l'échange d'information en définissant des caractérisations communes. La troisième partie présente les extensions proposées dans les modèles d'ontologies plus orientés vers l'inférence. Enfin dans la quatrième partie, nous faisons une synthèse des caractéristiques des deux familles d'ontologies décrites dans les deux parties précédentes.

2.1 Principaux composants

Les modèles d'ontologies sont basées sur cinq sortes de composants principaux [55] :

1. les classes,
2. les propriétés,
3. les types de valeurs,
4. les axiomes,
5. les instances.

2.1.1 Les classes

Une classe est la description abstraite d'un ou plusieurs objets semblables. Une classe correspond à ce qui a une existence propre, matérielle ou immatérielle, dans le domaine étudié du monde réel. Les classes sont toujours organisées en taxonomie (i.e., sous forme graphe acyclique) à l'aide de relations de subsomption. Une classe possède toujours un identifiant. Sa définition

comporte toujours une partie textuelle, qui permet de la rattacher à une connaissance préexistante de l'utilisateur, et une partie formelle constituée de relations avec d'autres composants de l'ontologie.

Selon le langage d'ontologie, des *équivalences conceptuelles* peuvent en plus être introduites en définissant des classes comme étant équivalentes à des expressions construites à partir d'autres classes. Ces expressions utilisent différents constructeurs (UNION, INTERSECTION, DIFFERENCE, COMPLEMETAIRE, etc...). De telles classes sont dites *classes définies* [87] contrairement aux autres classes dont l'intention ne peut être comprise que par une connaissance extérieure au modèle. Ces dernières classes sont appelées *classes primitives*. Ces expressions de classes permettent de distinguer deux familles d'ontologies [122] :

- les *ontologies canoniques* qui ne comportent que des classes primitives, et
- les *ontologies non canoniques* qui possèdent aussi des classes définies.

Du point de vue des bases de données, les ontologies non canoniques introduisent des redondances qui devraient faire objet de traitements particuliers (exemples vues).

2.1.2 Les propriétés

Les propriétés sont des éléments qui permettent de caractériser et de distinguer les instances d'une classe. Comme les classes, les propriétés possèdent toujours un identifiant et une définition comportant une partie textuelles et une partie formelle. Les propriétés peuvent être *fortement typées*, i.e., associées un *domaine* et un *co-domaine* précis qui permettent respectivement de spécifier les classes *susceptibles* d'initialiser une propriété et de contraindre les domaines de valeurs des propriétés. Une propriété peut prendre ses valeurs soit dans des types simples, soit dans des classes. Le terme *propriété* regroupe donc les notions parfois distinguées (par exemple dans le modèle entité-association) d'*attribut* et d'*association*.

Outre ces éléments communs, les différents langages permettent d'exprimer des caractéristiques spécifiques sur les propriétés. Par exemple, OWL permet d'exprimer des caractéristiques algébriques (symétrique, transitive, etc.) ou encore PLIB permet d'évaluer des propriétés selon leur contexte.

Enfin, certains langages de définition des ontologies (en particulier OWL), permettent la subsomption de propriétés. Une propriété est subsumée par une autre si le graphe de la relation binaire qu'elle représente est inclus dans le graphe de sa subsmante. Par exemple, supposons que nous ayons une propriété *TravailleurDans* ayant pour domaine la classe *Salarié* et pour co-domaine la classe *Organisation*. On pourrait définir la propriété *EstDirecteurDe* comme spécialisation de *TravailleurDans*. La propriété *EstDirecteurDe* aura pour domaine soit la classe *Salarié* soit une de ses sous-classes et pour co-domaine la classe *Organisation* ou l'une ses sous-classes.

2.1.3 Les types de valeurs

Les types de valeurs définissent des ensembles de valeurs dans lesquels les propriétés doivent prendre leurs valeurs, ainsi le cas échéant, que les opérations pouvant porter sur ces valeurs. Les types autorisés incluent toujours (1) des types simples (en particulier : entier, réel, caractère, booléen, etc.), (2) les classes (de telles propriétés représentent alors des associations) et (3) des collections, en général associées à des cardinalités minimum et maximum. Certains modèles d'ontologies (e.g., PLIB) introduisent des types supplémentaires.

2.1.4 Les axiomes

Les axiomes sont des prédicats qui s'appliquent sur les classes ou les instances des classes de l'ontologie et qui permettent de restreindre les interprétations possibles d'une ontologie et/ou de déduire de nouveaux faits à partir des faits connus.

Pratiquement tous les langages permettent d'exprimer des axiomes de *typage* (pour des instances, des valeurs ou des liens), de *subsomption* et de *cardinalité*. Les autres axiomes exprimables dépendent de chaque modèle particulier d'ontologie et des prédicats particuliers définis au niveau du langage. Par exemple, PLIB permet de contraindre les domaines de valeurs (e.g, RANGE) et OWL permet de contraindre une propriété à avoir une valeur dans une certaine classe (SOME).

2.1.5 Les instances

Les instances ou objets ou individus représentent des éléments spécifiques d'une classe. Chaque instance est caractérisée par son appartenance à une (ou éventuellement plusieurs) classes(s) et par des valeurs de propriétés.

Selon les langages, les instances peuvent, ou non, être associées à un identifiant unique qui permet de distinguer une instance à une autre. C'est l'hypothèse d'unicité de nom (*Unique Name Assumption* - UNA) faite par certains langages (OWL Flight) et non par d'autres (OWL). Il est possible dans certains langages de définition d'ontologie qu'une instance appartienne à plusieurs classes en même temps. C'est la notion de multi-instanciation.

2.1.6 Bilan

De la description précédente, on peut conclure que pratiquement tous les formalismes usuels de modélisation d'ontologies possèdent un noyau commun qui consiste à appréhender la connaissance d'un domaine sous la forme suivante : *Une ontologie de domaine est constituée d'un ensemble canonique de classes, structurée par des relations de subsomption et associées à des propriétés qui peuvent avoir pour valeur des classes, des types simples ou des collections. Les classes et les propriétés sont associées à des identifiants et sont définies partiellement de façon textuelle et partiellement par les relations existantes entre classes et propriétés. Des populations d'instances peuvent être définies et caractérisées à la fois par leur appartenance à des classes et par des valeurs de propriétés. Enfin des axiomes permettent de limiter les instances licites et éventuellement de déduire de nouvelles informations (nouvelles instances, nouvelles valeurs de*

propriétés ou valeur de prédicats) à partir des instances existantes.

Nous affirmons que toute architecture de bases de données à base ontologique doit être basée autour de ce noyau.

En se basant sur ce noyau commun, chaque formalisme de modélisation définit des constructions et contraintes spécifiques.

Selon les objectifs poursuivis lors du développement du formalisme de modélisation, et en nous situant dans l'optique de gestion de données qui est le notre, on peut distinguer deux types de formalismes [87].

- Les formalismes "orientés caractérisation" visent principalement à permettre d'utiliser, au sein de communauté plus ou moins étendue, des caractérisations identiques pour les mêmes objets, ceci afin de permettre l'échange ou l'intégration d'information distribuées en résolvant les problèmes d'hétérogénéité sémantiques.
- Les formalismes "orientés inférence" visent principalement à supporter des mécanismes de raisonnement et d'inférence de façon à inférer de nouveaux faits à partir des faits existants ou à calculer les similitudes entre de caractérisations différentes.

Les formalismes "orientés caractérisation" contiennent en particulier les langages du types RDF-RDF Schéma, OWL-Lite et PLIB. Les formalismes "orientés inférence" contiennent essentiellement les langages basés sur la logique de description (DAML+OIL, OWL DL, OWL Full) et ceux basés sur les F-logique (OWL Flight).

Nous présentons successivement dans les deux parties suivantes, d'abord les principaux formalismes "orientés caractérisation", puis ceux "orientés inférence". Pour chacun de ces formalismes, après avoir brièvement présenté leur contexte de conception, nous présentons les extensions qu'il propose au noyau commun et nous discutons de leur impact dans une perspective de base de données.

2.2 Ontologies orientées caractérisation

Les ontologies orientées caractérisation permettent de définir et de partager des identifications de classes et de propriétés correspondants à des sémantiques bien définies. Ces identifiants peuvent alors être référencées pour caractériser le contenu d'un document (sous forme de métadonnées) ou pour définir la sémantique d'une donnée. Nous présentons brièvement ci-dessus les caractéristiques de RDF/RDFS, OWL-Lite et PLIB.

2.2.1 RDF / RDF Schéma

RDF (Resource Description Framework) un standard du W3C[7] dont l'objectif premier était d'associer aux documents du Web des données sémantiques exploitables par machine. Les instances RDF sont décrites sous forme d'un ensemble de triplets constituant une structure séman-

[7] http ://www.w3.org/

tique. Un triplet RDF est constitué de trois éléments : <*Sujet*, *Prédicat*, *Objet*> noté communément (S, P, O) :

- Le *Sujet* est une *URI* (Uniform Resource Identifier) qui est un identifiant d'un objet appelé ressource. Une ressource peut être soit un document accessible sur le Web (dont différents triplets pourront par exemple définir différentes propriétés) soit une partie d'une page Web, soit une collection de pages, soit même un objet quelconque qui n'est pas accessible par le Web mais auquel on peut donner une URI.

- Le *Prédicat* est une propriété qui permet de caractériser la ressource. Chaque propriété possède une signification spécifique, définit ses valeurs permises, les types de ressources qu'elle peut décrire, et les relations qu'elle entretient avec les autres propriétés [98].

- L'*Objet* est la valeur de la propriété qui peut être une URI (i.e., une ressource) ou une valeur littérale (i.e. de type de données primitif défini par XML).

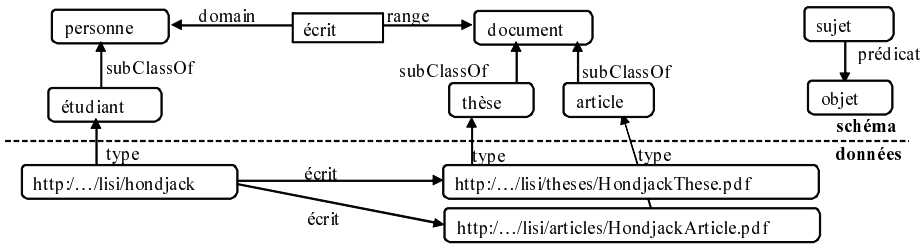
Un *Sujet* peut être *Objet* dans un autre triplet. Ce qui fait qu'une description RDF correspond à un multi-graphe orienté étiqueté. Les triplets correspondent aux arcs orientés dont le label est le prédicat, le nœud source est le *sujet* et le nœud cible est l'objet.

RDF définit lui-même très peu de prédicats, ceux-ci pouvant être définit librement. Il a donc été très rapidement été complété par RDF Schéma qui fournit les prédicats essentiels pour représenter (en RDF) une ontologie. Ces constructeurs sont les suivants.
- *rdfs :class* qui permet de créer une classe,
- *rdfs :subClassOf* qui permet de spécifier une organisation hiérarchique des classes,
- *rdfs :property* qui permet de spécifier des propriétés caractéristiques d'une classe,
- *rdfs :subProperty* qui permet au concepteur de spécifier une organisation hiérarchique des propriétés,
- *rdfs :domain* qui permet de spécifier le domaine d'une propriété i.e. à associer une propriété à une classe,
- *rdfs :range* qui permet de spécifier le co-domaine de valeur d'une propriété,
- *rdf :type* qui permet de "typer" une ressource (ou instances RDF) à une propriété ou une classe définie en RDF Schéma.

La figure 2.2 présente un exemple d'une telle information, dans la partie supérieure sous forme de multi-graphe, et, dans la partie inférieure dans la syntaxe XML définie pour RDF(S).

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

En RDF Schéma, même si dans un très nombre de cas pratiques, d'une part, les propriétés sont associées à un domaine précis et ne peuvent donc s'appliquer qu'aux objets instances de la classe correspondant à leur domaine, et d'autre part, les instances ne sont classifiées que dans



Schéma

```
<rdfs:class rdf:ID= personne/>
<rdfs:class rdf:ID=etudiant>
        <rdfs:subClassOf rdf:resource='personne />
</rdfs:class>
<rdfs:class rdf:ID=document/>
<rdfs:class rdf:ID=thèse>
        <rdfs:subClassOf rdf:resource='document/>
</rdfs:Class>
<rdfs:class rdf:ID=article>
        <rdfs:subClassOf rdf:resource='document/>
</rdfs:Class>
<rdf:property rdf:about="ecrit">
 <rdfs:range rdf:resource="document>
 <rdfs:domain rdf:resource="personne">
</rdf:property>
```

Données

```
<rdf:Description about=" http:/…/lisi/hondjack ">
<rdf:type rdf:resource="person" />
<s:ecrit> http:/…/lisi/theses/HondjackThese.pdf</s:ecrit>
<s:ecrit>http:/…/lisi/articles/HondjackArticle.pdf</s:ecrit>
</rdf:Description>
```

FIG. 2.2 – Exemple de RDF Schema

une seule classe ayant des propriétés, il n'en reste pas moins vrai que ceci n'est pas imposé par le formalisme. Chaque instance peut donc être porteuse de sa propre structure : les classes auxquelles elle appartient et les propriétés qui la décrivent. Si ceci ne pose pas de problèmes dans un univers objets où les instances sont représentées en mémoire centrale par un mécanisme pointeurs, cela a par contre beaucoup de conséquences dans une perspective base de données. Cela impose en effet, et c'est ce qui a été fait jusque là dans les approches proposées pour ranger de l'information RDF/RDFS dans des bases de données, d'utiliser un schéma éclaté dans lequel chaque classe, respectivement chaque propriété, est représentée séparément, soit sous forme de triplet, soit sous forme de table unaire (respectivement binaire).

2.2.2 OWL-Lite

Le langage OWL, nouveau standard du W3C, est une extension de ces deux prédécesseurs *RDF* [98], *RDF Schéma* [28] défini également par le W3C. En effet, OWL emprunte la syntaxe de *RDF* basé sur les triplets pour définir les concepts et emprunte également du *RDF Schema* sa faculté de décrire les entités d'un domaine d'application au moyen de concepts ou classes et propriétés qui peuvent être organisées en hiérarchies. OWL se compose en trois sous-langages : OWL Lite, OWL DL et DL. Le choix qui offre une expressivité croissante (OWL Lite $\subset$ OWL DL $\subset$ OWL Full). Le choix fait par le W3C de définir trois sous-langages résulte de l'impossibilité d'obtenir un langage à la fois compatible avec RDF et qui est un support pour faire des inférences efficaces et puissants. Dans cette section, nous présentons uniquement les constructions admises en OWL Lite. La description des constructions admises en OWL DL et OWL Full, qui sont orientés inférences, est faite dans la section 2.3.2.

2.2.2.1 Les classes

Les classes en OWL Lite sont définies à l'aide des constructeurs "*OWL :Class*" et "*OWL :subClass*". Le premier constructeur sert à créer une classe racine et le deuxième permet de faire hériter une classe par une autre en utilisant l'héritage classique. OWL autorise l'héritage multiple. OWL Lite associe à chaque concept des ontologies (classes, propriétés, etc.) une URI (*Uniform Ressource Identifier*) permettant d'être référencée par une autre ontologie ou un système. Une description textuelle peut être associée à chaque concept au moyen de la construction *owl :label*.

2.2.2.2 Les propriétés et les types de valeurs

OWL Lite définit deux types de propriétés *owl :ObjectProperty* et *owl :DatatypeProperty* qui se distinguent par leur codomaine. Les *owl :ObjectProperty* correspondent aux propriétés dont leur codomaine est une classe et les *owl :DatatypeProperty* pour les propriétés dont leur domaine sont de types simples (entier, réel, booléen, chaîne, etc.). Ce dernier type de propriétés peuvent être n'importe quels types de XML Schema. Les propriétés peuvent être hiérarchisées au moyen du constructeur *owl :subPropertyOf*.

En OWL Lite, les propriétés ne sont pas forcement rattachées aux classes (i.e., pas de domaine). En conséquence, il peut exister dans une ontologie (1) des propriétés n'ayant pas de domaine et (2) inversement, des propriétés ayant plusieurs domaine en même temps. Par exemple, on peut déclarer que la propriété *date* est liée aux classes *Personne* (date de naissance), *Concours* (date du concours), Vis (date de fabrication).

2.2.2.3 Les axiomes

Par défaut en OWL, une propriété est de cardinalité multiple, i.e., une propriété peut être initialisée plusieurs fois par une instance de classe. OWL offre un constructeur (*owl :cardinality*) pour spécifier les cardinalités (*owl :minCardinality* et *owl :maxCardinality*) d'une propriété. Les seules restrictions de cardinalité acceptées et autorisées en OWL Lite ne peuvent prendre les valeurs 0 ou 1.

OWL Lite autorise des restrictions sur le co-domaine des propriétés au moyen des constructions suivantes :

- *allValuesFrom* impose que les valeurs prises par une propriété des instances *doivent* prendre leurs valeurs dans la population d'instances d'une classe bien précise.
- *someValuesFrom* impose qu'au moins une valeur prise par une propriété *doit* appartenir à la population d'une classe précise.

OWL propose des constructeurs pour qualifier les propriétés. Celles-ci peuvent être qualifiées de :

- **inverse** (*owl :inverseOf*) : une propriété inverse correspond à la symétrie entre deux propriétés. Par exemple, si la propriété *ApourEnfant* est inverse de la propriété *Apourparent*, et si on spécifie que Sylvain *ApourEnfant* Alex, alors un raisonneur pourrait déduire que Alex *ApourParent* Sylvain.
- **transitive** (*owl :TransitiveProperty*) : une propriété transitive correspond à la transitivité

au sens Mathématiques du terme. Par exemple, Si la propriété *EstParentDe* est transitive, et si on a Antoine *EstParentDe* Patrice et Patrice *EstParentDe* Sylvain, alors, on pourrait déduire à l'aide d'un raisonneur que Antoine *EstParentDe* Sylvain.

- **symétrique** (*owl :SymetricProperty*) : une propriété symétrique correspond à la symétrie au sens mathématiques du terme. Par exemple, si la propriété *EstCollégueDe* est symétrique, et si on déclare Dung *EstCollegueDe* Hondjack, alors un raisonneur pourrait déduire que Hondjack *EstCollegueDe* Dung.
- **fonctionnelle** (*owl :FunctionalProperty*) : une propriété fonctionnelle correspond à la définition d'une fonction au sens mathématiques (f(x)=y et f(x)=z $\Longrightarrow$ y=z). Une propriété fonctionnelle pourrait être en réalité vue comme une contrainte sur une propriété, exigeant que cette dernière ait une valeur unique pour une instance donnée. Par exemple, la propriété *age* peut être qualifiée de fonctionnelle parce qu'une personne ne peut avoir plusieurs valeurs pour cette propriété.
- **fonctionnelle symétrique** (*owl :InverseFunctionalProperty*) : une propriété inverse fonctionnelle correspond à une application injective en mathématiques (f(x)=z et f(y)=z $\Longrightarrow$ x=y). Ce type de propriété correspond à la combinaison des propriétés fonctionnelles et symétriques. Par exemple, si on a la propriété *ApourNumeroDeSS* comme fonctionnelle symétrique alors un raisonneur pourrait automatiquement déduire que deux instances qui ont la même valeur pour ces propriétés sont identiques.

Afin de pouvoir établir des correspondances (mapping) entre classes, propriétés de différentes ontologies, OWL Lite propose respectivement les constructions *equivalentClass* et *equivalentProperty*. *equivalentClass* permet de spécifier que deux classes sont identiques ou synonymes. Autrement elles ont la même population d'instances. *equivalentProperty* permet de spécifier que deux propriétés sont identiques , i.e., qu'elles prennent les mêmes valeurs pour une instance. Si P_1 est équivalente à P_2, la connaissance de la valeur de P_1 pour une instance i donnée alors P_2 contient aussi la même valeur.

2.2.2.4 Les instances

Les instances des classes en OWL Lite sont définies en RDF. L'opérateur *rdf :type* de RDF Schéma permet de rattacher une instance à une classe. Comme en RDF Schema, les instances sont identifiées par une URI. OWL Lite n'impose pas également que (1) une instance puisse appartenir à plusieurs classes ou (2) elle puisse initialiser qu'un sous-ensemble des propriétés de ses classes et (3) elle puisse initialiser plusieurs fois la même propriété.

Par rapport au noyau commun à tous les modèles ontologies, comme RDF Schéma, OWL offrent également une certaine "légèreté" pour la définition des concepts et des instances. En voici quelques unes :

1. une propriété peut exister sans être attachée à une classe,
2. une propriété peut ne pas avoir de domaine,
3. une instance peut appartenir à plusieurs classes,

4. une instance d'une classe peut initialiser une propriété qu'elle ne caractérise pas (i.e., qui n'est pas définie localement et par aucune des ses super-classes).

En conséquence, les limites concernant la représentation des instances des ontologies RDF Schéma dans des schémas de tables éclatés s'appliquent également aux instances des classes OWL.

2.2.3 Le modèle d'ontologie PLIB

La norme ISO 13584 fut initiée en 1987 au niveau Européen, puis développée depuis 1990 au niveau ISO. Elle est une norme destinée à permettre la modélisation, l'échange et le référencement de catalogues informatisés de composants ou objets techniques préexistants. Baptisée sous le nom de PLIB pour Parts LIBrary, elle permet aussi bien la représentation de composants abstraits, tels qu'ils sont par exemple utilisés dans un processus de conception fonctionnelle que celle de composants fournisseurs ou normalisés [125]. Elle permet enfin d'intégrer dans un environnement homogène et cohérent des bibliothèques fournies par différentes sources [124].

Le modèle d'ontologie PLIB a été défini dans le langage de modélisation EXPRESS [79] et est composé de 218 entités. Dans les sous-sections suivantes, nous présentons brièvement comment les cinq principaux composants des ontologies (classes, propriétés, types de valeurs, axiomes et instances) sont exprimés en PLIB.

2.2.3.1 Les classes

En PLIB, les classes sont construites au moyen des entités *item_class*. Les classes sont organisées en taxonomies à l'aide de la relation de subsomption avec héritage *isa* et/ou la relation de subsomption sans héritage *case-of*. La relation de subsomption *isa* est la relation d'héritage simple qui permet l'héritage systématique de toutes les caractéristiques de toutes ses classes subsumantes. La relation de subsomption *case-of* qui permet d'importer explicitement des propriétés des classes subsumantes que la classe subsumée souhaite *effectivement* utilisées. La relation de subsomption *case-of* est aussi une relation inter-ontologie permettant de mettre en correspondance des classes provenant de différents ontologies.

Tous les concepts (classes, propriétés, types, etc.) en PLIB sont identifiés par un identifiant nommé BSU (*Basic Semantic Unit*). Précisément, l'identifiant d'une classe est nommé *class_BSU*. Chaque identifiant *class_BSU* est rattaché à l'identifiant de son fournisseur (supplier_BSU), ce qui permet de l'identifier universellement. Cet identifiant universel est appelé *absolute_id*. Chaque définition d'une classe référence sont BSU au moyen de son attribut *identified_by*. PLIB permet d'associer à chaque classe une description textuelle (nom, définition, note, remarque) et/ou graphique qui peuvent être exprimée en plusieurs langues. Ces descriptions permettent de clarifier le sens du concept.

Les classes en PLIB énumèrent explicitement les propriétés qui le caractérisent. Ces propriétés, nommées *propriétés applicables* dans le jargon PLIB, sont à distinguer des propriétés définies dans le contexte d'une classe (domaine de la propriété), dites *propriétés visibles*. Les propriétés

applicables d'une classe sont sélectionnées parmi les propriétés visibles et sont potentiellement *utilisables* pour décrire des instances. Ces deux niveaux de propriétés (visibles et applicables) ont été introduits dans le PLIB pour permettre l'utilisation de mêmes propriétés à des niveaux différents d'une hiérarchie de classe [142].

2.2.3.2 Les propriétés et les types de valeurs

Comme les classes, les propriétés sont aussi décrites textuellement et graphiquement et identifiées par des *property_BSU*. Les *property_BSU* sont rattachés à l'identifiant (class_BSU) de la classe qui les définies pour permettre de l'identifier universellement.

En plus des propriétés caractéristiques ordinaires, construit au moyen de l'entité *non_dependent_P_DET*, PLIB définit deux nouveaux types de propriétés :

- les propriétés dépendantes du contexte, et,
- les propriétés paramètres de contexte.

Les propriétés dépendantes du contexte sont les propriétés ayant une dépendance fonctionnelle avec au moins un paramètre de contexte. Elles permettent de caractériser le comportement d'une instance dans un contexte particulier [142]. Par exemple, la *température* d'une ville dépend des propriétés *date* ou *heure* qui sont des propriétés paramètres de contextes. Une propriété dépendante de contexte est définie au moyen de l'entité *dependent_P_DET* et les paramètres de contextes au moyen de l'entité *condition_DET*.

Le modèle PLIB définit plusieurs types de valeurs qui peuvent être soit des types simples (entier, réel, caractères, etc.), ou des types complexes (composition, quantitatif), ou des types nommées (types avec un identifiant absolu), ou des types agrégats (tableau, liste, ensemble, sac).

Une description plus détaillée de ces types de valeurs est fournie dans l'annexe A. Notons d'Ore et déjà que PLIB propose parmi les types simples des types avec à des unités de valeurs (physiques, monétaires, etc.).

2.2.3.3 Les instances

Les instances des classes en PLIB sont définies sous forme de liste de couples propriété-valeur. Les classes ayant une population, énumèrent explicitement toutes leurs instances. Les instances en PLIB appartiennent à une *unique* classe.

Une particularité du modèle PLIB est qu'il permet de représenter les multiples connaissances que l'on peut avoir sur une instance selon différents contextes ou points de vue ou disciplines qui manipulent l'instance. Pour réaliser cela, PLIB définit pour chaque contexte ou point de vue un objet particulier. Ces objets sont appelés des *modèles techniques* ou *vues fonctionnelles*. Ainsi, un objet du monde réel est représentée dans le modèle d'information de PLIB, comme un *agrégat d'instances*, où chaque objet correspond à un ensemble d'instances, chacune correspondant à un point de vue particulier et appartenant à une classe. Cette approche permet justement d'ajouter autant de représentations que nécessaire à une instance. Elle permet également de ne pas modifier

le modèle lorsqu'un point de vue nouveau apparaît nécessaire. C'est l'approche adoptée par PLIB pour mettre en œuvre la multi-instanciation. Notons que chacune des vues fonctionnelles sont définies sous forme de classe et liée à leur classe caractéristique à l'aide de la relation sémantique *is_ view_ of*. Pour éviter, toute redondance au niveau des valeurs des propriétés des instances, chaque point de vue n'initialise pas les mêmes propriétés.

Nous faisons dans l'annexe A, une description détaillée du modèle d'ontologie PLIB, pour résumer en quelques lignes les caractéristiques de PLIB, notons que :

1. l'identifiant des classes et propriétés est appelé BSU (*Basic_ Semantic_ Unit*),
2. une propriété est *obligatoirement* attachée à une classe. Toute propriété est forcement liée à une classe et constitue dans le jargon PLIB une *propriété visible*. Cette propriété pourra ensuite être rendue *applicable* dans la classe qui l'a définie ou dans une de ses sous-classes.
3. le domaine d'une propriété est *unique* et *obligatoire* faisant partie de la définition de la propriété,
4. une instance appartient à une classe de base, autrement, une instance appartient à une et unique classe. La multi-instanciation est réalisée sous forme d'*agrégation d'instances* grâce à l'utilisation de vue fonctionnelle,
5. les propriétés initialisées par une instance d'une classe sont un sous-ensemble des propriétés applicables de sa classe,
6. le modèle d'ontologie PLIB offre pour cela une relation d'extension particulière, appelée "*case_ of*" (est_un_cas_de). Cette relation permet à un utilisateur de définir sa propre ontologie à partir d'une ontologie de référence. Une classe qui subsume une classe d'une ontologie de référence par la relation *case_ of*, peut importer implicitement de cette classe un sous-ensemble quelconque des propriétés qui y sont définies. Ceci permet de définir, à partir d'une même ontologie de référence, des ontologies utilisateurs de structures très différentes.

2.3 Ontologies orientées inférences

Nous présentons dans cette partie les principaux langages de définition d'ontologies permettant de définir les ontologies orientées inférences : OWL DL et Full. Avant de présenter les constructions qu'ils proposent, nous faisons dans la sous-section suivante une brève de la logique de description sur lequel se base ces langages.

2.3.1 Un bref aperçu de la logique de description

La logique de description (ou encore la logique terminologique) est une famille de langage de représentation de connaissance basée sur des *concepts* (ou classes) et *rôles*. Un *rôle* est une relation binaire entre deux *individus*. Un *concept* en logique de description correspond à une *entité* générique du domaine d'application et un *individu* à une instance d'une entité. Les concepts et rôles peuvent être organisés par niveau de généralité (ou hiérarchique) au moyen de relation de *subsomption*. En logique de description, on désignera par *TBox*, la représentation et manipulation des *concepts* et *rôles*, qui correspond autrement au niveau *terminologique*. La description

et manipulation des instances qui correspond au niveau *factuel* ou niveau *assertion*, est appelée *ABox*.

Les concepts en logique de description peuvent être soient des *concepts primitifs* ou des *concepts définis*. Les concepts définis sont construits à partir des concepts primitifs par une combinaison de constructeurs et éventuellement de restrictions sur les rôles. A l'image des concepts, les rôles eux-aussi peuvent être définis à partir d'autres rôles.

En logique de description, les opérations de bases au raisonnement terminologique sont la *classification* et l'*instanciation*. La *classification* s'applique aux concepts et aux rôles et permet de déterminer la position d'un concept ou d'un rôle dans leurs hiérarchies respectives. L'*instanciation* est une opération qui permet de retrouver les concepts dont un individu est susceptible d'être une instance.

La table 2.1, présente quelques constructeurs de la logique de description dans la syntaxe *lispienne* et *allemende*.

Syntaxe lispienne	**Syntaxe allemande**	**Définition**
Top	$\top$	Concept le plus général
Bottom	$\bot$	Concept le plus spécifique
(and C D)	C$\sqcap$D	Conjonction de concepts
(or C D)	C$\sqcup$D	Disjonction de concepts
(not A)	$\neg$ A	Négation
(all r C)	$\forall\ r.C$	Quantificateur universelle
(some r C)	$\exists\ r.C$	Quantification existentielle
(atleast n r) ou ($\leq\ n\ r$)		Cardinalité minimale d'un rôle
(atmost n r) ou ($\geq\ n\ r$)		Cardinalité maximale d'un rôle
(restrict r C)	$r\|C$	Contrainte sur le co-domaine d'un rôle
	C$\sqsubseteq$D	Subsomption entre concepts
	C$\equiv$D	Équivalence entre concepts

Tab. 2.1 – Constructeurs de la logique de description dans la syntaxe *lispienne* et *allemande*. A, C, D sont des noms de concepts et r un nom de rôle

.

Ils existent plusieurs familles de langages de la logique de description. On peut citer $\mathcal{AL}$, $\mathcal{FL}$, $\mathcal{SHOQ}$[76], $\mathcal{SHIQ}$[72, 77], $\mathcal{SHOIN}$[78], etc. Toutes ces familles de langages de la logique de description se distinguent par les constructeurs qu'ils définissent. Par exemple, la famille de $\mathcal{AL}$ qui est définie par
$\mathcal{AL} = \{\top, \bot, \neg A, C \sqcap D, \forall r.C, \exists\ r\}$ peut être enrichi au moyen des constructeurs de la DL pour donner de nouveau langage. La table suivante présente un exemple de quelques langages de la famille de $\mathcal{AL}$.

Langage	**Définition**
$\mathcal{ALC}$	$\mathcal{ALC} \cup \{\neg A\}$
$\mathcal{ALU}$	$\mathcal{ALU} \cup \{C \sqcup D\}$
$\mathcal{ALE}$	$\mathcal{ALE} \cup \{\exists\ r.C\}$
$\mathcal{ALN}$	$\mathcal{ALN} \cup \{(\leq nr), (\geq nr)\}$
$\mathcal{ALR}$	$\mathcal{ALR} \cup \{r_1 \sqcap r_2\}$

La famille des langages de la logique de description offre des constructeurs pour la description de concepts (ou classes) et rôles. Les constructeurs de descriptions des concepts ou rôles possèdent une sémantique précise par l'intermédiaire d'une *interprétation*. Les manipulations effectuées sur les concepts et rôles sont réalisées en fonction de cette sémantique. Les concepts sont interprétés comme un sous-ensemble d'un domaine d'interprétation Δ^I et les rôles comme des sous-ensembles du produit $\Delta^I \times \Delta^I$.
Par définition [116], *une interprétation* $I = (\Delta^I, .^I)$ *est la donnée d'un ensemble* Δ^I *appelé domaine de l'interprétation et d'une fonction* $.^I$ *qui fait correspondre à chaque concept un sous-ensemble de* Δ^I *et à un rôle un sous-ensemble de* $\Delta^I \times \Delta^I$ *et vérifiant les équations de la table 2.2.*

Opérateurs	**Sémantique**
$\top^I$	Δ^I
$\bot^I$	$\oslash$
$(\neg C)^I$	$\Delta^I - C^I$
$(\exists r.C)^I$	$\{x \in \Delta^I / \exists y, (x,y) \in R^I \wedge y \in C^I\ \}$
$(\forall r.C)^I$	$\{x \in \Delta^I / \forall y, (x,y) \in R^I \Rightarrow y \in C^I\ \}$
$(\leq nr)^I$	$\{x \in \Delta^I / \lvert\{y, (x,y) \in R^I \wedge y \in C^I\}\rvert \leq n\}$
$(\geq nr)^I$	$\{x \in \Delta^I / \lvert\{y, (x,y) \in R^I \wedge y \in C^I\}\rvert \geq n\ \}$
$(C \sqcap D)^I$	$C^I \cap D^I$
$(C \sqcup D)^I$	$C^I \cup D^I$

TAB. 2.2 – Opérateurs (*de base*) de construction de concepts de la logique de description (LD)

Dans les deux sections qui suivent, nous présentons les langages OWL DL/Full.

2.3.2 OWL DL / OWL Full

La conception du langage OWL a été influencée à la fois par (1) la logique de description, (2) les langages ontologiques préexistants en particulier DAML+OIL, (3) les langages de frames et (4) les langages RDF et RDF Schéma du W3C. OWL s'est inspiré des langages de la logique de description en particulier pour la formalisation de la sémantique, et du choix des constructeurs des classes et de l'intégration des types de bases des propriétés et des données. La figure 2.3 présente la correspondance des différents constructeurs de OWL avec la logique de description.

Les constructeurs de classes proposées dans le langage OWL (Full et DL) qui peuvent être combinées pour définir de nouvelles classes dites *classes définies*. Ces constructeurs sont entre autres :

OWL Construct	DL	OWL Axiom	DL
intersectionOf	$C_1 \sqcap ... \sqcap C_n$	SubClassOf	$C_1 \sqsubseteq C_2$
unionOf	$C_1 \sqcup ... \sqcup C_n$	EquivalentClasses	$C_1 \equiv ... \equiv C_n$
complementOf	$\neg C$	SubPropertyOf	$P_1 \sqsubseteq P_2$
oneOf	$\{o_1, ..., o_n\}$	EquivalentProperties	$P_1 \equiv ... \equiv P_n$
allValuesFrom	$\forall P.C$	SameIndividual	$o_1 = ... = o_n$
someValuesFrom	$\exists P.C$	DisjointClasses	$C_i \sqsubseteq \neg C_j$
value	$\exists P.\{o\}$	DifferentIndividuals	$o_i \neq o_j$
minCardinality	$\geq nP.C$	inverseOf	$P_1 \equiv P_2^-$
maxCardinality	$\leq nP.C$	Transitive	$P^+ \sqsubseteq P$
cardinality	$= nP.C$	Symmetric	$P \equiv P^-$

FIG. 2.3 – Correspondances des constructeurs de OWL avec ceux de la logique de description

1. **Union** (*owl :UnionOF*) permet de déclarer une nouvelle classe en étant l'union de deux ou plusieurs classes. Par exemple, on pourrait déclarer que la classe *CitoyenDuMonde* est l'union des classes *CitoyenFrance*, *CitoyenTchad*, *CitoyenUS*, etc. ($CitoyenDuMonde \equiv CitoyenFrance \cup CitoyenTchad \cup CitoyenUS \cup ...$)
2. **Complément** (*owl :complementOF*) permet de déclarer une classe comme étant le complémentaire de deux classes. Par exemple, on pourrait déclarer que la classe *Femme* est le complémentaire de la classe *Personne* dans la classe *Homme* ($Femme \equiv Personne - Homme$).
3. **Intersection** (owl :intersectionOF) permet de déclarer une nouvelle classe comme étant l'intersection de deux ou plusieurs classes. Par exemple, la classe *hermaphrodite* peut être défini comme l'intersection des classes *Homme* et *Femme*. $Hermaphrodite \equiv Homme \cap Femme$.
4. **Disjonction** (*owl :disjointWith*) permet de déclarer la contrainte que l'extension de deux classes données sont disjointe. Les deux classes ne partagent pas d'instances en commun. Par exemple les classes *Homme* et *Femme* sont disjointes.
5. **Restriction** (*owl :Restriction*) permet de déclarer une nouvelle classe à partir d'une autre tout en définissant des conditions sur les instances de la classe. Ces conditions peuvent être entre autres des restrictions sur le domaine de valeurs d'une ou plusieurs propriétés de la classe, la cardinalité des propriétés, etc. Par exemple, on peut déclarer la classe *Adulte* comme étant une classe *Personne* dont l'age est inférieur à 18 ans.
6. **Énumération** (*owl :oneOf*) permet de déclarer une classe par extension. La population d'instances de la liste est listée dans un ensemble. Par exemple, on peut déclarer que la classe continent ={Afrique, Europe, Amérique, Asie, etc.}.
7. **hasValue** permet de déclarer une variable de classe (équivalente à propriété statique en programmation objet) et impose donc la valeur spécifiée à toutes les instances de la classe en question.

Le langage OWL DL (Description Logic) est un sous-langage d'OWL. OWL DL restreint l'utilisation des constructeurs d'OWL. Les constructeurs autorisés ont été choisis de manière à

ce que OWL DL soit décidable sur lequel des raisonnements puissent être possible. Le problème d'inférences en OWL DL a ainsi pu être classé comme étant aussi difficile que celui de l'inférence dans les langages de la famille $\mathcal{SHOIN}$ (temps exponentiel non déterministe [158]). Le langage OWL DL est destiné aux applications qui demandent une expressivité maximale et qui requiert à ce que toutes les inférences soient garanties calculables et décidables, i.e., que tous les calculs s'achèveront dans un intervalle de temps fini [44]. L'avantage d'OWL DL est qu'il supporte bien le raisonnement mais n'est malheureusement pas compatible avec RDF [9].

Le langage OWL Full est la version complète d'OWL. Le langage OWL Full utilise tous les constructeurs possibles en OWL. Il autorise également toutes les combinaisons possibles de tous ces constructeurs. Les avantages de OWL Full sont que (1) il est entièrement compatible avec RDF, i.e., les classes, les propriétés et les instances sont au même niveau. Par exemple, il est possible d'indiquer qu'une classe donnée ne doit pas avoir plus de deux superclasses. (2) Il a un grand pouvoir d'expression.

Son inconvénient est que les ontologies définies avec ce sous-langage peuvent être indécidable [9]. Le langage OWL Full est donc destiné aux applications qui demandent une expressivité maximale et la liberté syntaxique de RDF mais sans garantie de calcul. Par exemple, dans OWL Full, une classe peut se traiter simultanément comme une collection d'individus ou comme un individu à part entière [44].

Les ontologies orientées inférences introduisent la difficulté de représentation des constructions (qui peuvent être combinées entre elles pour former une expression) dans une perspective base de données. Notons que de pareilles constructions n'existent dans des bases de données et qu'il faudrait leur trouver une équivalence. Deux solutions sont envisageables.

- La première solution est l'utilisation des vues SQL (matérialisées ou virtuelles) pour calculer l'expression des classes définies.
- La deuxième solution consiste à saturer la base de données en représentant explicitement pour chaque classe définie, ses instances dans une table dans la base de données.

La première solution, même si elle parait la plus raisonnable, présente néanmoins des inconvénients lorsqu'il s'agit d'intégrer dans la base de données des instances provenant de certaines classes définies. Le cas le plus simple est celui d'une classe définie (C_3) construit par une intersection entre deux classes C_1 et C_2. La classe C_3 peut être simplement calculée par une vue SQL qui fera l'intersection de l'extension des deux classes C_1 et C_2. Mais lorsqu'on dispose d'une instance de C_3 à intégrer dans la base de données, dans quelle classe faudrait-il la ranger, dans C_1 ou C_2 ou dans les deux ? La réponse n'est pas évidente et tout choix pourrait avoir des conséquences ailleurs. La deuxième solution introduit de la redondance dans la base de données et peut poser des difficultés lors des mises à jour des instances.

2.4 Discussion sur les langages de définitions d'ontologies : notre position

A la lumière de la description de tous ces langages de définitions d'ontologies, on peut remarquer qu'ils présentent à la fois des points communs et des différences. Les points communs

se situent au niveau du fait qu'ils sont centrés sur un noyau commun. Tous,

- conceptualisent l'univers à l'aide de classes et propriétés. Les classes ont organisées en hiérarchie au moyen des relations de subsomption.
- associent à chaque concept des ontologies (classes, propriétés, etc.) un identifiant unique permettant d'être référencé par une autre ontologie ou un système. En PLIB, cet identifiant est appelé BSU (*Basic_ Semantic_ Unit*). En OWL, cet identifiant est appelé URI (*Uniform Ressource Identifier*).

Mais pour ce qui concerne leurs différences, on constate les faits suivants :

- certaines ontologies adoptent un point de vue non typé. Une instance appartient à plusieurs classes. Une propriété peut ne pas avoir de domaine et de co-domaine définis,
- d'autre au contraire, comme le modèle PLIB, impose un *typage fort* des instances et des propriétés. La multi-instanciation est remplacée par la notion d'agrégat d'instances ;
- les langages issus de la logique de description (OWL DL/Full) sont plus centrés sur des opérateurs de classes qui permettent de définir ces classes définies (ou non-primitives). Ces opérateurs (ou constructeurs) sont à la base des inférences. Ces langages proposent un ensemble d'opérateurs (UNION, INTERSECTION, etc.) qui peuvent être combinés pour définir de nouvelles classes.
- le langage de définition PLIB est orienté *propriété* et canonique, c'est-à-dire que les concepts définis avec PLIB sont primitifs et caractérisés par un ensemble de propriétés.

La conclusion de cette présentation sur les langages de définition d'ontologies, bien qu'ils se basent tous sur un noyau commun, ils ont des caractéristiques qui leur son propre. Aucun de ces langages ne peut donc remplacer un autre sans aucune perte d'information [55].

Dans la section suivante, nous abordons la persistance des ontologies et des données à base ontologique dans des bases de données.

3 Gestion de données à base ontologique

Dans le début du chapitre, nous avons décrit les ontologies en donnant leurs principales composantes et caractéristiques. Notons que le terme "ontologie" est souvent utilisé pour recouvrir deux aspects. Une partie "ontologie" à proprement parler dans laquelle les classes et les propriétés sont décrites ainsi que les relations (fonctionnelles) entre elles, une deuxième partie constituée des instances des classes. Chaque instance est constituée d'un ensemble de valeurs de propriétés qui caractérisent l'instance. Nous réserverons le terme ontologie pour la partie classe, et nous parlerons de données à base ontologique pour la partie instance.

Les ontologies de domaine se construisent de plus en plus et des données à base ontologique associées à ces ontologies sont produites également de plus en plus dans de nombreuses applications (portail Web, base de données technique, etc.) [43, 75, 115, 117]. Ces données à base ontologique étaient gérées depuis longtemps en mémoire par les systèmes informatiques (comme les moteurs d'inférence). Avec la croissante de ces données, leur gestion en mémoire devient dé-

licate, d'où les efforts fournis ces dernières années pour leur représentation dans des systèmes de gestion de bases de données pour profiter de leur performance. Plusieurs approches de ce type de base de données, appelées *bases de données à base ontologique*, ont été proposées dans la littérature [6, 25, 30, 23, 121, 102, 68, 149]. Ces approches se distinguent les unes des autres par les schémas de représentation qu'elle propose.

La suite de cette section s'articulera comme suit. Nous proposons dans la sous-section 3.1, quelques définitions des bases de données à base ontologique et de leurs systèmes de gestion. Nous présentons dans la sous-section 3.2, les différents schémas de données existants dans la littérature pour le stockage des ontologies et des données à base ontologique et qui permettent d'atteindre les fonctions attendues. Dans la sous-section 3.3, nous présentons les caractéristiques et les fonctionnalités des principaux systèmes de gestion de BDBOs existants dans la littérature. Enfin dans la sous-section 3.3.5, nous faisons une synthèse des systèmes que nous présentés pour identifier les caractéristiques et les fonctionnalités générales des systèmes de gestions de BDBOs que nous considérerons comme essentielles pour notre architecture. Par rapport à notre domaine d'application cible qui est le domaine technique, nous présentons quelques fonctionnalités supplémentaires que notre architecture devrait prendre en compte.

3.1 Définition

On propose dans cette section une définition de base de données à base ontologique et de leur système de gestion.

On appelle *base de données à base ontologique* (BDBO) [127, 45] une source de données qui contient :
- une ontologie (locale) ,
- éventuellement, des références de l'ontologie locale à des ontologies (partagées) externes,
- un ensemble de données,
- une relation entre chaque donnée et la notion ontologique qui en définit le sens.

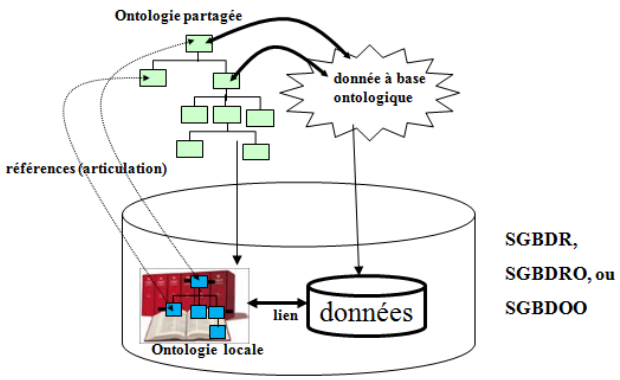


FIG. 2.4 – Définition d'une base de données à base ontologique

Une *base de données à base ontologique* possède deux caractéristiques :

- les ontologies et les données sont toutes deux représentées dans une unique base de données et peuvent faire l'objet des mêmes traitements (insertion, mise à jour, interrogation, versionnement, etc.) ;
- toute donnée est associée à un élément ontologique qui en définit le sens et vice versa.

Plusieurs architectures pour représenter dans des SGBD relationnels des méta-données RDF, RDFS, DAML+OIL ou OWL ont été implémentées (Sesame [30], RDFSuite [6, 74], Jena [23, 165, 31], DLDB [121], RStar [102], KAON [25], 3Store [68], PARKA [149]). Chacun de ces systèmes implémente des schémas spécifiques pour le stockage des ontologies et des données à base ontologique dans des bases de données. Avant de décrire les principaux systèmes de gestion des bases de données à base ontologique existants, nous présentons dans la section suivante des schémas de bases de données existantes dans la littérature pour le stockage des ontologies et des données à base ontologique.

3.2 Schémas de représentation des ontologies et des données

La problématique de la représentation des ontologies dans les bases de données est donc :

(1) de définir un schéma de tables pour le stockage des données de chacune des parties (ontologie et données). En effet, à la fois, les ontologies et les instances des classes d'ontologies ont une structure objet, c'est-à-dire qu'elles représentent l'information sous forme de classes avec des relations (de composition et de subsomption) entre elles et sont constituées d'un ensemble de propriétés caractéristiques. La représentation des objets dans des bases de données relationnelles ou relationnelles-objets pose des difficultés car tous les concepts objets ne se traduisent pas directement en relationnel et les procédures de gestion (ajout, mise à jour, suppression, versionnement) des instances sont entièrement à définir [141, 145, 151].

(2) de définir un mécanisme pour lier les instances (de la partie "données") avec les éléments ontologiques (de la partie "ontologie") qui en définit le sens et vice versa. En effet, afin de pouvoir d'accéder aux instances des classes via l'ontologie, les schémas de tables de chacune des deux parties de l'architecture de base de données à base ontologique doivent prendre en compte ce paramètre pour permettre à ce que l'on puisse accéder aux données via les ontologies et inversement d'accéder aux ontologies via les données.

Nous présentons dans les deux sous-sections suivantes les différentes approches de représentation des ontologies et des données à base ontologique. Nous illustrerons les différents schémas proposés à l'aide de l'exemple de la figure 2.5.

3.2.1 Représentation des ontologies

Les différentes approches de représentation des ontologies peuvent être classées en deux catégories :

- approche de représentation *verticale*,
- approche de représentation *spécifique*

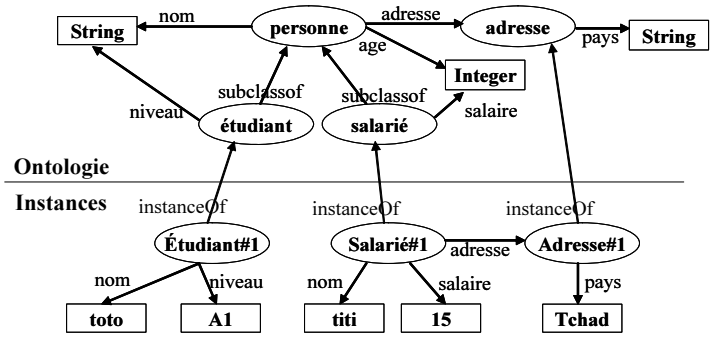


FIG. 2.5 – Exemple d'ontologie et données à base ontologique en RDF Schéma

3.2.1.1 Approche de représentation *verticale*

Dans cette approche, la description des classes et des propriétés des ontologies (RDF Schéma) sont stockées naturellement sous forme de triples (sujet-prédicat-objet) dans une unique table nommée *triples* [23, 5, 23, 165, 31]. La table *triples* est constituée de trois colonnes :

- la colonne *subject* qui représente l'URI des ressources RDF/S ;
- la colonne *prédicate* représente le nom (ou l'identifiant) de la propriété de l'instance à initialiser ;
- enfin la colonne *object* représente la valeur de la propriété qui peut être soit une autre instance (ou ressource) ou une valeur littérale. Le domaine de la propriété est de type *VARCHAR*.

Dans la figure 2.6, nous présentons un exemple d'un tel schéma initialisé avec l'ontologie de la figure 2.5.

Table Triples		
SUBJECT	PREDICAT	OBJECT
personne	rdf:type	rdfs:class
étudiant	rdf:type	rdfs:class
salarié	rdf:type	rdfs:class
étudiant	rdfs:subClassOf	personne
salarié	rdfs:subClassOf	personne
nom	rdf:type	rdfs:property
nom	rdf:range	xsd:String
...	...	...

FIG. 2.6 – Schéma de l'approche verticale de la représentation des ontologies

3.2.1.2 Approche de représentation *spécifique*

La structure des tables de cette approche est simple et se rapproche la structure du (ou des) langage(s) de définition d'ontologie(s) des ontologies à intégrer. Le schéma de ces tables est défini de façon ad hoc (c'est-à-dire, de façon non systématique) d'une implémentation à une autre. Toutes les implémentations actuelles définissent chacune la structure de toutes ces tables suivant

les détails des informations qu'ils veulent capturer et du lien fait avec la partie *données*. Par exemple pour des ontologies RDF Schema, leur schéma contient au moins les tables suivantes : *class*, *subclass*, *property*, *subproperty*, *domain*, *range* (cf. figure 2.7). Nous présentons dans la figure 2.7 un exemple de ce type de schéma.

RANGE	
TYPE	PROPERTY
xsd:string	1
xsd:integer	1
4#CLASS	1
xsd:string	2
xsd:float	3
xsd:string	4

PROPERTY	
ID	NAME
1	Nom
2	Age
3	Adresse
4	Niveau
5	Salaire
6	pays

DOMAIN	
PROPERTY	CLASS
1	1
2	1
3	1
4	2
5	3
6	4

CLASS	
ID	NAME
1	personne
2	étudiant
3	salarié
4	adresse

SUBCLASS	
super	sub
1	2
1	3

SUBPROPERTY	
super	sub

FIG. 2.7 – Schéma de l'approche *spécifique* de la représentation des ontologies

3.2.2 Représentation des données

Les différentes approches de représentation des données à base ontologique peuvent être classées en trois catégories :

- approche de représentation *générique* ou *verticale*,
- approche de représentation *binaire*,
- approche de représentation *hybride*.

Dans les sous-sections suivantes, nous présentons les schémas de chacun de ces approches.

3.2.2.1 Approche de représentation *verticale*

Les données à base ontologique sont représentées dans cette approche sous forme de triplets (subject, predicate, object) comme dans l'approche de verticale de la représentation des ontologies (cf. section 3.2.1.1).

Une autre variante de cette approche consiste à stocker respectivement l'identifiant (URI) des ressources et toutes les valeurs littérales dans des tables spécifiques : *ressources* et *littéral* (cf. figure 2.8b). 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 ressources et des littéraux dans la colonne *id* de ces deux tables sont référencés sans la table *triples*.

Dans la suite, nous désignons la variante avec URI par "approche verticale avec URI" et la deuxième variante par "approche verticale avec ID".

Ces deux variantes de l'approche de représentation *verticale* sont illustrées dans la figure 2.8.

Table Triples		
SUBJECT	PREDICAT	OBJECT
étudiant#1	rdf:type	étudiant
salarié#1	rdf:type	salarié
étudiant#1	nom	toto
salarié#1	nom	titi
...	...	...

(a) Approche avec URI

Table Triples		
SUBJECT	PREDICAT	OBJECT
5	rdf:type	2
6	rdf:type	3
5	4	1
6	4	2
...	...	...

(b) Approche avec ID

RESOURCES	
id	URI
1	personne
2	étudiant
3	salarié
4	nom
...	...
5	étudiant#1
6	salarié#1
...	...

LITTERAL	
id	Value
1	toto
2	titi
...	...

FIG. 2.8 – Exemple de l'approche verticale de représentation des données à base ontologique

3.2.2.2 Approche de représentation *binaire*

Dans cette approche, les instances des classes et les valeurs de leurs propriétés sont stockées dans des tables séparées (les instances à part et leurs valeurs de propriétés d'un autre coté).

3.2.2.2.1 Représentation des identifiants des instances

Pour la représentation des instances, trois approches existent (ISA, NOISA et table unique). Elles se distinguent des unes des autres par le choix de l'héritage adopté par chacune.

- Une première approche [102], 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 (*id*, *classID*) : la colonne *id* permet de représenter l'identifiant (ou l'URI) des instances (ou ressource) et la colonne *classID*, l'identifiant (l'URI) de la classe stockée dans l'ontologie (cf."Table unique" dans la figure 2.9).
- Une deuxième approche dite *ISA* [157] qui (cf. figure 2.9), consiste à associer à chaque classe, stockée dans la partie *ontologie*, une table spécifique permettant de stocker les identifiants des instances. Une table d'une classe C_i héritera éventuellement de la classe de super-classe. Chaque table d'une classe C_i est constituée d'une colonne *ID* qui servira à stocker l'identifiant (URI) des instances de la classe. Cette approche a l'avantage de profiter des potentialités du SQL99 lorsqu'il s'agit de faire des requêtes hiérarchiques (i.e polymorphiques), mais n'est pas efficace pour les requêtes de mise à jour portant sur la structure. Par exemple l'insertion d'une classe entre deux classes demande de supprimer des tables puis de la remettre en relation [6].

- La troisième variante, une alternative de la première, appelée *NOISA* [157], consiste à ne pas utiliser l'héritage de tables offert par les SGBDROs. Les tables des propriétés et des classes sont définies séparément sans être mises en relation. La réalisation d'une requête polymorphe (ou transitive) nécessite d'accéder à l'ontologie pour récupérer toutes les classes et/ou propriétés concernées impliquées dans la requête. Même si cette approche est beau-

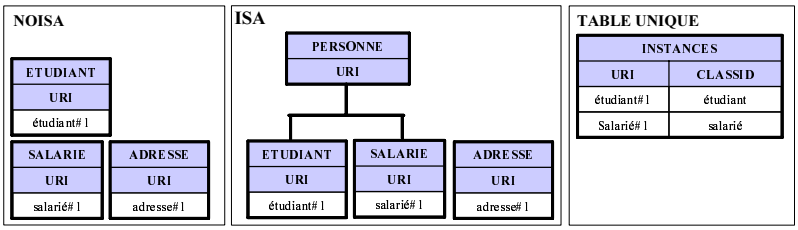


FIG. 2.9 – Tables partie *données* de l'approche *spécifique* : représentation des instances

coup moins coûteuse par rapport à la première variante lors des mises à jour, elle présente de mauvaises performances si les requêtes polymorphes sont exécutées sur des racines très profondes. Les opérations de calcul de sous-classes ou de sous-propriétés sont des opérations récursives et très coûteuses en temps.

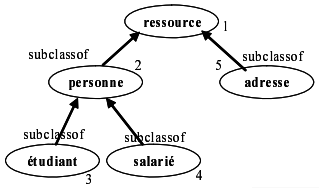
Toutefois, en adoptant les techniques de marquage (ou "labelling") des classes [36] des hiérarchies existantes dans d'autres environnements (BDs XML, représentation de connaissances, langages de modélisation objets, etc.) [4, 41, 99], on arrive à des bonnes performances. La technique de marquage consiste à associer à chaque classe d'une hiérarchie un *label* qui permettra en une seule requête (moins coûteuse) de calculer les sous-classes ou super-classes d'une classe donnée [36]. Trois techniques existent pour la définition du *label* des classes des hiérarchies (cf. figure 2.10) :

1. *bit-vector schemes* [32] : dans cette technique, le label d'une classe est un ensemble ordonné de n bits $(b_0, b_1, .., b_i, ..., b_n)$ où n représente le nombre de classes dans la hiérarchie et $b_i \in \{0, 1\}$. Soit C_i la classe i dans la hiérarchie, si dans un vecteur de bits d'une classe C_j donnée, on a b_i=1 alors $C_i = C_j$ ou C_j est une sous-classe de C_i.
2. *interval schemes* : dans cette approche, un couple de valeurs entières (*start*,*end*) est attribué à chaque classe de la hiérarchie. Cette intervalle de valeurs est définie de telle sorte qu'elle soit contenue dans l'intervalle de celle de sa super-classe. [4, 48, 49, 99] proposent des algorithmes de calcul des bornes de l'intervalle (*start*,*end*) basés sur l'indice des classes respectivement sur un parcours préfixé et postfixé de la hiérarchie des classes.
3. *prefix schemas* [155, 89] : dans cette approche, le *label* d'une classe est une chaîne formée par la concaténation du *label* de sa super-classe et de l'identifiant de la classe. Ainsi pour vérifier qu'une classe C_j est super-classe (directe ou transitive) d'une classe C_i donnée, il suffit que le *label* de C_i soit contenu dans celui de C_j.

3.2.2.2.2 Représentation des valeurs de propriétés

Pour la représentation des valeurs de propriétés, deux approches sont possibles :

- La première approche, appelée *triples*, consiste à représenter les valeurs de propriétés dans une table à trois colonnes (*Subject*, *Predicate*, *Object*) comme dans la figure 2.8.
- La deuxième approche, appelée *table par propriété*, vise à définir une table spécifique pour chaque propriété des instances des classes. La table associée à chaque propriété est consti-



Tables	bit-vector	préfix	Interval
ressource	00001	1	[1,6]
personne	00011	12	[2,4]
étudiant	00111	13	[3,3]
salarié	01011	14	[4,4]
adresse	10001	15	[5,5]

CLASS		
ID	NAME	bit-vector

CLASS		
ID	NAME	préfix

CLASS			
ID	NAME	start	end

FIG. 2.10 – Marquage de classes

tuée de deux colonnes : *id* pour l'identifiant de l'identifiant (URI) et *value* pour la valeur des propriétés. Le co-domaine de la colonne *value* correspond à celui de sa propriété décrite dans la partie *ontologie*. La figure 2.11 montre un exemple de tables pour la représentation *table par propriété*. Ce schéma se caractérise par le fait qu'il est *dynamique* parce qu'il évolue au fur et à mesure que de nouvelles propriétés sont initialisées dans les données à base ontologique. Ces ajouts se traduisent dans la partie *données* par de nouvelles tables.

Table SALAIRE	
ID	VALUE
salarié#1	1500

Table PAYS	
ID	VALUE
salarié#1	Tchad

Table NOM	
ID	VALUE
étudiant#1	toto
salarié#1	titi

Table ADRESSE	
ID	VALUE
salarié#1	Adresse#1

Table NIVEAU	
ID	VALUE
étudiant#1	A1

Table AGE	
ID	VALUE

FIG. 2.11 – Tables de la partie *données* de l'approche spécifique : représentation des valeurs des propriétés

3.2.3 Approche de représentation hybride

L'approche *Hybride* combine l'approche de représentation par triples et l'approche de représentation *binaire*. Les valeurs des propriétés selon leur co-domaine (Entier, String, etc.) sont représentées dans des tables séparées. On aura une table *triples* pour chacun des types de bases (INT, VARCHAR, FLOAT, etc.) : une table pour les valeurs des propriétés de type entier, etc. Le type de la colonne *Objet* est du type de base (cf. figure 2.12).

3.2.4 Bilan

Dans toutes les approches de représentation des données à base ontologique, les instances et les valeurs des propriétés sont représentées séparément soit sous forme de tuples dans une grande table ou dans des tables de propriétés. Ces choix de représentation sont justifiés par la *souplesse*

Triples (ENTIER)		
SUBJECT	PREDICAT	OBJECT
étudiant#1	Salaire	15
...	...	...

Triples (STRING)		
SUBJECT	PREDICAT	OBJECT
étudiant#1	nom	toto
salarié#1	nom	titi
...	...	...

FIG. 2.12 – Tables de l'approche hybride

qu'offre les langages de définition d'ontologies (RDF Schéma, DAML+OIL, OWL) comme nous l'avons montré dans la section précédente. En effet, ces langages de définition d'ontologies sont caractérisés par le fait qu'ils supportent (1) la multi-instanciation, (2) les propriétés multi-valuées et (3) le fait qu'une instance initialise des propriétés non définies dans le contexte de sa classe.

Notons également que ces approches de représentation des données à base ontologique sous forme *éclatée* peuvent être efficaces pour les requêtes de mises à jour, où les ontologies utilisées ne subissent pas des évolutions constantes (l'ajout d'une nouvelle propriété, la suppression d'une propriété).

Nous présentons dans la section suivante, les principaux systèmes de gestions des ontologies et leurs données, existants dans la littérature, en mettant en œuvre les schémas que nous venons de présenter.

3.3 Architectures existantes de bases de données à base ontologique

Dans la présentation de chacun de ces systèmes, nous dégagerons l'approche de représentation des ontologies et données à base ontologique qu'ils ont adoptées leurs caractéristiques particulières et les différentes fonctionnalités qu'ils proposent.

3.3.1 Architecture *Sesame*

Sesame [30], développé par Aidministrator Nederland b.v. [8] fait partie du projet Européen IST On-To-Knowledge [57], est une architecture de base de données à base ontologique pour le stockage et l'interrogation des données et des méta-données RDF [98] et RDFS[28]. Sésame est une architecture générique, c'est-à-dire, qu'elle a été modélisée et implémentée indépendamment de tout système de base de données particulier (relationnel, relationnel objet, stockage sous forme de triplets). En d'autres termes, *Sesame* peut être greffée sur chacun de ces systèmes sans pour autant changer les modules fonctionnels et le moteur de requêtes. Les principaux modules de *Sesame* sont les suivants.

1. Le module "*RDF SAIL*" (RDF Storage And Inference Layer) est le module de *Sesame* qui permet (1) de rendre persistant les ontologies et données à base ontologique dans des bases de données et (2) d'offrir une interface de programmation pour rendre indépendant l'accès aux bases de données. Cette couche est implémentée spécifiquement pour chaque système de base de données. *Sesame* propose deux implémentations de cette couche. Chaque

[8] http ://www.aidministrator.nl/

implémentation est basée sur un schéma de base de données spécifique (*PostgreSQLSAIL* et *MySQLSAIL*) :

(a) Le schéma *PostgreSQLSAIL*, basé sur PostgreSQL[9] [50], tire toutes les possibilités et avantages du système PostgreSQL, notamment, l'héritage des tables et la gestion des tableaux [60, 52]. Le schéma de *PostgreSQLSAIL* adopte le schéma de données de l'approche de représentation des ontologies *spécifique*. Pour les données à base ontologique, l'approche *ISA* pour les instances des classes et des propriétés et l'approche de représentation *binaire* pour les valeurs des propriétés (cf. figures 2.7 et 2.9).

(b) Le schéma *MySQLSAIL*, basé sur le MySQL[10], a été défini par *Sesame* suite à certains inconvénients de la première implémentation sur PostgreSQL, notamment la lenteur lors du chargement dans le système de nouvelles données et schéma (classes), la non-portabilité des données sur d'autres SGBD et la complexité de mise à jour du schéma de la partie *données* lors de la modification de la hiérarchie des classes ou des propriétés de l'ontologie stockée dans la partie *ontologie*. Le schéma de *MySQLSAIL* est entièrement basée sur la norme SQL92. Le schéma de MySQLSAIL adopte le schéma de l'approche de représentation *spécifique* pour la représentation des ontologies et l'approche de représentation *hybride* avec la variante *ID* pour les données à base ontologique (cf. figures 2.12 et 2.8).

2. Le module "*query module*" est la couche où a été implémentée le moteur de requêtes du langage RQL (RDF Query Language)[29]. RQL est un langage de requêtes proposé par ICS-FORTH (cf. section 3.3.2)[90]. RQL a une syntaxe très proche du langage OQL [34] et permet de faire des requêtes aussi bien sur les classes et les propriétés (requêtes dites ontologiques) que sur les instances (requêtes dites de "données").

3. Les modules "*Export module*" et "*Admin module*" permettent respectivement d'exporter les instances de la base de données en format XML-RDF et d'importer dans la base de données des ontologies et données définies en XML-RDF. L'objectif de ce module est de permettre l'échange avec d'autres outils basés sur RDF.

3.3.2 ICS-FORTH RDFSuite

RDF Suite [6] développé par ICS-FORTH[11]a été partiellement supporté par les projets Européen C-Web[12], MesMuses[13] et QUESTION-HOW[14]. RDF Suite est constitué d'un ensemble de module pour la gestion (stockage, interrogation, validation, échange) de grand volume de données RDF dans des SGBDs Relationnels ou Relationnels Objets. Ces principaux modules sont :

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

[9] www.prosgresql.org
[10] www.mysql.com
[11] http ://www.ics.forth.gr/
[12] http ://cweb.inria.fr/
[13] http ://cweb.inria.fr/Projects/Mesmuses
[14] http ://www.w3.org/2001/qh/

– VRP (Validating RDF Parser) : pour la validation des données (documents RDF).

RDFSuite propose deux schémas pour représenter les ontologies et les données à base ontologique.

– Le premier schéma, implémenté sous PostgreSQL, adopte l'approche de représentation *spécifique* pour le stockage des ontologies et l'approche de représentation *binaire* pour les valeurs de propriétés et l'approche *ISA* avec héritage des tables des classes et propriétés pour les instances.
– Le deuxième schéma adopte de l'approche de représentation *spécifique* pour les ontologies. Les instances des classes sont représentées dans l'approche à une unique table à deux colonnes (*id*, *classid*) et les valeurs de propriétés dans l'approche de représentation *binaire* (cf. figures 2.7 et 2.9).

Il faut préciser que *RDFSuite* exige trois contraintes [6] 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. Cette exigence provient du fait qu'en OWL et RDF Schéma, une propriété peut exister sans domaine et/ou co-domaine. Sans cette contrainte, il y a des risques d'ambiguïté lors de l'héritage de propriétés. Car, en effet, une sous-propriété spécialise normalement la définition de sa superclasse. En OWL et RDF Schéma, lorsque le co-domaine d'une propriété n'est pas défini, par défaut, elle aura pour co-domaine l'union des classes et des types littéraux, et cette union est dépourvue de sens [6].
2. Le domaine et le co-domaine d'une propriété doivent être uniques. Cette exigence vient toujours du fait qu'en OWL et RDF Schema, on peut déclarer plusieurs domaines et co-domaine d'une même propriété. Sans cette exigence, d'une part, il y aurait une ambiguïté dans le choix du type des colonnes des tables des propriétés, et d'autre part, RDF Schéma ne supporte l'union de classes [6].
3. Enfin, la définition des classes et propriétés doivent être complètes.

3.3.3 Jena Architecture

Jena est un framework open-source[15] et initié au départ par les laboratoires HP. *Jena* est constitué d'un ensemble d'outils implémenté en Java qui offre :

– une API de programmation pour la gestion des données (RDF, RDFS, DAML+OIL et OWL) des applications de Web sémantique,
– un langage de requêtes AQL qui est une implémentation du langage SPARQL [133],
– une structure relationnelle pour un stockage persistent de données RDF, RDFS, DAML+OIL et OWL,
– un parseur RDF/XML,
– un moteur d'inférence couplé aux moteurs d'inférence Racer [160], Fact [71] et Java Theorem Prover [59]),

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

– des outils pour faire migrer des instances RDF d'un format à un autre.

La version courante de *Jena* est *Jena2* [165] et précède *Jena1* [23]. Jena est intégré dans l'outil *protégé2000* qui est un éditeur d'ontologie et de données à base ontologique. Dans chacune de ces versions des schémas différents de bases de données ont été proposés. Dans *Jena1*, tant les ontologies et les données à base ontologique sont stockées dans le schéma de l'approche de représentation *verticale* (cf. figure 2.8 - approche avec URI). Dans *Jena2*, l'approche utilisée est la deuxième variante de l'approche *verticale* avec ID. Les ressources et les valeurs des propriétés de type littérale (types simples) sont stockées séparément dans les tables *ressources* et *litteral* (cf. figure 2.8 - approche avec ID).

3.3.4 Le système *kwOWLer*

kwOWLer [38] est un système de base de données à base ontologique développé par l'Université de Suisse. *kwOWLer* est une version remise à jour du système Parka [150] initialement implémenté pour les systèmes médicaux pour gérer les problèmes de facturation telle que la validation des factures ou l'analyse de la neutralité des coûts des protections médicales *health-care*.

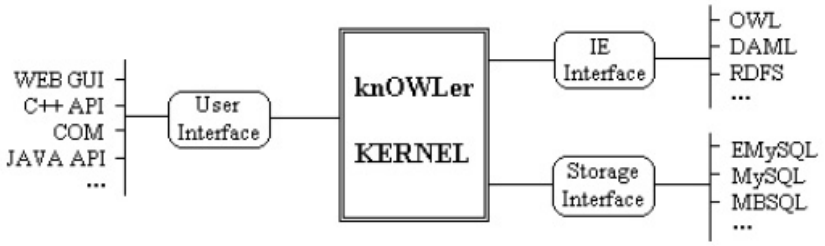


FIG. 2.13 – Architecture du système *knOWLer*

L'architecture du système *knOWLer* est présentée dans la figure 2.13. Elle est composée de quatre modules :

– le module *kernel* est la partie centrale de l'architecture qui offre les fonctionnalités nécessaires pour la manipulation de la structure et de la sémantique définie par une ontologie comme faire du raisonnement sur les ontologies stockées ou exécuter des requêtes RDQL (RDF Query Language). Précisons que *knOWLer* supporte la représentation en natif des ontologies de types OWL Lite.
– Le module *Import/Export* permet l'importation dans la base de données et l'extraction des ontologies de différents modèles ontologies.
– Le module *Storage* permet le stockage des ontologies (RDF Schema, DAML+OIL, OWL) et les données à base ontologique de façon persistante dans des fichiers classiques ou soit dans un SGBD relationnel. Comme le système *RDFSuite*, *knOWLer* opte pour une représentation séparée des ontologies et des données à base ontologique. Les ontologies sont stockées dans des tables spécifiques (class, subclass, property, subproperty, etc.). Les données à base ontologique sont stockées dans une approche de représentation *binaire*.

– Le module *User* qui offre un ensemble d'APIs (Java, C et C++) et d'applications graphiques pour l'accès aux fonctionnalités du système.

Les systèmes de bases de données à base ontologique que nous venons de décrire constituent un réel progrès de la persistance des ontologies et des données à base ontologique dans les bases de données relationnelles. Nous résumons dans la sous-section suivante, les caractéristiques et fonctionnalités communes à tous ces systèmes que nous mettrons en œuvre également dans le futur système que nous proposerons dans le chapitre 4 pour notre architecture de BDBO.

3.3.5 Caractéristiques et fonctionnalités des systèmes de gestions des bases de données à base ontologique

L'étude de tous ces systèmes, permettent de dégager les fonctionnalités et caractéristiques suivantes :

1. **Respect d'un standard.** Les systèmes de gestion de BDBOs sont tous basés sur un ou plusieurs langages de définition d'ontologies principalement : RDF, RDFS, DAML+OIL, OWL.

2. **Échange de données.** Les systèmes de gestions de BDBOs offrent la possibilité d'importer dans la base de données des ontologies et des données à base ontologie définies dans l'un des langages du point précédent. Et inversement, ils permettent d'extraire un sous-ensemble des ontologies et/ou un sous-ensemble des populations d'instances dans un langage quelconque. Cette fonctionnalité permet le partage des données entre des systèmes.

3. **Langage de requêtes.** Les systèmes de gestion des BDBOs offrent des langages de requêtes (RQL ou SPARQL) qui permettent (1) la création des concepts (classes, propriétés, etc.), (2) la création d'instances des concepts et (3) l'interrogation pour retrouver des concepts ou/et des instances.

4. **API d'accès aux ontologies et leurs données**. Les systèmes de gestion des BDBOs offrent des interfaces de programmations pour l'accès aux ontologies et leurs données représentées dans les bases de données. Ceci en vue d'abstraire les schémas de données sous-jacents pour les applications accèdent aux données, notamment les moteurs d'inférences.

Notons toutefois qu'il existe d'autres fonctionnalités et caractéristiques *souhaitables* auxquelles ces systèmes ne supportent malheureusement pas et utile pour les applications des domaines qui nous intéressent particulièrement. Ces caractéristiques et fonctionnalités sont entre autres :

1. **Notion de schéma.** Tous ces systèmes, que nous avons présentés, représentent les données à base ontologique dans l'une des approches que nous avons présentées sous forme de *verticale* ou sous forme *binaire*. Une instance donnée dans ces systèmes ayant un certain nombre de propriétés est décomposée en plusieurs tuples dans différentes tables (dans

l'approche *binaire*) ou dans la grande table *triples* (dans l'approche verticale). L'accès à une instance nécessitera donc de faire de nombreuse jointures (ou auto-jointure) qui sont coûteuses. La notion de schéma de données au sens usuel des bases de données classiques n'est pas présente. Les schémas de données sont importants car ils permettent de traiter de façon rapide de très gros volume de données, car ils offrent la possibilité d'appliquer les différents mécanismes (indexation, regroupement (clustering), etc.) offerts par les SGBDs pour l'optimisation des requêtes.

2. **La gestion des versions et des évolutions des ontologies** est fonctionnalité nécessaire pour un certain types d'applications utilisant des ontologies susceptibles d'évoluer. En effet, les ontologies évoluent, ce qui occasionne la définition de nouvelles versions de ces ontologies. Particulièrement, pour les ontologies des domaines techniques où leur processus de normalisation peut prendre plusieurs années, des versions préliminaires sont définies progressivement jusqu'à la version finale. Leur réintégration dans les BDBOs doit être faite de sorte à la fois de préserver les relations sémantiques existantes entre les différents concepts des ontologies. Il est nécessaire d'offrir également des primitives pour accéder et manipuler des versions quelconques des concepts.

3. **Cycle de vie des instances.** Comme les ontologies, les données à base ontologique évoluent également. Deux types d'évolutions peuvent se produire. Le premier type d'évolution des données à base ontologique est directement lié à l'évolution des ontologies. En effet, d'une version à une autre de concepts des ontologies (les classes exemples), des modifications surviennent systématiquement sur la structure des nouvelles instances de ces concepts. Le deuxième type d'évolution est lié aux opérations (l'ajout, la suppression, la mise à jour) sur les instances, leurs propriétés ou leur schéma qui peuvent être appliquées sur les données à base ontologique. Pour les applications n'utilisant pas les mêmes versions des concepts d'une ontologie, il est souhaitable d'offrir d'une part, un mécanisme pour pouvoir accéder à toutes les instances des concepts existantes dans une version donnée dans la BDBO et d'autre part des mécanismes pour une gestion du cycle de vie des instances des classes. On devrait pouvoir interroger la BDBO pour retrouver entre autres :
   - la version à laquelle une instance d'une classe a été insérée dans la base de données ou supprimée,
   - les différentes propriétés utilisées d'une classe à une certaine version, vu que d'une version à une autre, les classes peuvent initialiser de nouvelles propriétés ou en supprimées d'autres,
   - les systèmes devraient pouvoir permettre de déterminer les versions auxquelles une propriété donnée a été initialisée ou supprimée.

Notons que la mise en œuvre de la plupart de ces fonctions exige à ce qu'elle soit prise en compte directement au niveau des schémas de tables où sont stockées les ontologies et les données à base ontologique. Dans l'approche que nous proposons dans le chapitre suivant, nous considérerons ces caractéristiques et fonctionnalités comme des exigences que nous prendrons en compte dans les schémas de représentation des ontologies et de leurs données.

4 Conclusion

Nous avons fait dans ce chapitre un état de l'art sur les ontologies, sur la modélisation à base ontologique et la représentation des ontologies et des données à base ontologique dans des bases de données.

Nous avons d'abord décrit les caractéristiques et les composants des langages de définition d'ontologies. Cette description nous a permis d'identifier un noyau commun, en termes de caractéristiques, aux divers langages de définition d'ontologie existants dans la littérature tels que RDF Schema, DAML+OIL, OWL, PLIB, FLogic. Ce noyau commun peut s'énoncé comme suit : *une ontologie est constituée d'un ensemble de classes organisées en hiérarchie à l'aide de la relation de subsomption. Les classes sont caractérisées par un ensemble de propriétés associées à des types de valeurs. Les propriétés peuvent avoir des co-domaines de type simple ou un type complexe telle qu'une collection ou une classe. Dans le dernier cas, la propriété établit une association entre classes. Les classes et les propriétés d'une ontologie sont associées à des caractéristiques logiques qui s'expriment sous forme de contraintes, d'axiomes et/ou de règles. Enfin, des instances, ou encore données à base ontologique, sont définies en termes des concepts, classes et propriétés, définies dans l'ontologie.*

La plupart des langages de définition d'ontologies proposent des constructions et des caractéristiques spécifiques au dessus ce noyau commun. L'analyse de ces constructions et caractéristiques, nous a permis de proposer une classification de ces langages suivant qu'ils sont orientés vers une caractérisation des objets de l'univers du discours ou qu'ils sont orientés vers l'inférence. Les ontologies de la première catégorie sont canoniques dans le sens où les concepts décrits ne sont pas redondants, et permettent donc une représentation unique de chaque objet appartenant à l'univers du discours. De telles ontologies ne comportent, en particulier, que des classes *primitives*. Les ontologies appartenant à la deuxième catégorie se distinguent par le fait qu'elles proposent des constructions permettant de décrire des équivalences conceptuelles, que ce soit des équivalences de classes, en utilisant des expression de classes (UNION, DIFFERENCE, etc.) qui permettent de définir des classes non canoniques, également appelées classes *définies*, ou que ce soit en définissant des équivalences de propriétés. Le schéma d'une base de données étant toujours fondamentalement canonique, c'est aux ontologies canoniques que nous nous intéresserons dans le chapitre prochain.

Nous avons montré alors les avantages que présente l'utilisation d'ontologies pour la modélisation de l'information par rapport aux modèles conceptuels classiques. A la différence des modèles conceptuels toujours très dépendants du contexte applicatif particulier dans lequel ils sont conçus, les ontologies permettent, en effet, de conceptualiser de façon explicite et consensuelle, l'ensemble des concepts d'un univers. Le caractère consensuel des ontologies leur permet d'être partagées entre différents experts du domaine et d'offrir ainsi un vocabulaire commun. Leur utilisation dans la conception de bases de données leur permet donc de réduire l'hétérogénéité des bases de données et en conséquence de faciliter l'intégration de celles-ci débouchant même sur des méthodes d'intégration automatiques ou semi-automatiques des bases de données.

Concernant la représentation des ontologies et des données à base ontologique, nous avons défini le concept de *base de données à base ontologique* qui permet de représenter dans une unique base de données, des ontologies, des données ainsi que les liens qui les unissent. Nous avons présenté les principaux schémas de représentation et les systèmes de gestion proposés dans la littérature pour le stockage simultané d'ontologies et des données à base ontologique. Concernant les ontologies, elles sont représentées (1) soit dans un schéma ayant une unique table nommée *triples* possédant trois colonnes (sujet, prédicat, objet), (2) soit dans un schéma de données constitué de tables spécifiques au modèle d'ontologie utilisé. Les données à base ontologique, instances des classes des ontologies, sont représentées de façon *éclatée*, soit (1) dans une grande table *triples*, soit (2) sous forme binaire dans des tables spécifiques définies pour chaque propriété des classes de l'ontologie. Ces choix de stockage qui reviennent à décomposer chaque instance en la séparant de ses valeurs de propriété sont dictés par la souplesse de représentation des données à base ontologique que permettent les langages de définition d'ontologies. Les instances dans ces langages ne sont pas fortement typées. Elles peuvent appartenir à un nombre quelconque de classes et peuvent même initialiser des propriétés non définies dans le contexte de leur(s) classe(s).

Or, dans beaucoup de domaines et en particulier dans le domaine technique, qui est notre domaine cible, les ontologies sont fondamentalement canoniques pour éviter les ambiguïtés, et les objets du domaine sont caractérisés par leur appartenance à une seule classe. Un objet est rarement à la fois une vis et un roulement. Par contre les objets peuvent être décrits par de nombreuses propriétés fortement typées. Les requêtes, qui consistent alors à collecter les objets d'une classe (et de ses sous-classes) ainsi que les valeurs des propriétés de ces objets, sont très coûteuses à évaluer sur des représentations éclatées. En effet, cette évaluation nécessite de réaliser de nombreuses jointures ou auto-jointures qui rendent le passage à grande échelle pratiquement impossible. Par contre, les restrictions que nous avons énoncées : le caractère canonique des ontologies, l'absence de multi-instanciation, et enfin le typage fort des propriétés, permettent d'envisager d'autres structures de représentation beaucoup plus proches des représentations usuelles des bases de données. Dans le chapitre 3, nous proposerons précisément une nouvelle approche de représentation des données à base ontologique basée sur la notion de schéma de données et les tests comparatifs réalisés au chapitre 5 confirmeront bien la beaucoup plus grande efficacité de ces propositions.

Enfin, concernant les fonctionnalités nécessaires, la description des principaux systèmes de gestion de base de données à base ontologique nous a permis d'identifier les fonctionnalités communes à tous que nous considérerons comme des exigences de notre futur système. D'autre part, nous avons également identifié des fonctionnalités et des caractéristiques utiles dans les domaines techniques et que notre futur système devrait supporter. Ce sont en particulier :

(1) la gestion du versionnement des ontologies,
(2) la gestion du cycle de vie des données à base ontologique.

Nous présentons donc dans le chapitre suivant, l'architecture *OntoDB* pour les bases de données à base ontologique que nous proposons dans le cadre de notre thèse, ainsi que les choix et les hypothèses faits pour la représentation des ontologies et des données à base ontologique

dans cette architecture.

Deuxième partie

Notre proposition d'architecture

Chapitre 3

Le modèle d'architecture de base de données à base ontologique OntoDB

Introduction

Dans ce chapitre, nous présentons d'abord le modèle d'architecture de base de données à base ontologique OntoDB (**Onto**logie **D**ata**B**ase). Nous décrivons ensuite chacune des parties qui composent notre architecture, ainsi que les principales fonctions supportées par ce modèle de base de données à base ontologique. Nous montrons enfin l'intérêt de notre approche par rapport aux approches existantes également présentées dans le chapitre précédent.

Le chapitre s'organise comme suit. Dans la section 1, nous énonçons les hypothèses sur lesquelles se base l'architecture *OntoDB* et les principaux objectifs que doivent atteindre cette architecture. Les hypothèses énoncées dans cette section définissent les contraintes que doit respecter les modèles d'ontologies, les ontologies et les données à base ontologique pour pouvoir être représentés dans une base de données conforme au modèle *OntoDB*. Ces différentes conditions nous permettent de caractériser les ontologies et les instances que nous envisageons de gérer dans des bases de données. Ceci nous permet alors de donner une définition formelle du modèle *OntoDB*, des ontologies et des données à base ontologique qui sont susceptibles d'être gérées.

Dans les sections 2 et 3, nous discutons respectivement de la représentation des ontologies et de la représentation des données à base ontologique. Pour chacune de ces deux composantes (ontologie et instances) nous présentons, dans un premier temps les exigences que doivent satisfaire leurs représentations ainsi que les problèmes liés à la mise en œuvre des objectifs que nous nous sommes fixés. Nous discutons ensuite des différentes solutions possibles. Nous définissons enfin les choix proposés pour de la définition de notre architecture.

La section 4 synthétise les différentes solutions retenues et présente de façon globale le modèle d'architecture *OntoDB* que nous proposons pour les bases de données à base ontologique. Le chapitre s'achève par une conclusion.

1 Objectifs et Hypothèses

1.1 Objectifs

Comme nous l'avons défini dans le chapitre précédent, un système de gestion de BDBOs doit permettre la représentation des ontologies, des données et le lien entre eux de sorte à pouvoir accéder à l'une des parties à partir de l'autre. Ontologies et données doivent toutes deux être représentées dans une unique base de données et doivent pouvoir faire l'objet des mêmes traitements (insertion, mise à jour, interrogation, versionnement, etc.). En plus de ces exigences, nous souhaitons que notre système puisse atteindre les quatre objectifs supplémentaires ci-dessous.

O_1- Nous souhaitons pouvoir intégrer automatiquement et gérer de façon homogène, des populations d'instances dont les données, le schéma et les ontologies sont chargés dynamiquement.

O_2- Nous souhaitons que notre système puisse s'adapter aisément (1) à des évolutions du modèle d'ontologie utilisé et (2) au changement de modèle d'ontologie. L'adaptation (1) est particulièrement importante pour le modèle d'ontologie PLIB utilisé dans le cadre de notre thèse qui était en cours normalisation à l'ISO. Ce modèle subissait en moyenne tous les 6 mois des modifications. L'adaptation (2) visait, en particulier, la possibilité de représentater des ontologies basées sur les logiques de description (exemple *OWL*). Cette modification devrait être possible par une simple extension du schéma logique de la base de données.

O_3- Nous souhaitons offrir des accès génériques tant aux ontologies qu'aux instances, en permettant à l'utilisateur de faire abstraction de l'implémentation particulière sur un SGBD particulier (relationnel, relationnel objet ou objet par exemple).

O_4- Nous souhaitons que notre système puisse permettre de gérer l'évolution des ontologies et fournissent des mécanismes permettant la gestion du cycle de vie des instances.

1.2 Hypothèses

Compte tenue de la grande diversité des modèles d'ontologies existants, il est peu vraisemblable de trouver un modèle d'architecture de BDBO qui s'avère efficace dans tous les contextes. Nous définissons donc à la lueur de l'application que nous visons à traiter un certain nombre de restrictions. Cette section présente les hypothèses que doivent remplir (1) les modèles d'ontologies qui peuvent être utilisés pour la mise en œuvre de notre architecture *OntoDB* et (2) les ontologies et les données à base ontologiques qui peuvent être représentées dans une base de données *OntoDB*.

H_1 **-** Nous reprenons tout d'abord les exigences de l'architecture RDFSuite [6] (déjà énoncées dans la section 3.3.2 du chapitre 2) :

R_1 **-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.

R_2 **-Complétude de définition.**

- les descriptions complètes de tous les concepts qui contribuent à la définition d'un concept doivent pouvoir être représentées dans le même environnement (e.g., fichier)

que celui où ce concept est défini. Ceci suppose en particulier que les super-classes d'une classe, si elles existent, soient connues, de même que les domaines et le co-domaine de ses propriétés.

H_2 - Les ontologies considérées sont celles qui peuvent s'exprimer sous forme de modèle, au sens de Bernstein [21, 20], c'est-à-dire d'un ensemble d'objets accessibles à partir d'un objet racine, et qui décrit un univers sous forme de classes et de propriétés.

H_3 - Les données considérées sont celles qui représentent des instances des classes de l'ontologie (aussi appelées par certains auteurs : *individus*).

H_4 - On exige que ces instances respectent l'hypothèse de *mono-instanciation* définie par la règle suivante :

R_1 **-Mono-instanciation**

– toute instance du domaine 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 subsomption de l'ensemble des classes auquel elle appartient (elle appartient bien sûr également à ses super-classes).

Notons que de l'hypothèse $H_1(R_1)$ et $H_4(R_1)$, on peut déduire la règle R_2 : *typage fort des instances.*

R_2 **-Typage fort des instances**

– 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.

H_5 - Enfin, pour permettre la gestion automatique des différentes versions d'une même ontologie, on exige que l'évolution des ontologies obéisse au principe de continuité ontologique [167, 166]. Cette hypothèse sera discutée dans la section 2.4.1.1.

Principe de continuité ontologique : *si l'on considère chaque ontologie intervenant dans le système de base de données à base ontologique comme un ensemble d'axiomes, alors tout axiome vrai pour une certaine version de l'ontologie restera vrai pour toutes les versions ultérieures.*

Avant de décrire l'architecture que nous avons proposée, nous définissons formellement dans un premier temps, les ontologies et le concept de base de données à base ontologique. Nous utiliserons ces notations formelles tout au long du chapitre pour illustrer nos exemples et certains algorithmes.

1.3 Formalisation

Ces hypothèses permettent alors de définir de façon formelle à la fois les ontologies et les données à base ontologique que l'architecture *OntoDB* doit permettre de gérer. Les ontologies gérées par *OntoDB* peuvent être définies par un quadruplet [127] :

$O :< C, P, Sub, Applic >$, avec :

– C : l'ensemble des classes utilisées pour décrire les concepts d'un domaine donné. Chaque classe est associée à un identifiant universel unique ;

– P : l'ensemble des propriétés utilisées pour décrire les instances de l'ensemble des classes C. Nous supposons que P définit toutes les propriétés consensuelles dans le domaine. Chaque

propriété est associée à un identifiant globalement unique ;
- *Sub* est la relation de subsomption de signature *Sub* : C→ 2^C, qui à chaque classe c_i de l'ontologie, associe ses classes subsumées directes.
- *Applic* est une fonction de signature *Applic* : C→ 2^P, qui associe à chaque classe de l'ontologie les propriétés qui sont applicables pour chaque instance de cette classe. Seules les propriétés ayant pour domaine une classe ou l'une de ses super-classes peuvent être applicables pour décrire les instances de cette classe.

Notons que cette définition formelle, est en particulier, respectée par les ontologies PLIB, la relation *Sub* étant l'union de *is-a* et *is-case-of*. Soulignons qu'une telle ontologie n'est pas un modèle conceptuel. Le fait qu'une propriété soit applicable pour une classe signifie qu'elle est rigide [66, 126], c'est-à-dire que chaque instance de cette classe devra posséder une caractéristique correspondant à cette propriété. Cela ne signifie pas que la valeur d'une telle propriété devra être explicitement représentée pour chaque représentation d'une telle instance dans la base de données. Dans les données à base ontologique que nous voulons gérer (cf. hypothèse H_4-R_2), le choix parmi les propriétés applicables d'une classe des propriétés effectivement utilisées pour ses instances est fait au niveau du schéma.

Une base de données à base ontologique (BDBO) peut alors être définie formellement comme un quadruplet [127] : BDBO $:< O, I, Pop, Sch >$, avec :
- O représente son ontologie ($O :< C, P, Sub, Applic >$).
- I représente l'ensemble des instances de la base de données. La sémantique de ces instances est décrite par O en les associant à des classes et en les décrivant par les valeurs de propriétés applicables définies dans l'ontologie,
- Pop : $C \rightarrow 2^I$, associe à chaque classe les instances qui lui appartiennent (directement où par l'intermédiaire des classes qu'elle subsume). $Pop(c_i)$ constitue donc la population de c_i.
- Sch : $C \rightarrow 2^P$, associe à chaque classe c_i les propriétés qui sont applicables pour cette classe et qui sont effectivement utilisées pour décrire tout ou partie des instances de $Pop(c_i)$. Pour toute classe c_i, $Sch(c_i)$ doit satisfaire : $Sch(c_i) \subset Applic(c_i)$.

Nous discutons dans les sections suivantes des principales fonctions que doit disposer un système de BDBO. Leur analyse nous permettra de décrire avec précision les différents éléments du modèle d'architecture que nous proposons. Nous discuterons de ces fonctions séparément pour (1)la représentation des ontologies dans section 2 et (2) la représentation des données à base ontologique dans la section 3.

2 Analyse de besoins et propositions pour la représentation des ontologies

Nous avons vu (hypothèse H_2) que les ontologies auxquelles nous nous intéressons sont celles susceptibles d'être représentées sous forme d'un modèle au sens de Bernstein [21], c'est-à-dire d'un ensemble d'objets accessibles à partir d'un objet racine par des relations de composition et

qui décrivent un univers sous forme de classes et de propriétés. Cette définition correspond à la plupart des modèles d'ontologies récents tels que OWL [44], PLIB [164, 80, 126]. Une telle ontologie est donc représentée comme une instance d'un schéma objet (souvent appelé méta-modèle) dans un formalisme de modélisation particulier (XML-Schema pour OIL et OWL, EXPRESS pour PLIB). Cette représentation, à son tour, fournit à la fois un modèle conceptuel et un format d'échange pour les ontologies visées (document XML pour OWL, fichier physique d'instances EXPRESS pour PLIB).

Pour définir notre modèle, nous commençons par analyser les besoins que nous souhaitons qu'il satisfasse. Ces exigences sont définies sous forme d'un ensemble de fonctions devant être supportées par notre système. L'analyse de ces fonctions va alors nous permettre de discuter et de décrire avec précision les différents éléments ou composantes du modèle d'architecture nécessaire.

- **F1 : capacité de stockage interne des ontologies au sein d'un schéma logique adapté au SGBD cible**. Celui-ci doit pouvoir être de type relationnel, relationnel-objet ou orienté objet.

- **F2 : interface générique d'accès par programme aux entités définissant une ontologie**. Cette interface est dite *générique* parce qu'elle doit être définie indépendamment de tout modèle d'ontologie particulier, ceci pour permettre que notre architecture de BDBO puisse supporter la représentation d'ontologies de modèles évolutifs ou issus de modèles différents.

- **F3 : interface spécifique d'accès par programmation orienté-objets aux entités de l'ontologie**. Il s'agit ici de proposer une interface qui soit (1) orientée objet, (2) spécifique d'un langage de programmation particulier et (3) spécifique d'un modèle d'ontologie donné (par exemple OWL, PLIB, RDF Schéma ou DAML+OIL) mais qui permet de contrôler la correction syntaxique de tous les accès aux entités définissant une entité. Le caractère spécifique de cette API est liée au fait que les ontologies sont des instances de modèles objets particulier et bien définis, et que les fonctions de l'API sont définies de façon à retourner les données selon la structure *objet* spécifique de ces modèles.

- **F4 : capacité de lire les ontologies représentées dans leur format d'échange et de les stocker dans la BDBO**. Notons que lorsque les ontologies et leurs instances sont susceptibles d'évoluer, cette fonction est particulièrement délicate car elle doit permettre de gérer l'évolution des ontologies. Les concepts et les instances des ontologies pouvant être utilisés par des applications dans différentes versions, il est indispensable que la BDBO possède des mécanismes pour garder une trace de toutes les versions des concepts successives de façon à assurer une *compatibilité ascendante* pour les applications.

- **F5 : capacité d'exporter des concepts des ontologies avec éventuellement leurs instances**. Cette fonction est duale de la précédente fonction de lecture. Il s'agit ici d'être capable d'extraire de la BDBO un sous-ensemble des concepts de l'ontologie (dans une cer-

taine version) ainsi qu'un sous-ensemble des instances associées à ces concepts. Les concepts extraits de la base de données doivent être cohérents, i.e., leurs définitions doivent pouvoir être complètes (cf. hypothèses H_2). Un concept est dit *complet* lorsque tous les éléments qui participent à sa définition sont présents dans le référentiel où il se trouve. Par exemple, pour une classe C, on doit trouver dans son référentiel, la définition de ses super-classes, de ses propriétés applicables et non uniquement leur identifiant (URI, BSU). Si l'utilisateur le souhaite, la fonction d'extraction doit permettre d'extraire *sémantiquement* les concepts et d'analyser toutes les dépendances de sorte à les extraire également et de façon récursive.

Nous discutons successivement dans les sous-sections suivantes la problématique associée à chacune de ces fonctions et les choix que nous avons faits. Ceux-ci nous ayant conduit à notre proposition d'architecture de base de données à base ontologique.

2.1 F1 : Capacité de stockage interne des ontologies au sein d'un schéma logique adapté au SGBD cible

Les modèles d'ontologies considérés étant des modèles objets, le choix d'une représentation correspond au problème classique de la représentation de modèles objets au sein d'une base de données qui peut ne pas être de type objet comme nous avons discuté dans le chapitre 1. Selon que le SGBD cible est de type objet, relationnel ou relationnel objet, la solution est plus ou moins naturelle.

Le principe général de toutes les approches [37, 134, 152, 110], que nous avons présentées dans le chapitre 1, consiste à définir des règles de correspondance entre les mécanismes du formalisme de modélisation objet considéré et les concepts du SGBD cible et ensuite à appliquer ces règles sur le modèle particulier à traiter pour déduire le schéma de base de données où seront stockées les instances de ce modèle. Ces techniques de correspondance objet-relationnel ont montré leur efficacité [70, 132, 94]. On peut donc utiliser ces techniques pour la définition de schéma interne de stockage des ontologies vu que les modèles d'ontologies sont des modèles objets définis dans un langage de représentation objet (UML [83], XML [26], EXPRESS [79]). On trouve dans la littérature des propositions de correspondances entre les mécanismes objets et le formalisme relationnel pour UML [3] et XML [26]. A défaut, des règles de correspondances doivent être définies. C'est ce que nous avons fait pour le langage EXPRESS utilisé pour définir le modèle d'ontologie PLIB. Nous présentons les règles de correspondance que nous proposons dans l'annexe C. De plus, pour illustrer le principe de transformations effectuées, nous présentons dans l'exemple suivant une version simplifiée de nos transformations de modèles en utilisant le formalisme objet UML.

Exemple 1 :
La figure 3.1b donne un exemple simplifié de modèle d'ontologie représenté en UML. La figure 3.1a présente un exemple d'une ontologie particulière conforme à ce modèle (cette ontologie est aussi représentée en UML). La figure 3.1c représente sous forme d'instances l'ontologie 3.1a dans un schéma de tables générées à partir du modèle d'ontologie 3.1b en utilisant les règles de correspondance suivantes.

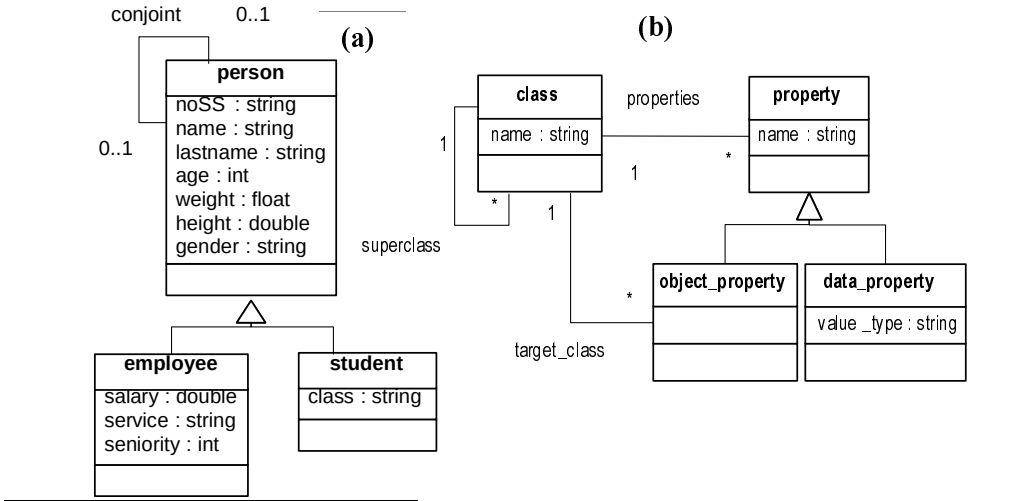
1. L'approche de représentation des relations d'héritage entre classes est l'approche verticale (cf. chapitre 1 section 3.4.1) : une classe par classe concrète. Ce qui explique le fait qu'on n'ait pas de table pour la classe *property* du modèle d'ontologie.
2. Chaque attribut est traduit par une colonne dans la table associée à sa classe.
3. Les relations de cardinalité 1 : N entre deux classes (*composant-composite*) sont traduites par une colonne (de nom : inv+NomDeLaRelation) dans la table de la classe *composant*. Exemple : *invProperties*.

Ainsi défini, le schéma logique doit être connu et accédé directement à partir du système de gestion de la BDBO. Le système de gestion de BDBO n'est donc pas indépendant du modèle d'ontologies. Pour atteindre l'objectif (O_2) et permettre au système de gestion de BDBO de s'adapter aux évolutions du modèle d'ontologie ou aux changements de modèles d'ontologies, il va falloir rendre générique le système de gestion de BDBO par rapport au modèle d'ontologie utilisé.

Rendre générique une application par rapport aux données conformes à un ensemble de modèles exige que l'on puisse (1) définir un méta-modèle tel que tout modèle envisageable puisse être représenté et accédé comme instance de ce méta-modèle et (2) concevoir une interface d'accès aux données qui puisse être définie indépendamment de tout modèle particulier.

Les modèles d'ontologies qui nous intéressent s'expriment tous dans un formalisme qui dispose de son propre méta-modèle. L'utilisation de ce méta-modèle permettra donc de représenter toutes les modifications possibles du modèle d'ontologie considéré, à condition que celui-ci continue à s'exprimer dans le même formalisme (par exemple XML-Schema pour RDF Schema et OWL ; EXPRESS pour PLIB).

Dans l'approche que nous proposons, le schéma de données permettant le stockage des ontologies est généré automatiquement en définissant des correspondances entre les mécanismes objets du formalisme utilisé pour définir le modèle d'ontologie et le SGBD cible. De plus, pour atteindre l'objectif (O_2) que nous nous sommes fixés, nous proposons de représenter le modèle d'ontologie utilisé au sein de la BDBO sous forme d'instances d'un méta-modèle. Cette représentation du modèle d'ontologie offrira la possibilité de définir des interfaces de programmation permettant d'accéder aux ontologies de façon indépendante du modèle particulier d'ontologie. Nous présentons dans la section suivante la réalisation d'une telle interface. Dans la section 2.6, nous discuterons de la représentation du modèle d'ontologie dans la base de données.



class			
OID	**name**	**superclass**	**…**
id_person	person		
id_student	student	id_person	
id_employee	employee	id_person	

object_property				
OID	**name**	**invproperties**	**target_class**	**…**
id_conjoint	conjoint	id_person	id_person	

(c)

data_property				
OID	**name**	**invproperties**	**value_type**	**…**
id_noSS	noSS	id_person	String	
id_name	name	id_person	String	
id_lastname	lastname	id_person	String	
id_age	age	id_person	Int	
id_weight	weight	id_person	Float	
id_height	height	id_person	Float	
id_gender	gender	id_person	char	
id_service	service	id_empoyee	String	
id_salary	salary	id_empoyee	Float	
id_seniority	seniority	id_empoyee	Int	
id_class	class	id_student	String	

FIG. 3.1 – Exemple d'ontologie (a) de modèle d'ontologie simplifié (b) et de représentation de l'ontologie sous forme d'instances dans le schéma de base de données issu du modèle d'ontologie (c).

2.2 F2 : Interface générique d'accès par programme aux entités définissant une ontologie

Pour atteindre l'objectif (O_2), le système de gestion de BDBO doit fournir une interface générique d'accès aux entités de chaque ontologie et ce, indépendamment du schéma logique de la base de données et du modèle d'ontologie. Une telle interface générique est usuellement appelée "API à liaison différée" [131]. L'exemple ci-dessous, présente ce que pourrait être la syntaxe d'une telle API indépendante du modèle d'ontologie particulier. On discute, ensuite, comment une telle API pourrait être programmée effectivement et indépendamment du modèle d'ontologie, grâce à la représentation du modèle d'ontologie sous forme d'instances d'un méta-modèle comme proposée en conclusion de l'étude de la fonction F_1.

Les fonctions d'une API générique par rapport au modèle d'ontologie pourraient, par exemple, se définir comme dans l'algorithme 1.

La programmation effective des fonctions de l'algorithme 1 nécessite alors soit (1) de représenter à l'intérieur de chaque fonction la méthode de calcul de son résultat *pour chacun de ses paramètres possibles*, soit (2) de représenter le modèle d'ontologie lui-même à l'intérieur de la base de données, et d'exploiter cette représentation dans la programmation des fonctions de l'algorithme 1.

Par exemple, dans le premier cas, pour interpréter la fonction *get_value_attribut(id_person, 'class', 'properties')*, la solution serait de représenter dans la fonction le code de l'algorithme 2.

On devrait donc coder ainsi *tous* les noms d'attributs et *tous* les noms d'entités du modèle d'ontologie dans la fonction. On imagine la difficulté de l'implémentation de cette approche lorsque le modèle d'ontologie utilisé contient de nombreuses entités et attributs et/ou lorsque le modèle d'ontologie est en constante évolution (ce qui amènerait à modifier le code à chaque fois). C'est justement le cas du modèle d'ontologie PLIB qui est constitué de 218 classes avec un total de 486 attributs. D'autre part, celui-ci évoluait, au commencement de cette thèse, environ tous les 6 mois.

Au contraire, dans le cas où le modèle d'ontologie (tel celui défini dans la figure 3.1b) est représenté explicitement dans la base de données, on pourrait exploiter cette représentation, et, avec les règles de correspondances définies qui ont permis de générer les tables de la partie ontologie, programmer la fonction *get_value_attribut* de façon générique. Ceci suppose évidemment de pouvoir, dans l'exemple de la figure 3.1, représenter le modèle d'ontologie (figure 3.1b) comme une instance d'un méta-modèle. S'agissant d'un modèle UML, il pourrait être représenté comme une instance du méta-modèle d'UML en UML. Cette fonction pourrait alors, par exemple être écrite comme dans l'algorithme 3.

Si le modèle de l'ontologie est représenté, on voit donc aisément que chacune des fonctions ci-dessus peut être programmée de façon générique en tenant compte, dans tous les cas, de la correspondance choisie (par exemple celle de l'exemple 1). Notons par ailleurs qu'une telle in-

Algorithme 1 Exemple de syntaxe de fonctions génériques par rapport au modèle d'ontologie.

En notant par :
- oid : un type chaîne dont le contenu est un identifiant d'objet pour la base de donnée ;
- nom_entité : un type chaîne dont le contenu est un nom de type d'entité ;
- nom_attribut : un type chaîne dont le contenu est un nom d'attribut[16] ;
- value : un type générique (dont le contenu est la valeur d'un attribut) ;

- **create_instance : nom_entité -> oid** :
  Cette fonction permet de créer une instance d'une entité du modèle d'ontologie (représentant donc par exemple une classe ou une propriété d'une ontologie particulière).
  Exemple avec le modèle d'ontologie 3.1b de la figure 3.1 :
  id_person = create_instance('class') ;
  id_noSS = create_instance('data_property')

- **set_value_attribute : oid x nom_entité x nom_attribut x** $value^n$:
  Cette fonction permet d'initialiser la valeur des attributs des instances des entités du modèle d'ontologie.
  Exemple :
  set_value_attribute(id_person,'class','name','personne') ;
  set_value_attribute(id_noSS,'data_property','name','no_SS') ;
  set_value_attribute(id_noSS,'data_property','value_type','integer') ;

- **get_value_attribute : oid x nom_entité x nom_attribut ->** $value^n$:
  Cette fonction permet de récupérer la valeur des attributs des instances des entités du modèle d'ontologie.
  Exemple :
  name_class = get_value_attribute(id_person,'class','name') ;
  properties = get_value_attribute(id_person,'class','properties') ;
  name_property = get_value_attribute(id_noSS,'data_property','name') ;
  domain = get_value_attribute(id_noSS,'data_property','value_type') ;

- **usedin** : oid x nom_entité x nom_attribut -> $value^n$:
  Cette fonction retourne toutes les instances qui référencent l'instance de *oid* donné en paramètre à partir de l'attribut ou relation *nom_attribut* de l'entité *nom_entité*.
  Exemple :
  id_person = usedin(id_noSS,'class','properties') ;

- **get_final_type** : oid x nom_entité -> nom_entité :
  Cette fonction retourne le type effectif d'une instance.
  Exemple :
  nom_entité = get_final_type(id_noSS, 'property') ;

Algorithme 2 Code (partiel) de la fonction *get_value_attribute* sans méta-modèle.

```
FUNCTION get_value_attribut : oid x nom_entité x nom_attribut ->valeur^n
BEGIN
...
IF nom_entité = 'class' AND nom_attribut = 'properties' THEN
  ExecQuery :
      SELECT ID FROM DATA_PROPERTY
      WHERE invproperties = oid
         UNION
      SELECT ID FROM OBJECT_PROPERTY
      WHERE invproperties = oid
IF nom_entité = X AND nom_attribut = Y THEN
   ...
```

terface peut être utilisée quelque soit le modèle d'ontologie à condition qu'il s'exprime comme instance du même méta-modèle. Toute évolution ou changement du modèle d'ontologie nécessitera simplement de mettre à jour les données représentant le modèle d'ontologie dans la base de données. Nous discuterons dans la section 2.6 de la représentation du modèle d'ontologie dans la base de données.

Nous pouvons donc conclure donc que, comme nous l'avons proposé pour la fonction F_1, représenter le (ou les) modèles d'ontologies en tant qu'instances d'un méta-modèle est également nécessaire pour réaliser la fonction F_2.

Algorithme 3 Principe d'écriture de la fonction *get_value_attribute* avec un méta-modèle.

```
FUNCTION get_value_attribut : oid x nom_entité x nom_attribut_ou_relation ->valeur^n
BEGIN
...
Quel est le type de l'attribut nom_attribut ?
S'il est de type simple (i.e., UML_data_type) ALORS ...
S'il est de type classe (i.e., UML_class) (soit C cette classe) ALORS
   Récupérer toutes les sous-classes de la classe C .
   Parcourir toutes les sous-classes de C pour générer la requête analogue à l'algorithme 2.
S'il est de type relation (i.e., UML_relation) ALORS
   Quelle est la classe cible de la relation ? (soit C cette classe)
   Récupérer toutes les sous-classes de la classe C.
   Parcourir toutes les sous-classes de C pour générer la requête analogue à l'algorithme 2.
END FUNCTION ;
```

2.3 F3 : Interface d'accès par programme orienté-objet aux entités de l'ontologie spécifique à la fois du modèle d'ontologie et du langage de programmation

L'interface de programmation générique que nous avons décrite précédemment est encore appelée *API à liaison différée*, car les fonctions qui la constituent sont définies indépendamment du modèle d'ontologie particulier. La liaison avec le modèle n'est faite que *lors de l'appel* à partir des noms des entités et attributs (e.g., 'class','properties'). Par exemple pour récupérer les propriétés applicables de la classe *person*, on écrira à l'aide de l'*API à liaison différée* comme dans l'algorithme 4.

Algorithme 4 Exemple d'utilisation de l'API à liaison différée

String[] ses_proprietés = get_value_attribut(id_person, 'class', 'properties') ;
La variable *ses_propriétés* contient alors l'identifiant *interne* des propriétés de la classe *Person* : ses_propriétés = $\{id\_name, id\_age, ...\}$;

Ce type d'interface présente néanmoins deux inconvénients majeurs :

1. Les erreurs syntaxiques sont détectées pendant l'exécution du programme. Par exemple, imaginons que lors de l'appel de la fonction précédente on ait écrit :

   String[] ses_propriétés = get_value_attribut(id_person, 'class', '**propertiies**') ;

   L'erreur survenue dans le nom de l'attribut *propertiies* (avec deux "*i*") ne pourra être décelée qu'à l'exécution, et encore cela dépend de la programmation effectuée (la fonction pourrait par exemple rendre la valeur vide sans lever d'erreurs).
2. La nature des données manipulées par les fonctions de l'API sont seulement des types simples (Entier, Caractères, ...). Dans l'exemple précédent, on pourrait souhaiter, par exemple que la fonction retourne une liste d'objets Java de type *property*, et non un tableau d'entier qui contient l'identifiant internes des propriétés de la classe *Person*.

2.3.1 API objet

Une première solution à ces inconvénients est la définition d'une *API objet* java propre à chaque modèle d'ontologie particulier. L'API objet Java est constituée d'un ensemble de classe dont chacune correspond à une entité du modèle d'ontologie et les méthodes sont constituées de constructeurs et destructeurs d'objet de l'entité considérée, et de primitives d'accès (get et set) à chacun des attributs de l'entité. L'algorithme 5 présente un exemple simplifié d'une telle API pour le modèle d'ontologie défini à la figure 3.1b et pour le langage Java.

Ce type d'API est dit *API à liaison préalable* car la syntaxe des fonctions est définie à partir d'un modèle cible particulier qui doit être connu quand l'API est définie. Toute erreur syntaxique est automatiquement décelée à la compilation du programme. Ensuite les données manipulées sont de type objet comme montré dans l'algorithme 6.

Algorithme 5 API Java à liaison préalable pour l'accès à une ontologie

```
package OWLLiteApi ;
public class Property{
private Integer oid ;// oid de la propriété ;
private String name ;// nom de la propriété ;
private String value_type ;// type de la propriété ;
void Property(){// constructeur pour crée une propriété d'une classe.
//Toutes les valeurs des attributs sont initialisées à nulles.
} ;
void Property(Integer OID){
// constructeur pour initialiser les attributs de la propriété
//Récupération/initialisation les valeurs des attributs (name, range) de la propriété
//dans la base de données.
}
public String getName(){ ... }// récupère le nom d'une propriété
public void setName(String name){...} // initialise le nom d'une propriété
public String getValue_type(){ ... }// récupère le type d'une propriété
public void setValue_type(String value_type){...} // initialise le type d'une propriété
...
}
```

Algorithme 6 Exemple d'appel d'une API à liaison préalable

```
Data_Property prop = new Data_Property(id_nom) ;// la propriété nom de la classe personne
String type = prop.getValue_type() ;// le type de données de la propriété nom
Class person = prop.getDomain() ;// le domaine de la propriété nom qui est la classe personne
```

Nous présentons ici deux aspects fondamentaux qui doivent être pris en compte pour la mise en œuvre d'une API objets.

1. **Gestion d'une mémoire cache.**
   Une API objet doit pouvoir gérer les objets dans une mémoire cache pour éviter de ré-*instancier* des objets déjà instanciés. La gestion d'une mémoire cache d'objets est particulièrement importante (1) pour l'optimisation de la gestion de la mémoire centrale, car elle permet que les objets ne soient dupliqués et (2) pour une bonne performance du système, puisque les objets une fois instanciés en mémoire sont directement accessibles. Dans une configuration client-serveur, où plusieurs applications peuvent accéder à une BDBO, le fait de disposer d'un cache est presque indispensable, car cela peut considérablement améliorer les performances des applications. Par contre, la gestion d'un cache suppose que les objets en mémoire soient synchronisés avec les données dans la base de données.

2. **Récupération des valeurs des propriétés d'un objet instancié**
   L'instanciation d'un objet nécessite un accès à la base de données pour la récupération des valeurs des propriétés de celui-ci. Un choix doit être fait. Les valeurs des propriétés des objets référençant d'autres objets ou ayant un type collection doivent-elles aussi être récupérées dans la base de données au même moment que l'instanciation de l'objet ?
   - Si la réponse à la question est **OUI**, alors toutes les valeurs des propriétés sont récupérées lors de l'instanciation des objets. L'inconvénient de cette approche est que pour les objets ayant beaucoup de propriétés de type association ou de collection, le temps d'instanciation de l'objet pourrait être très long.

   - Si la réponse à la question est **NON**, on dit, dans ce cas, qu'on est dans une configuration paresseuse -*lazy-loading*-, alors la récupération des références des objets se fera au moment de l'appel des fonctions d'accès aux propriétés. Ici, on diffère la récupération des références uniquement lorsqu'on en a besoin d'elles mais on multiplie le nombre d'accès à la base de données.

La mise en œuvre d'un cache est une tâche très complexe et nécessite beaucoup de temps de développement [109]. Pour réduire ce temps de développement, il existe dans la littérature de nombreuses boîtes à outils [51, 109, 70, 107].

2.3.2 API fonctionnelle

A défaut d'utiliser des caches d'objets, une alternative peut consister à définir une *API objet fonctionnelle*. L'idée de base de cette API est de n'instancier aucun objet. Les objets sont manipulés au moyen de leur OID. Toutes les méthodes des classes de l'API sont définies **"static"** et chaque accès nécessite une connexion directe à la base de données via l'API à liaison différée. L'algorithme 7 donne un aperçu de ce type d'interface de programmation ainsi qu'un exemple d'utilisation (algorithme 8). Dans l'exemple, *Data_Property* est une sous-classe de la classe *Property* définie dans l'algorithme 7.

Algorithme 7 API java de type fonctionnel

```
package OWLLiteApi ;
public class Property{
private String name ;//nom de la propriété ;
private String value_type ;// co-domaine de la propriété
public static String CreateInstance(){//crée une instance avec tous les attributs initialisés à nul
return create_instance('Property') ;//appel de la fonction create_instance de l'API à liaison différée.
}
public static String getName(String ID){ ... }// récupère le nom de la propriété ayant l'ID donné en paramètre ;
public static void setName(String ID, String name){...} //initialise le nom de la propriété ayant l'ID donné en paramètre.
public static String getValue_type(String ID){ // récupère le co-domaine de la propriété ayant l'ID donné en paramètre ;
return get_value_attribut(ID,'Property','value_type') ;
}
public static void setValue_type(String ID, String value){//initialise le co-domaine de la propriété ayant l'ID donné en paramètre.
set_value_attribut(ID,'Property','value_type',value) ;
}
```

Algorithme 8 Exemple d'appels de l'API

- Création de la propriété "age" de la classe *person*

```
String id_age = Data_Property.CreateInstance() ;// On crée une instance de Data_Property
Data_Property.setName(id_age,"age")// On initialise le nom de la propriété age
Data_Property.setValue(id_age,'integer') ;// On initialise le type de la propriété age
...
```

- Manipulation de la propriété *age* de la classe *person*

```
String value_type = Data_Property.getValue_type(id_age) ;// le type de valeurs de la propriété age
String age_personne = Data_Property.getName(id_age) ;// le nom de la propriété age
```

2.3.3 Conclusion

Nous avons présenté dans cette section deux interfaces de programmation orientée objet permettant d'accéder aux entités définissant une ontologie en vérifiant la correction syntaxique des appels. Ces APIs, contrairement à l'API de la fonction F_2, sont spécifiques du modèle d'ontologie utilisé et permettent d'accéder aux concepts des ontologies sous forme d'objets. Ces APIs sont constituées de classes correspondantes aux entités du modèle d'ontologie utilisé. Les méthodes de ces classes sont composées d'un ensemble d'accesseurs (set/get/insert,delete) associées à chaque attribut des entités du modèle d'ontologie. Nous avons discuté de deux types d'API objets envisageables. La première *API objet* est caractérisée par le fait que (1) les données manipulées sont de type objet, (2) elle permet de gérer un cache d'objets. La deuxième dite *API fonctionnelle*, est caractérisée par le fait que (1) toutes les données manipulées sont de type simple (entier, chaîne de caractères, etc.), (2) les objets des classes sont gérés à partir de leur identifiant, (3) toutes les méthodes des classes sont *statiques* et accèdent directement à la base de données pour la récupération des données, en d'autres termes, il y a aucun cache d'objets.

Il est impératif que l'une au moins de ces APIs soit implantée sur une BDBO. Il convient de souligner que ces deux types d'API peuvent être être générées par un programme générique (i.e., indépendant du modèle d'ontologie particulier) si, comme nous l'avons proposé pour la fonction F_1, le modèle d'ontologie est lui-même représenté comme instance d'un méta-modèle d'ontologie.

2.4 F4 : Capacité de lire des ontologies représentées dans leur format d'échange et de les stocker dans la BDBO.

Cette fonction du système, vise à permettre d'atteindre l'objectif O_1. Les ontologies et leurs données sont créées le plus souvent au moyen d'éditeurs (exemple : protégé [61] et SWOOP [88] pour les langages du web sémantique, PLIBEditor [100] pour PLIB, etc.). Ces éditeurs, pour la plupart, gèrent les ontologies et les données en mémoire centrale et permettent de les sauvegarder dans des fichiers sous différents formats. L'intégration des ontologies et leurs instances posent deux types de problèmes. D'une part, il faut réussir à intégrer des instances dont le schéma dans la base de données n'est pas défini à l'avance. D'autre part, il faut tenir compte du fait que les ontologies comme tout autre modèle sont susceptibles d'évoluer [119]. Le premier problème sera traité dans la section 3. Notre discussions portera uniquement ici sur la représentation des ontologies.

La notion de version et de révision de concepts ontologiques est normalement représentée dans le modèle d'ontologie. En d'autres termes, c'est le modèle d'ontologie qui indique si une modification quelconque d'un concept de l'ontologie implique soit une nouvelle version du concept soit une révision. Nous avons détaillé, en particulier, dans l'annexe A section 5 les principes qui s'appliquent dans le cadre du modèle d'ontologie PLIB.

Avant de discuter de la problématique et des solutions possibles pour la gestion de l'évolution des concepts des ontologies, nous présentons, dans la section suivante, les contraintes d'évolution que nous proposons d'imposer, sur lesquelles la solution que nous avons proposée pour la gestion

des versions des ontologies est basée.

2.4.1 Discussion du principe de la continuité ontologique

Les contraintes que nous proposons d'imposer pour régler l'évolution des ontologies résultent des différences fondamentales existantes, de notre point de vue, entre les modèles conceptuels et les ontologies. Un modèle conceptuel est un modèle, c'est-à-dire, selon Minsky [113], un objet qui, permet de répondre à des questions que l'on se pose sur un autre objet. Lorsque les questions changent, c'est-à-dire, lorsque les objectifs organisationnels auxquels répond un système d'information sont modifiés, son modèle conceptuel est modifié, sans que cela signifie, le moins du monde, que le domaine modélisé soit modifié. Au contraire, une ontologie est une conceptualisation visant à représenter l'essence des entités d'un domaine donné sous forme consensuelle pour une communauté. C'est une théorie logique d'une partie du monde, partagée par toute une communauté, et qui permet aux membres de celle-ci de se comprendre. Ce peut être, par exemple, la théorie des ensembles (pour les mathématiciens), la mécanique (pour les mécaniciens), la comptabilité analytique (pour les comptables) ou les propriétés des roulements (pour des techniques). Pour de telles ontologies, deux types de changements doivent être distingués [166] :

1. L'évolution normale d'une théorie est son approfondissement. Des vérités nouvelles, plus détaillées s'ajoutent aux vérités anciennes. Mais ce qui était vrai hier reste vrai aujourd'hui.
2. Mais, il peut également arriver que des axiomes de la théorie soient remis en cause. Il ne s'agit plus là d'une évolution mais d'une révolution où deux systèmes logiques différents vont coexister ou s'opposer.

Les ontologies que nous visons correspondent à cette conception. Il s'agit d'ontologies qui sont soit normalisées, par exemple au niveau international, soit définies par des consortiums importants et qui formalisent de façon stable et partagée les connaissances d'un domaine technique. Les changements auxquels nous nous intéressons dans notre approche ne sont donc pas les révolutions, qui correspondent à un changement d'ontologie, mais les évolutions d'ontologie. C'est la raison pour laquelle nous proposons d'imposer aux versions successives d'une ontologie le principe de continuité ontologique défini par notre hypothèse H_5 que nous rappelons ici.

2.4.1.1 Principe de la continuité ontologique

Nous détaillons ci-dessous la signification de ce principe pour les ontologies conformes au modèle formel proposé dans la section 1.3. Nous noterons désormais par un indice supérieur les versions des différents composants d'une ontologie [166] : $O =< C, P, Sub, Applic >$.

– **Permanence des classes.**

1. L'existence d'une classe ne pourra pas être infirmée à une étape ultérieure :

$$\forall k, C^k \subset C^{k+1}.$$

Pour tenir compte de la réalité, il pourra apparaître pertinent de considérer comme obsolète telle ou telle classe. Elle sera alors marquée en tant que telle ("deprecated"),

mais continuera à faire partie des versions ultérieures de l'ontologie.

2. Concernant son contenu, la définition d'une classe pourra être affinée sans que l'appartenance à cette classe d'une instance antérieure ne puisse être remise en cause. Cela signifie que :
   a. la définition des classes pourra elle-même évoluer,
   b. chaque définition d'une classe sera associée à un numéro de version,
   c. la définition (intentionnelle) de chaque classe englobera les définitions (intentionnelles) de ses versions antérieures et donc,
   d. toute instance d'une version de classe sera également instance de ses versions ultérieures (même si elle peut, en fonction d'un cycle de vie dépendant du domaine d'application, être devenue obsolète).

– **Permanence des propriétés**.
  – Pour la même raison,
$$\forall k, P^k \subset P^{k+1}.$$
  Une propriété pourra, de même, devenir obsolète.
  – Concernant les valeurs, les valeurs existantes dans le co-domaine d'une propriété ne pourront être remises en cause. Une propriété pourra évoluer dans sa définition ou dans son domaine de valeurs. Mais le principe de continuité ontologique implique que les domaines de valeurs ne pourront être que croissants, certaines valeurs étant, éventuellement, marquées comme obsolètes.

– **Permanence de la subsomption.**
  La subsomption est également un concept ontologique qui ne pourra être infirmé. Notons $Sub^* : C \rightarrow 2^C$, la fermeture transitive de la relation de subsomption directe Sub. On a alors :
$$\forall C^k, Sub^*(C^k) \subset Sub^*(C^{k+1}).$$
  Notons que cette contrainte permet évidemment un enrichissement de la hiérarchie de subsomption des classes, par exemple en intercalant des classes intermédiaires entre deux classes liées par une relation de subsomption, ou en rajoutant d'autres classes subsumées.

– **Description des instances.**
  Le fait qu'une propriété $p \in Applic(c)$ signifie que la propriété est rigide pour toute instance de c. Il s'agit encore d'un axiome qui ne pourra être infirmé :
$$\forall C^k, Applic(C^k) \subset Applic(C^{k+1}).$$
  Soulignons que ceci ne suppose pas que les mêmes propriétés soient toujours utilisées pour décrire les mêmes instances d'une même classe. Il ne s'agit en effet plus là d'une caractéristique ontologique, mais seulement de son schéma de représentation.

2.4.2 Gestion d'ontologies évolutives : position du problème.

2.4.2.1 Problématique

Supposons que l'on souhaite intégrer un concept d'une ontologie dans une BDBO. Quatre cas

de figure se présentent.

Cas 1. le concept n'existe pas dans la base de données.

Cas 2. il existe déjà dans la base de données mais dans une version antérieure.

Cas 3. il existe déjà dans la base de données mais dans une version supérieure.

Cas 4. il existe déjà dans la base de données mais dans la même version.

L'intégration du concept dans le **Cas 1** et le **Cas 4** est évidente. Dans le **Cas 1**, le concept est directement inséré dans la base de données et toutes les références à ce concept sont maintenues. Dans le **Cas 4**, on n'insère pas le concept puisqu'il existe déjà dans la base de données, mais toutes les références portant sur ce concept dans l'ontologie en cours d'intégration sont converties en référence au concept existant.

Dans le **Cas 2** et le **Cas 3**, il est indispensable de définir une politique pour la gestion des différentes versions du même concept et pour la gestion des relations entre concepts. Supposons par exemple que le concept soit une classe C dont la version k appartient à la BDBO. Cette classe C^k de l'ontologie O est décrite par un ensemble de propriétés ($Applic(C^k)$), et est associée à une population d'instances ($Pop(C^k)$).

Supposons maintenant que nous intégrons une classe C^{k+1} à une version $k+1$. Plusieurs questions se posent.

- Pourra-t-on accéder aux instances de C^k à partir de la classe C^{k+1} ?
- Les deux versions C^k et C^{k+1} doivent-elles exister simultanément dans la base de données ?
- Si oui comment gérer les instances propres à chaque classe ?
- Doit-on transférer les relations associées à C^k sur la classe C^{k+1} ?

Illustrons ce dernier problème par l'exemple suivant (cf. figure 3.2). Soit C_j^k une super-classe directe de C_i^k ($C_i^k \in Sub(C_j^k)$) et soit C_i^{k+1} une nouvelle version de la classe C_i^k. Dans la base de données, le lien de subsomption est matérialisé entre les classes C_j^k et C_i^k. Si on intègre la classe C_i^{k+1} dans la base de données et que C_i^k et C_i^{k+1} co-existent, comment représenter la relation de subsomption entre les classes C_i et C_j ? Devrait-on toujours conserver la relation $C_i^k \in Sub(C_j^k)$ ou la remplacer par $C_i^{k+1} \in Sub(C_j^k)$ qui liera la nouvelle version de la classe avec sa superclasse, sachant qu'une seule de ces relations est évidemment possible pour définir une ontologie cohérente.

Ce problème de gestion des relations se rencontre chaque fois que des concepts (classes et/ou propriétés) se référencent. Il est donc indispensable de définir une approche globale pour résoudre ces problèmes de référence.

2.4.2.2 Solutions envisageables

Pour représenter les différentes relations entre les versions des concepts d'une ontologie, trois solutions peuvent, dans un premier temps, être considérées (cf. figure 3.3, solution 1, 2 et 3).

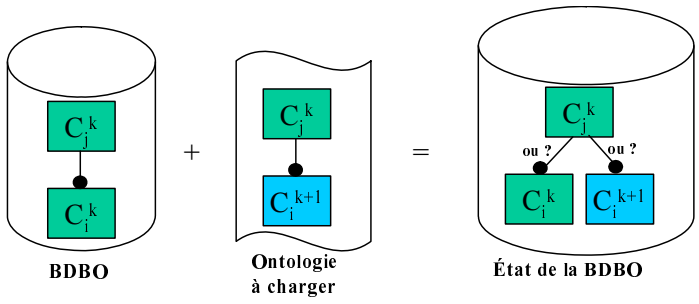


FIG. 3.2 – Problématique de la gestion des relations entre concepts des ontologies

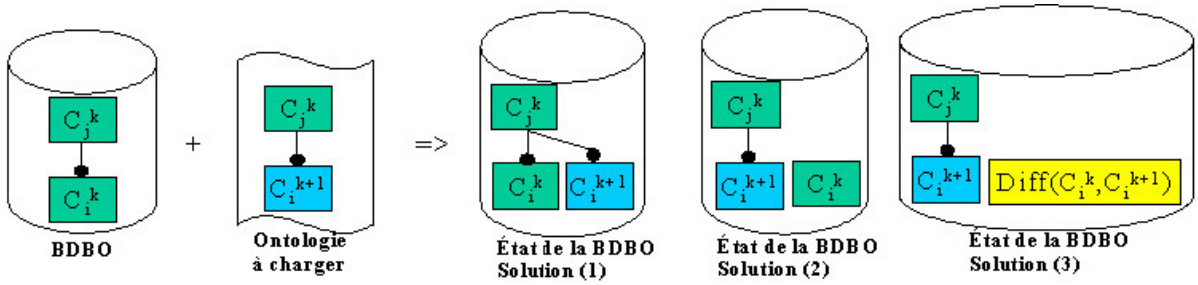


FIG. 3.3 – Différentes solutions pour la pérennisation des relations des concepts des versions des ontologies.

1. La première consisterait à stocker dans la base de données toutes les versions d'un concept ainsi que toutes les relations ayant existées, dans les différentes ontologies intégrées entre les versions des concepts (cf. figure 3.3, solution (1)). Pour chaque requête, par exemple [$Sub(C_j^k)$?], des règles de calcul devraient alors être définies pour répondre à la requête par parcours de l'ensemble des versions. L'avantage de cette solution serait de pouvoir restaurer par calcul tout état antérieur de l'ontologie. Un inconvénient de cette approche serait, par contre, de rallonger le temps de réponses de toutes les requêtes portant sur l'ontologie. Il resterait également dans cette approche à définir les règles de calcul.

2. La deuxième solution consisterait à transférer automatiquement lors de chaque mise à jour (i.e., arrivée d'une nouvelle version d'un concept) les relations de version à version (cf. figure 3.3, Solution (2)). Dans cette solution, tous les concepts sont dans leur dernière version disponible, permettant des réponses rapides aux requêtes portant sur l'ontologie la plus récente. Cette approche présente néanmoins une limite : toutes les relations entre les versions précédentes d'un concept sont rompues aux profits de la dernière version. Les versions archivées sont donc dans un état incohérent.

3. La troisième solution consisterait à ne conserver que la dernière version du concept et représenter dans la base de données les différences structurelles [118, 119] entre les versions des concepts ($DIFF(C^k, C^{k+1})$) [95](cf. figure 3.3, Solution (3)). De nombreux travaux ont été menés ces dernières années notamment par [20, 118], et en particulier les travaux de

Bernstein sur la gestion des modèles [20]. Ce dernier a développé un framework pour mettre en correspondance des modèles quelconques (définis dans différents formalismes). Dans son approche, un mapping entre deux modèles est lui-même défini comme un modèle contenant des expressions liant les concepts d'un modèle à un autre. En adoptant son approche, on pourrait imaginer de représenter dans la BDBO la différence entre les versions des concepts et de calculer au besoin les versions antérieures d'un concept.

2.4.2.3 Notre proposition

La solution que nous proposons consiste à combiner les avantages des deux premières solutions, tout en évitant leurs inconvénients respectifs. Elle consiste à représenter à la fois les concepts versionnés et non versionnés. Notre proposition consiste à introduire la notion de *version courante* de concept qui permet de représenter les dernières versions disponibles de chaque concept et ainsi de consolider toutes les relations existantes entre concepts de différentes versions à travers des relations en version courante (cf. figure 3.4, solution(4)). Cette approche se base sur l'hypothèse de continuité ontologique, en particulier, ainsi que nous l'avons vu : toutes les propriétés applicables d'une classe C_i^k sont également applicables à C_i^{k+1}, (2) toutes les classes subsumées par C_i^k sont également subsumées par C_i^{k+1}.

La mise en œuvre de cette proposition amène à la structuration suivante :

- le schéma des ontologies se divise deux parties (logiques) :
  1. la partie *historique* qui enregistre les différentes versions des concepts, ainsi que les relations qui les lient,
  2. la partie *courante* qui duplique les dernières versions des différents concepts (C_i^+ sur la figure 3.4) et consolide entre eux toutes les relations existantes.
- un concept courant correspond à la plus grande version du concept.
- tous les concepts courants ne sont en relation qu'avec d'autres concepts courants.
- toute nouvelle version d'un concept, est automatiquement insérée dans la partie *historique* et le même concept en version courante est soit créé, s'il n'existe pas, soit mis à jour avec les informations de la nouvelle version du concept, si une version antérieure existait déjà.
- Il ne peut exister deux versions courantes d'un même concept dans la base de données.

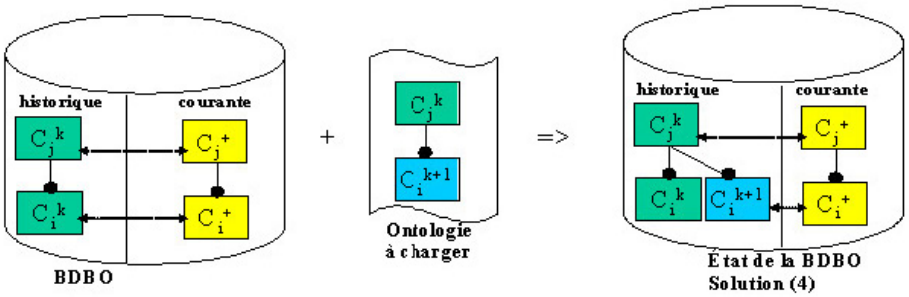


FIG. 3.4 – Notre proposition pour la pérennisation des relations des concepts des versions des ontologies.

Cette approche, contrairement aux approches précédentes, n'exige pas de faire des calculs pour répondre aux requêtes sur l'ontologie courante, tout en permettant si besoin est, de retrouver par calcul toute version antérieure de l'ontologie. Notons que cette approche offre également une solution cohérente pour gérer les mises à jour de BDBO lorsque la gestion des versions antérieures de l'ontologie n'est pas considérée comme nécessaire. Elle consiste simplement à gérer la version *courante* dans laquelle sont consolidées toutes les relations introduites par les différentes versions de l'ontologie.

Nous présentons dans la section suivante la fonction F_5 qui permet l'extraction de concepts d'une BDBO.

2.5 F5 : Capacité d'extraire des ontologies présentes dans la BDBO et de les exporter selon un format d'échange.

Cette fonction est l'inverse de la fonction précédente. Il est question ici d'extraire un sous-ensemble de concepts de l'ontologie dans la dernière version ainsi que tout ou partie des instances associées à ces concepts. La principale difficulté de cette fonction est le fait que les définitions des concepts extraits de la BDBO doivent pouvoir, si besoins est, être *complètes*, de sorte à pouvoir être exploiter de façon autonome. Un concept est dit *complet* si tous les *éléments* participant à sa définition sont eux mêmes définis (de façon récursive) dans le référentiel où il se trouve. Par exemple, pour une classe C, il doit pouvoir se trouver dans son référentiel, à la fois la définition de ses super-classes et de ses propriétés applicables, et non pas seulement l'identifiant (URI, BSU) de ses super-classes et propriétés applicables.

2.5.1 Extraction d'information d'une BDBO : Position du problème

Lors de l'extraction des concepts, il faut s'assurer de la *complétude syntaxique* et de la *complétude sémantique* des concepts.

- **Complétude syntaxique**
  Si une entité extraite référence, par un pointeur une entité, celle-ci doit également être extraite pour assurer la cohérence syntaxique (i.e, l'intégrité référentielle) de la population extraite.
- **Complétude sémantique**
  Si la ou les entités qui représentent la définition d'un concept sont extraites, il doit être possible d'extraire également automatiquement toutes les entités qui définissent les autres concepts qui participent à la définition du concept extrait.

Notons que la complétude sémantique n'est pas toujours nécessaire. Cela dépend des besoins de l'utilisateur et de l'usage qu'il veut faire de l'ensemble extrait. Néanmoins, nous considérons que la fonction doit pouvoir, au choix de l'utilisateur, assurer ou non la complétude sémantique.

La complétude syntaxique dépend du modèle. Il s'agit seulement de suivre, de façon récursive (si besoin est), les relations de composition pour assurer la cohérence syntaxique de l'ensemble extrait. Notons que ce problème ne se pose pas en RDF, RDFS et OWL qui n'utilisent pas de mécanismes de pointeur ou de clé (REF, REFKEY) mais de simples chaînes de caractères.

La complétude sémantique nécessite de définir les concepts qui contribuent à la définition d'un concept particulier. Nous nous basons, pour cela, sur notre modèle formel d'ontologie : $< C, P, Sub, Applic >$.

Notons par p_{def} ou c_{def} la ou les entités qui constituent la définition respective d'une propriété d'identifiant p et d'une classe d'identifiant c. La fonction $extraire(x \in P \cup C)$ permet d'extraire la définition du concept donné en paramètre. Notre algorithme (algorithme 9), suivant le type de concept, se présente comme suit :

- *Extraction d'une classe*
  Lorsqu'on extrait une classe C, nous extrayons également la définition de toutes ses classes subsumantes ainsi que la définition de ses propriétés applicables.
- *Extraction d'une propriété*
  Lorsqu'on extrait une propriété, nous extrayons également la classe la plus générale qui définit son domaine, et le cas échéant la classe qui constitue son co-domaine.

Algorithme 9 Extraction des concepts des ontologies.

1. Extraction d'une classe C_i.
   - $\forall\ C^k$ *tel que* $C_i \in Sub(C^k) \Rightarrow\ extraire(C^k_{def})$
   - $\forall\ P^k \in Applic(C_i) \Rightarrow\ extraire(P^k_{def})$
2. Extraction d'une propriété P_i.
   - Soit $C^k \in Applic^{-1}(P_i)$ *tel que* $[\forall C_l \in Sub^{-1}(C_k), P_i \notin Applic(C_l)] \Rightarrow extraire(C^k_{def})$ // racine du ou des arbres où P_i est applicable,
   - soit $c =$co-domaine$(P_i) \Rightarrow extraire(c_{def})$// si P_i est une association, extraire la classe qui est son co-domaine

Cet algorithme permet donc d'extraire, de façon récursive, tous les concepts dont la définition contribue à la définition d'un concept que l'utilisateur souhaite extraire. Dans notre algorithme, nous avons choisi d'extraire les classes avec la définition de toutes leurs super-classes ainsi que toutes leurs propriétés applicables. Pour les propriétés, nous décidons de les extraire avec leur domaine et co-domaine. Pour une application donnée, on pourrait, par exemple, vouloir extraire toutes les sous-classes et/ou les super-classes d'une classe. L'algorithme d'extraction sémantique sera donc implémenté suivant le besoin spécifique des utilisateurs. Avant l'implémentation de l'algorithme, il est nécessaire d'identifier toutes les relations entre les concepts qu'on voudrait extraire pour chaque type de concept.

La figure 3.5 illustre l'utilisation de cet algorithme lorsqu'on l'applique à une ontologie RDF. Supposons que nous voulons extraire le concept C_1. La zone grisée donne l'ensemble des concepts dans le cas d'une *extraction sémantique* : seules les propriétés applicables (P_1 et P_2) et la super-classe de C_1 (C_0) sont extraites de la base de données. Dans le cas d'une *extraction syntaxique*, uniquement la super-classe de C_1 (C_0) est extraite de la base de données.

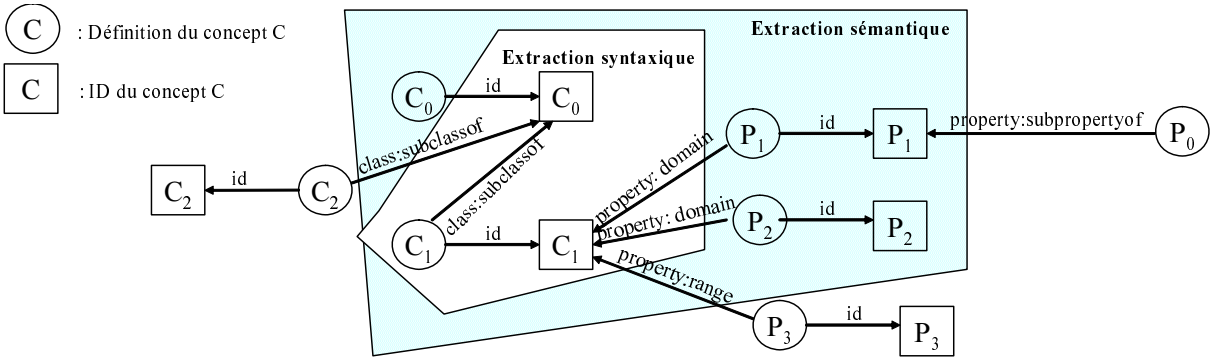


FIG. 3.5 – Exemple de graphe d'ontologie : extraction de concepts.

Pour résumé, nous avons présenté dans cette section la fonction F_5 qui permet de définir des mécanismes pour l'extraction des concepts des ontologies dans une BDBO. La principale exigence de cette fonction était qu'elle soit capable d'extraire de la BDBO des sous-ensembles de concepts cohérents ou autonomes. Nous avons proposé deux algorithmes selon que l'on souhaite faire des extractions cohérentes simplement du point de vue intégrité référentielle ou autonome du point sémantique. Nous étudierons, dans la section 3.3, les problèmes liés à l'extraction de données à base ontologique.

2.6 Représentation du méta-modèle

Nous avons vu précédemment que pratiquement toutes les fonctions identifiées (F_1 à F_5) suggéraient la représentation, dans la base de données, d'un méta-modèle d'ontologie, et d'une représentation des modèles d'ontologies en tant qu'instances de ce méta-modèle. La figure 3.6 compare les méthodes de codage des fonctions F_1 à F_5 avec ou sans représentation explicite du modèle d'ontologie comme une instance d'un méta-modèle.

Cette comparaison confirme clairement la nécessité de représenter sous forme d'instances, non seulement les ontologies, mais également les modèles d'ontologies si l'on veut pouvoir satisfaire l'objectif O_2 (cf. section 1.1). Comme nous le voyons dans la figure 3.6, la représentation du modèle d'ontologie dans la base de données permet de rendre générique, par rapport aux modèles d'ontologies utilisés dans la BDBO, toutes les fonctions que nous avons identifiées et qui devront faire partie du système. De plus, ce méta-modèle peut être utilisé pour représenter les modifications possibles du modèle d'ontologie considéré.

Nous discutons dans la section suivante de la structure des tables devant permettre la représentation, en tant qu'instances, des modèles d'ontologies dans la BDBO et de leur initialisation. Dans la section 2.6.2, nous discutons de la nécessité de représenter le méta-modèle lui-même pour permettre la mise en œuvre de certaines opérations.

Fonction	Sans représentation explicite du méta-modèle	Avec représentation explicite du méta-modèle
F1 - Définition du modèle logique de représentation des ontologies.	Définition manuelle du schéma (ou génération par programme externe)	– Définition de règles de représentation logique (R1) pour toute instance du méta-modèle (exemple: tout schéma XML pour OWL, ou tout modèle EXPRESS pour PLIB). – Programme de génération qui interprète les instances du méta-modèle correspondant au modèle courant et met en œuvre (R1).
F2 - API à liaison différée	Tout le modèle doit être codé dans l'API (cf. algorithme 2)	L'API devient un interprète des instances du méta-modèle (cf. algorithme 3) en exploitant (R1)
F3 - API à liaison préalable	Toute l'API doit être codée à la main	– Définition de règles génériques (R3) de représentation d'un modèle EXPRESS ou d'un schéma XML dans le langage cible (Par exemple java). – Programme de génération interprétant le méta-modèle en exploitant (R1) et (R3)
F4-F5 - Lecture / extraction des concepts d'ontologie dans la BD	Ecriture d'un programme spécifique	– Des règles génériques (R2) de représentation externe existent déjà (par exemple: document XML pour OWL, fichier d'instances EXPRESS pour PLIB) – Programme générique qui interprète les instances du méta-modèle correspondant au modèle courant en exploitant (R1) et (R2)

FIG. 3.6 – Implantation des fonctions nécessaires : avec et sans méta-schéma.

2.6.1 Structure pour la représentation des modèles d'ontologies

La structure du méta-modèle nécessaire pour permettre la représentation des modèles d'ontologies va dépendre du fait que le SGBD utilisé : (1) est un SGBD Objet, et (2) permet de représenter tous les mécanismes existant dans le formalisme de définition du modèle d'ontologie (par exemple, si besoin est, l'héritage multiple). Si ces deux conditions sont vérifiées, alors le méta-modèle préexiste en général dans le SGBD utilisé et s'identifie avec la *méta-base* du système. Dans les autres cas, il s'agit d'une partie nouvelle pour une base de données et qui est *nécessaire* dans une BDBO. Nous nommerons *méta-schéma* cette partie dans la suite de la thèse. Représenter le modèle d'ontologie dans la base de données sous forme d'instances d'un méta-modèle, peut se ramener à la problématique de définition de la structure de tables nécessaire pour la représentation d'ontologies. Ce point a été discuté à l'occasion de la fonction F_1 (cf. section 2.1) :

- le méta-modèle est lui-même défini par un formalisme de modélisation orienté-objet comme le modèle d'ontologie,
- le modèle d'ontologie doit pouvoir se représenter comme un ensemble d'instances du méta-modèle, comme c'est le cas des ontologies pour le modèle d'ontologie,
- le méta-modèle et le modèle d'ontologie seront exprimés dans le même formalisme de modélisation (XML pour RDFS, OWL ou DAML+OIL, EXPRESS pour PLIB).

Dans le cas de la fonction F_1, nous avons préconisé d'utiliser les techniques de correspondance objet-relationnel pour la définition de la structure des tables permettant la représentation des ontologies. Vu que le méta-modèle est défini dans le même formalisme que le modèle d'ontologie, on pourrait envisager d'utiliser la même technique pour définir la structure de tables permettant la représentation des modèles ontologie dans la base de données.

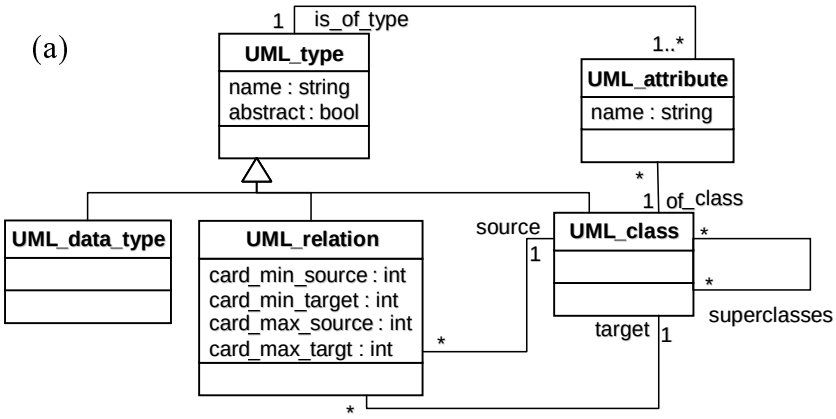
Notons que si le formalisme utilisé pour définir le modèle d'ontologie et le méta-modèle est le même, *alors le même code générique pourra être réutilisé* pour la mise en œuvre des mêmes besoins concernant les modèles d'ontologies. Ceci s'applique aux fonctions F_1 (pour définir des tables), F_2 et F_3 (pour accès aux modèles d'ontologies), F_4 (pour lire les modèles d'ontologies).

Nous présentons, dans la figure 3.7a, un exemple simplifié de méta-modèle UML. La figure 3.7b présente la représentation du modèle d'ontologie de la figure 3.1b dans des tables issus du méta-modèle UML de la figure 3.7a par transformation objet-relationnel en utilisant les règles de correspondance définies dans l'exemple 1 (cf. section 2.1).

Notons que la représentation du modèle d'ontologie dans la base de données offre la possibilité de représenter au sein d'une même base de données plusieurs ontologies issues de modèles d'ontologies différents. Il suffit dans ce cas d'appliquer sur chacun des modèles d'ontologies, les traitements mis en oeuvre pour le déploiement d'un modèle d'ontologie particulier (PLIB) sur la BDBO OntoDB. Ceci n'est néanmoins possible que sous deux conditions :

1. Tous les modèles d'ontologies doivent être définis dans un même et unique formalisme de modélisation (celui sur lequel les programmes de déploiement se basent). Pour le prototype que nous avons implémenté, le formalisme utilisé fut le langage EXPRESS [143].

(a)



(b)

UML_relation							
OID	name	source	target	s_min	s_max	t_min	t_max
8	properties	1	2	1	1	0	null
9	superclass	2	1	0	1	0	null
21	target_class	4	1	1	1	0	null

UML_class			
OID	name	superclasses	abstract
1	class		false
2	property		true
3	data_property	2	false
4	object_property	2	false

UML_data_type	
OID	name
10	String
11	String
12	String

UML_attribute				
OID	name	of_class	is_of_type	...
5	name	1	10	
6	name	2	11	
7	value_type	3	12	

FIG. 3.7 – (a) Un exemple de méta-modèle d'ontologie et (b) son utilisation pour représenter le modèle de la figure 3.1b.

Par exemple, supposons que nous souhaitons créer/insérer des ontologies PLIB, RDF Schema, DAML+OIL et OWL dans OntoDB, alors les modèles d'ontologies RDF Schema, DAML+OIL, OWL doivent être "re-modélisés"' dans le formalisme EXPRESS.

2. Pour pouvoir importer des ontologies RDF Schema, DAML+OIL, OWL, un *convertisseur*, i.e., un programme, qui servira à faire migrer les ontologies dans leur format d'origine (XML pour RDF Schema, DAML+OIL, OWL) dans le format d'instances du formalisme sur lequel se base les programmes de déploiement, doit être défini. Dans notre prototype, ce sera donc des instances de fichiers physiques EXPRESS [79].

Cette approche automatique a néanmoins l'inconvénient de ne pas factoriser les éléments communs de tout modèle d'ontologie (i.e., les classes, les propriétés et les types). Une autre approche possible consiste à étendre manuellement, dans le langage OntoQL (cf. section 4.2.5 du chapitre 4) le modèle d'ontologie existant (PLIB) pour y introduire de nouvelles spécifications.

Dans la section qui suit, nous discuterons de la représentation des instances dans la base de données à base ontologique.

2.6.2 Méta-schéma réflexif

Les exigences existant jusqu'alors sur le *méta-schéma*, étaient d'être capable de représenter l'ensemble des variantes des modèles d'ontologies envisagés. Si nous imposons alors au *méta-schéma* d'être, de plus, réflexif, c'est-à-dire de pouvoir se représenter lui-même, on pourra représenter le *méta-modèle* lui-même comme instance de sa propre structure (cf. figure 3.8). Nous illustrons ci-dessous le besoin de représenter ainsi le méta-modèle dans la base de données à travers un exemple.

Reprenant le méta-modèle de la figure 3.7a et supposons que nous souhaitons appliquer une action spécifique suivant le type d'un attribut d'une classe de l'ontologie. Soit oid_{att}, l'oid de l'attribut représenté en tant que *UML_ attribute*. Dans l'exemple suivant, nous présentons un extrait de programme qui fait appel à quelques fonctions de l'API à liaison différée (cf. section 2.2). Le fragment de programme de l'algorithme 10 vise à exécuter une action spécifique suivant le type d'un attribut.

La fonction *get_final_ type* est sensée donner le *type* (i.e., le domaine de valeur) de l'attribut ayant l'oid oid_{att} qui doit normalement être l'une des sous-classes de *UML_ type* : *UML_ data_ type*, *UML_ relation* ou *UML_ class*. On devrait donc dans tous les cas, faire des requêtes sur chacune des tables *UML_ data_ type* et *UML_ relation* et *UML_ class* pour déterminer dans laquelle le type se trouve. La question qui se pose est donc comment déterminer ces tables ?

Deux solutions sont possibles :

– soit on code tout le méta-modèle dans la fonction. Dans ce cas, le corps de la fonction *get_final_ type* se présentera comme dans l'algorithme 11.
  Le corps de la fonction *get_final_ type* peut être écrit manuellement mais nécessitera malheureusement de décrire toutes les hiérarchies des entités du modèle dans celui-ci.

Algorithme 10 Exécution d'une action spécifique suivant le type d'un attribut.

```
...
  oid_i :=get_value_attribute(oid_att,"UML_attribute","is_of_type")
  IF get_final_type (oid_i, "UML_type") == "UML_data_type THEN"
    doSomeThing ...
  ELSE ...
  IF get_final_type (oid_i, "UML_type") == "UML_relation THEN"
    doSomeThing ...
  ELSE ...
  IF get_final_type (oid_i, "UML_type") == "UML_class THEN"
    doSomeThing ...
  END IF;
...
```

Algorithme 11 Codage du méta-modèle dans la fonction *get_final_type*.

```
FUNCTION get_final_type : oid x nom_entité -> nom_entité;
...
  IF nom_entité = 'UML_Type' THEN
  ExecQuery :
    SELECT "UML_data_type" FROM UML_data_type
    WHERE OID = oid
      UNION
    SELECT "UML_relation" FROM UML_relation
    WHERE OID = oid
      UNION
    SELECT "UML_class" FROM UML_class
    WHERE OID = oid
  IF nom_entité = 'UML_XXX' THEN
...
```

– Soit en représentant toutes les entités méta-modèle sous forme d'instances dans les tables du *méta-schéma* (cf. figure 3.8). Ainsi celles-ci pourront être parcourues pour déterminer toutes les sous-classes de la classe *UML_ type*. Grâce à la représentation du *méta-schéma* dans la base de données, l'on pourra utiliser l'*API à liaison différée* définie précédemment (cf. section 2.2) dans l'écriture de la fonction *get_final_ type* peut être réalisé comme dans l'algorithme 12.

Algorithme 12 Codage de la fonction *get_final_ type* en accédant au méta-schéma.

FUNCTION *get_final_ type* : oid x nom_entité -> nom_entité ;
BEGIN
...
Retrouver l'oid de l'instance de *UML_ class* qui pour nom "nom_entité"
dans la table *UML_ class*. soit oid_{class}
// On récupère toutes les sous-classes de la classe dont l'oid est donné en paramètre.
Array subclasses := used_in(oid_{class},'UML_class','superclasses') ;
// On parcourt toutes les classes récupérées
FOREACH c IN subclasses DO
On génère la requête de l'algorithme 11
END FOREACH ;
On exécute la requête générée.
...
END FUNCTION ;

UML_relation							
OID	**name**	**source**	**target**	**s_min**	**s_max**	**t_min**	**t_max**
8	properties	1	2	1	1	0	null
9	superclass	2	1	0	1	0	null
21	target_class	4	1	1	1	0	null
18	is_of_type	16	13	1	1	0	null
19	of_UML_class	16	14	1	1	0	null

UML_data_type	
OID	**name**
10	String
11	String
12	String
20	String

UML_class			
OID	**name**	**superclasses**	**abstract**
1	class		false
2	property		true
3	data_property	2	false
4	object_property	2	false
13	UML_type		true
14	UML_class	13	false
15	UML_data_type	13	false
16	UML_attribute		false
21	UML_relation	13	false

UML_attribute				
OID	**name**	**of_class**	**is_of_type**	**...**
5	name	1	10	
6	name	2	11	
7	value_type	3	12	
17	name	13	20	

FIG. 3.8 – Auto-représentation du méta-modèle réflexif.

L'autoreprésentation du méta-modèle réflexif au sein de sa propre structure constitue donc un avantage significatif pour l'écriture de programmes plus simple. L'accès à un niveau "méta" en base de données pouvant avoir pour effet de ralentir les programmes, il convient de noter que la représentation explicite du méta-modèle dans la base de données n'empêche pas d'optimiser

les programmes afin qu'ils y accèdent le moins possible. C'est d'ailleurs ce qui est fait, dans notre implantation, concernant l'API à liaison préalable permettant l'accès, en Java, aux ontologies PLIB.

3 Représentation des données à base ontologique

Nous analysons dans cette section le problème de la représentation des données décrites en tant qu'instances des classes d'ontologies. Dans la sous-section 3.1, nous présentons d'abord du schéma logique de bases de données que nous proposons. Dans la sous-section 3.2, nous discutons de la gestion du versionnement des instances et du cycle de vie des instances dans les bases de données à base ontologiques. Enfin la section 3.3 étudie la possibilité d'extraire des sous-ensembles cohérents du contenu d'une BDBO.

3.1 Schéma de représentation des instances des classes.

3.1.1 Position du problème et hypothèses

Une ontologie vise à représenter la sémantique des objets d'un domaine en les associant à des classes, et en les décrivant par des valeurs de propriétés. Selon les modèles d'ontologies utilisés, plus ou moins de contraintes existent sur ces descriptions. A titre d'exemple, si l'on n'introduit pas de restrictions particulières sur les descriptions OWL, un objet peut appartenir à un nombre quelconque de classes et être décrit par n'importe quelles propriétés. Ceci donne à chaque objet du domaine une structure qui peut lui être spécifique. A contrario, un schéma de base de données vise à décrire des ensembles d'objets "similaires" par une structure logique identique de façon à pouvoir optimiser les recherches de tels ensembles par des techniques d'indexation.

En absence de toute hypothèse particulière sur la description à base ontologique des objets du domaine, une structure commune à l'ensemble des objets doit permettre d'associer à chaque objet :

- un sous-ensemble quelconque de l'ensemble des classes,
- un nombre quelconque de propriétés.

Cette exigence entraîne alors soit un coût de stockage supplémentaire, avec une représentation systématique des appartenances ou des propriétés non pertinentes pour une instance, soit des temps de traitements importants résultant de la nécessité de réaliser un nombre important de jointures, si seules les appartenances et les propriétés pertinentes sont représentées. Nous présentons d'ailleurs une comparaison de l'efficacité des deux méthodes évoquées ci-dessus et de la nouvelle représentation que nous proposons dans le chapitre 5 de cette thèse.

En fait, dans bien des cas, et c'est en particulier le cas dans le domaine de l'ingénierie et du commerce électronique [123] :

1. Les propriétés sont typées : chacune est associée à une classe qui définit l'ensemble des objets auxquels la propriété peut être appliquée.

2. Chaque objet est caractérisé par l'appartenance à une et une seule classe, même s'il peut, par ailleurs être décrit par des instances d'autres classes pour prendre en compte une multiplicité de points de vue de modélisation.

Dans le cadre de notre modèle d'architecture *OntoDB*, nous imposerons deux restrictions désignées sous le terme d'hypothèse de *typage fort* (cf. section 1).

- **R1-** tout objet du domaine est *caractérisé* par son appartenance à une et une seule classe, dite classe de base qui est la borne inférieure unique, pour la relation de subsomption, de l'ensemble des classes auquel il appartient (il appartient bien sûr également à ses super-classes).
- **R2-** tout objet du domaine ne peut être décrit que par les propriétés applicables à sa classe de base (ceci signifie conformément à notre modèle formel, si C est la classe de base d'un objet o seule les propriétés appartenant à $applic(C)$ peuvent être utilisées pour décrire o).

Soulignons que l'hypothèse R_2 n'impose en aucun cas que *toutes* les propriétés applicables d'une classe soient effectivement utilisées pour en décrire les instances. Le choix des propriétés à utiliser effectivement pour les instances d'une classe d'une BDBO, ce que l'on peut appeler son *schéma* ou son *modèle conceptuel* dépend de l'objectif particulier de la BDBO. Il est, par exemple, possible qu'une propriété définie comme applicable au niveau d'une classe A soit effectivement représentée dans le schéma de deux sous-classes (A_1 et A_2) de A, non dans celui d'une troisième sous-classe A_3. Ainsi, pour l'ontologie de la figure 3.1, on pourrait ne pas vouloir initialiser la propriété *gender* pour les instances de la classe *Employee*. Cette caractéristique d'ailleurs constitue une des grandes différences entre les BDBOs et les BDOOs. Dans une BDOO, toutes les classes ont un schéma d'instances et les propriétés qui caractérisent les instances sont constituées *automatiquement* de *toutes* les propriétés applicables des classes. La seule possibilité pour que deux sous-classes partagent une *même* propriété est que cette propriété soit *héritée* d'une super-classe commune. Elle est alors *également héritée par toutes les sous-classes* de cette super-classe.

3.1.2 Notre proposition de schéma des instances des classes.

Notre hypothèse de typage fort R_1 et R_2 permettant de définir, pour chaque classe, un schéma maximum permettant de décrire toutes les instances de cette classe. Il s'agit du schéma comportant toutes les propriétés applicables de la classe.

Néanmoins, comme nous l'avons souligné, le choix des propriétés applicables effectivement utilisées dans une BDBO dépend de l'objectif applicatif de la BDBO. Pour chaque classe donnée, il se peut que seul un sous-ensemble des propriétés soit effectivement utilisé pour décrire tout ou partie des instances.

Il convient de remarquer que ces schémas *ne sont pas nécessairement liés par une relation d'héritage*. Ainsi, comme nous l'avons noté ci-dessus, deux sous-classes A_1 et A_2 d'une classe A peuvent utiliser toutes deux une propriété P, définie comme applicable pour A, quand une troisième sous-classe A_3 peut ne pas l'utiliser. Utiliser l'héritage comme cela est fait dans la représentation verticale usuelle dans les BDBOs (où chaque propriété est représentée dans la table

(a)

Objet1:student
noSS:string=02489 name:string=Woopy lastname:string=Golder age:int=35 gender:char=F

Objet2:employee
noSS:string=13457 name:string=Jean lastname:string=Claude age:int=40 salary:double=1500 service:string=CRCFAO

(b)

id_student					
OID	**id_noSS**	**id_name**	**id_lastname**	**id_age**	**id_gender**
100	02489	Woopy	Golber	35	F

id_employee						
OID	**id_noSS**	**id_name**	**id_lastname**	**id_age**	**id_salary**	**id_service**
103	13457	Jean	Claude	40	1500	CRCFAO

FIG. 3.9 – (a) Exemple d'une population d'instances de classes et (b) leur représentation dans la table de leur classe.

correspondant au niveau où la propriété est définie) s'avère peu adaptée. Cela reviendrait, en effet, à représenter également la propriété P pour la classe A_3.

Nous proposons d'opter pour une technique de représentation se basant sur l'approche de *représentation horizontale*. Dans cette approche :

- chaque classe qui se trouve être une classe de base d'au moins une instance est associée à une table,
- le schéma de cette table comporte toutes les propriétés applicables de la classe qui sont utilisées par *au moins une des instances de la classe*.

Notre approche est donc caractérisée par les points suivants.

- **Tables des classes.** On ne crée des tables que pour un sous-ensemble des classes de l'ontologie : celles associées à des instances.
- **Colonnes des propriétés.** On ne crée des colonnes que pour un sous-ensemble des propriétés applicables des classes : celles utilisées par au moins une instance de la classe.

Le choix des classes et des propriétés est normalement fait par le concepteur de la base de données. Il sélectionne dans l'ontologie les classes pour lesquelles il souhaite créer des instances, ainsi que les propriétés à initialiser pour chaque table de classe. L'ensemble des classes et des propriétés sélectionnées forme le schéma de la base de données encore appelé le modèle conceptuel.

Lorsque la BDBO est destinée à charger automatiquement le contenu d'une autre BDBO basée sur la même ontologie normalisée, le schéma peut également être défini dynamiquement en fonction des classes et des propriétés effectivement référencées dans les données à base ontologiques lues.

La figure 3.9 montre un exemple de données à base ontologique pour l'ontologie définies dans la figure 3.1. Nous n'avons créé de tables que pour les classes *Student* et *Employee*. La table de la classe *Employee* n'initialise pas la colonne *gender* de sa super-classe *Person*.

3.1.3 Représentation du modèle conceptuel des données

La proposition de représentation des instances définie dans la section précédente nécessite d'être complétée. En effet :

1. Lors d'une requête de type polymorphe i.e., réalisée au niveau d'une classe non feuille, comment pourrait-on identifier les tables des classes à interroger ?

   Exemple. Retrouver toutes les instances de la classe *Person*. Comment savoir que seule les tables des classes *Student* et *Employee* sont concernées par la requête et que celle-ci doit se traduire sous la forme :

   SELECT ID **FROM** Student
   UNION
   SELECT ID **FROM** Employee

2. si l'on admet que l'utilisateur doit pouvoir faire des requêtes sur les propriétés applicables d'une classe, comment, lors d'une *requête polymorphe* au niveau d'une classe non feuille, peut-on savoir qu'une propriété est utilisée ou non pour certaine sous-classe ?

   Exemple. Retrouver le *nom*, l'*age* et le *sexe* (*gender*) de toutes les instances de la classe *person* (y compris celles de ses sous-classes). Comment savoir que la table *Employee* ne définit pas la propriété *gender* et que la requête doit s'écrire :

   SELECT ID, name, age, gender **FROM** Student
   UNION
   SELECT ID, name, age, NULL as gender **FROM** Employee

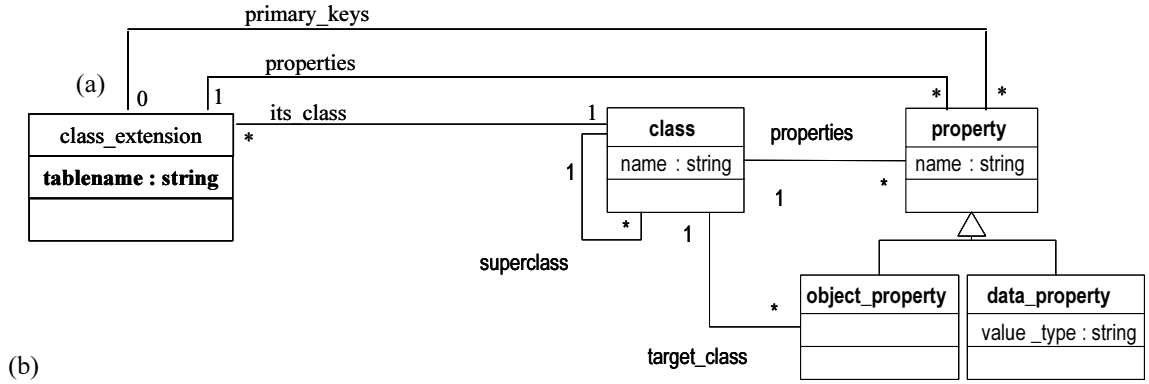
Les problèmes évoqués ci-dessus, résultent de l'absence d'information concernant le schéma des instances des classes d'ontologie nécessaire pour la génération des requêtes. Deux solutions sont possibles pour accéder au schéma des instances.

- **Par l'accès à la méta-base** de la base de données. La *méta-base* est la partie traditionnelle des bases de données dans laquelle sont stockées des méta-données utiles pour le SGBD pour la gestion de données. Elle contient entre autres une table des tables et une table pour les colonnes des tables de la base de données. Ces tables peuvent être exploitées pour l'identification des tables et des colonnes impliquées dans les requêtes précédentes.
  L'inconvénient de cette solution est que la structure de la *méta-base* n'est pas la même d'un fournisseur de bases de données à un autre. Par exemple, en PostgreSQL, la table des tables est nommée *pgtable* et en SQL Server *sysobjects*. Du point de vue portabilité, l'utilisation de la *méta-base* n'est donc pas la solution idéale.

- **Par la représentation du modèle conceptuel des instances** dans la base de données. A défaut d'utiliser la *méta-base*, l'on pourrait représenter le modèle conceptuel des instances des différentes classes. La figure 3.10a, présente un diagramme de classes UML décrivant un modèle simplifié de schéma des instances des classes. Dans cette représentation, on peut remarquer que chaque extension de classe référence sa classe à travers l'attribut *its_class*. L'ensemble des propriétés initialisées par la classe est donné par la relation *properties*. La relation *primarykeys* permet de spécifier les propriétés de la classe qui formeront la clé qui

permettront d'identifier une instance. La figure 3.10b montre un exemple de schéma de base de données issu du diagramme UML.

Notre proposition est de représenter effectivement le modèle conceptuel des instances, ce qui permettra en plus, ainsi que nous le verrons dans la section 3.2, de *garder la trace de l'historique des différents schémas d'une même classe.*



(b)

class_extension		
ID	its_class	tablename
1	id_person	id_person
2	id_employee	id_employee

Primary keys	
IDClass_Extension	IDProperty
1	id_noSS
2	id_noSS

properties	
IDClass_extension	IDProperty
1	id_noSS
1	id_name
1	id_lastname
1	id_age
1	id_gender
2	id_noSS
2	id_name
2	id_lastname
2	id_age
2	Id_salary
2	id_service

FIG. 3.10 – Exemple de représentation du modèle conceptuel des instances des classes de la base de données

3.1.4 Relations entre la partie ontologie et la partie données

3.1.4.1 Position du problème

Les données à base ontologique et les ontologies étant gérées séparément, il est nécessaire d'établir deux mécanismes.

(1) Un mécanisme de *liaison bilatère entre les deux parties* doit permettre d'accéder aux données d'une partie via l'autre. L'un des principaux objectifs des BDBOs étant de représenter explicitement la sémantique des données à travers l'ontologie, ce mécanisme doit permettre à la fois d'interroger les données à partir de l'ontologie et de présenter les données en termes de l'ontologie.

(2) Un mécanisme de cohérence doit permettre d'assurer la cohérence et l'intégrité des don-

nées des instances. Des contraintes essentielles sur les données sont en effet représentées au niveau de l'ontologie. Il est donc nécessaire d'assurer le respect de ces contraintes par les données dans la base de données. Par exemple, supposons qu'au niveau de la classe *Student* de l'ontologie, nous ayons défini une contrainte spécifiant que l'*age* des étudiants doit être compris entre *16* et *100* ans, il convient que parmi la population d'instances de la classe, il n'y ait pas d'instances violant cette contrainte. D'autres contraintes de ce type doivent également vérifiées comme l'unicité des valeurs de propriétés, ou encore les contraintes d'intégrités référentielles.

3.1.4.2 Notre proposition

Nous proposons de représenter le mécanisme de liaison bilatère entre les ontologies et leurs instances par l'intermédiaire de deux fonctions partielles :

- **Nomination :** classe $\cup$ propriété $\rightarrow$ table $\cup$ attribut ;
- **Abstraction :** table $\cup$ attribut $\rightarrow$ classe $\cup$ propriété ;

La fonction de *Nomination* associe à tout concept de niveau ontologique les éléments (table / colonne) qui en représentent les instances. La fonction d'*Abstraction* associe à tout élément de données le concept de l'ontologie qui en définit le sens. Ces deux fonctions sont partielles car :

- certaines classes et / ou propriétés peuvent ne pas être représentées. Comme nous l'avons expliqué dans la section 3.1.2, seul un sous-ensemble des classes et des propriétés est sélectionné par le concepteur de la base de données pour constituer son modèle conceptuel.
- certaines tables et / ou attributs, de nom prédéfinis, correspondent à des informations de type système et non à des éléments ontologiques, c'est le cas par exemple : (1) des tables qui permettent de représenter le schéma des instances des classes (cf. figure 3.10) ; (2) des colonnes).

Ces fonctions pourront ensuite être utilisées en particulier pour connecter les contraintes (ontologiques) aux données. La mise en œuvre de ces fonctions par adressage fonctionnel (calculé) ou associatif (table de correspondances) dépendent ensuite des choix d'implémentation.

3.2 Gestion du cycle de vie des instances

3.2.1 Position du problème

Comme les concepts des ontologies, les instances des concepts subissent également des changements. Ces changements sont de trois ordres :

1. Les changements engendrés par les modifications effectuées dans l'ontologie. Par exemple une propriété applicable peut être définie comme obsolète. Sa valeur ne sera alors plus intégrée dans le nouveau schéma d'instances.
2. Les changements de schéma. Même sans qu'une propriété ne change au niveau de l'ontologie, il peut apparaître pertinent de ne plus fournir sa valeur car elle n'est pas considérée comme utile dans le contexte applicatif. Par exemple, tous les objets mécaniques ont une masse. Sa valeur n'est en général pas fourni pour une "vis"' car elle n'est pas considérée comme cruciale.

3. Les changements sur les populations d'instances. Ces modifications peuvent consister à ajouter/supprimer des instances.

Pour les besoins de certaines applications (et c'est en particulier le cas pour les bases de données de composants industriels), il peut être indispensable de tracer le cycle de vie des instances des concepts car différentes instances, introduites dans la BDBO à différents instants, peuvent devenir obsolètes, à partir d'un certain moment. Ceci signifie qu'une BDBO devrait pouvoir retrouver toutes les instances de chaque classe pour chaque version de celle-ci et dans la structure qu'avaient les instances à l'époque de cette version. Cette gestion du cycle de vie des instances des classes demande de gérer l'historique tant de la structure des classes que de leurs instances. Cet historique concerne précisément :

1. Les définitions des classes de l'ontologie : on devrait pouvoir avoir la date où elles ont été définies, les propriétés qui leur étaient applicables, etc.
2. Les schémas des instances des classes : étant donné que ceux-ci peuvent être modifiés d'une version de classe à une autre, on doit pouvoir retrouver les propriétés ayant appartenu à telle ou telle version du schéma d'une classe et, si possible les dates de leur insertion ou suppression dans le schéma des instances de la classe, etc.
3. La population des instances : étant donné que la même instance peut appartenir à des versions successives d'une classe, on doit pouvoir retrouver la version où elle a été insérée, les versions des classes où elle a été valide, etc.

Ces informations sur le cycle de vie des instances des classes sont indispensables pour pouvoir reconstituer les populations d'instances d'une classe à une version quelconque de la base de données. Le modèle que nous proposons doit prendre en considération cet aspect.

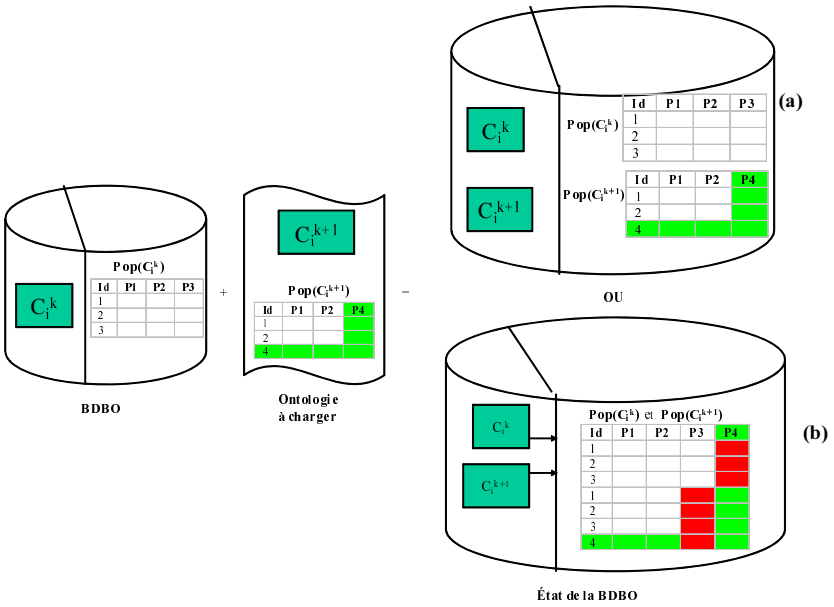


FIG. 3.11 – Problématique de la gestion du cycle de vie des instances des classes

3.2.2 Solutions existantes

La problématique de la gestion du cycle de vie des instances a été largement discutée dans les bases de données temporelles [144] et les entrepôts de données [167, 163, 39, 33]. Dans la littérature, deux principales de gestions des versions des instances ont été proposées :

1. **Stockage explicite.** Dans cette approche [163, 39, 14, 33], toutes les populations (i.e., toutes les tables) correspondant aux versions successives d'une même classe sont explicitement stockées dans la base de données (cf. figure 3.11a). Les principaux avantages de cette solution sont que : (1) elle est facile à implémenter et (2) le traitement des requêtes est rapide dans le cas où l'on précise la (ou les) version(s) sur laquelle porte la recherche. Par contre (1) le coût peut être important si la requête nécessite un parcours de toutes les versions disponibles dans la base de données et (2) le coût de stockage des données est élevé à cause de la duplication des données.

2. **Stockage implicite.** Dans cette approche [163, 33], toutes les populations correspondant aux versions successives d'une même classe sont stockées dans une même table dans laquelle une (ou plusieurs) colonne(s) additionnelle(s) sont ajoutées afin d'identifier à quelle version appartient chaque instance (cf. figure 3.11b). La structure de la table est définie en faisant l'*union* des propriétés de toutes les versions du concept. Les instances sont complétées par des valeurs nulles pour les colonnes qu'elles n'initialisent pas. L'avantage de cette approche est qu'elle évite le parcours de plusieurs tables de versions de classes. Les inconvénients de cette approche sont les suivants :
   (1) le problème de duplication des données existe comme dans la solution précédente, i.e., la "même" instance peut être représentée plusieurs fois,
   (2) le calcul du schéma des instances appartenant à une version donnée n'est pas possible ;
   (3) l'identification du cycle de vie d'une "même" instance est difficile [163],
   (4) une valeur nulle peut avoir deux significations différentes : soit la valeur de la propriété n'était pas fournie, soit la valeur de la propriété est nulle.

3.2.3 Notre proposition

La solution que nous proposons est basée sur la deuxième approche. Toutes les versions des instances sont stockées dans une unique table. Néanmoins, les mécanismes de gestion sont complètement modifiés et nous résolvons les inconvénients du stockage implicite de la manière suivante :

(1) **Duplication des instances.** Pour résoudre ce problème, nous proposons d'associer à chaque classe la définition d'une ou plusieurs contraintes d'unicité permettant une *identification sémantique unique*. Grâce à cette clé sémantique, chaque instance de classes est identifiée de façon unique par un sous-ensemble de propriétés. En se basant sur cette clé, on peut éviter la duplication des instances d'une classe si elles continuent à exister dans une nouvelle version. Cela est réalisé en vérifiant qu'il n'existe dans la table de la classe une instance ayant les mêmes valeurs pour l'ensemble des propriétés de l'une quelconque des clés candidates.

(2) **Calcul du schéma des instances.** Ce problème est résolu en archivant le schéma des instances de classe pour chaque version. Cet archivage offre donc la possibilité de retrouver la structure des instances d'une classe pour une version quelconque.

(3) **Tracé du cycle de vie des instances.** La connaissance de la période de validité d'une instance est résolue en ajoutant à chaque instance la liste des versions de classes pour lesquelles cette instance était valide.

(4) **Ambiguïté de la sémantique de la valeur nulle.** Cette ambiguïté est résolue grâce au schéma des instances que nous archivons dans la base données. Ce schéma nous permet, en effet, de déterminer les propriétés exactes initialisées pour une instance d'une version donnée.

3.3 Extraction des données à base ontologique

3.3.1 Position du problème

Il s'agit ici, dans le contexte du besoin F_5 identifiée en la section 2, d'extraire des sous-ensembles d'instances des classes de la base de données. Lors de l'extraction d'une instance, il doit être possible d'assurer que toutes les définitions des propriétés des instances sont extraites simultanément présentes dans le référentiel. En effet, si les instances extraites de la BDBO doivent être intégrées dans d'autres systèmes, la définition des propriétés doit être disponible. Si dans le système receveur, la définition existe, elle est alors disponible. Dans le cas contraire, celles-ci doivent être extraites de la BDBO en même temps que les instances.

L'algorithme décrit dans la section 2.5 pour l'extraction des concepts n'extrait malheureusement pas toutes les définitions des propriétés pour respecter l'exigence ci-dessus. Illustrons cela à l'aide de l'exemple suivant (cf. figure 3.12).

Exemple :
Soit une instance I d'une classe A_1 qui possède une propriété a_2 dont le co-domaine est une classe B_0. Par polymorphisme, a_2 peut référencer une instance d'une sous-classe de B_0 : B_1. En appliquant l'algorithme précédent pour extraire la classe A_1, la classe B_1 et toutes ses propriétés ne seront pas extraites, alors qu'elles sont indispensables pour l'interprétation de l'instance I.

La **Zone 1** de la figure 3.12 contient les concepts extraits en appliquant l'*algorithme* dans la section 2.5 pour l'extraction de la classe A_1. La **Zone 2** contient toutes les instances extraites lors de l'extraction de la population d'instances de la classe A_1. La **Zone 3** contient les concepts non extraits de la base de données mais qui sont nécessaires pour l'interprétation des instances de A_1. Remarquons sur la figure que l'instance a_{11} de A_1 référence une instance de la classe B_1 qui décrit la propriété (b_3) qui malheureusement n'est pas extraite de la BDBO. L'algorithme d'extraction des instances, doit prendre en compte le polymorphisme de sorte à pouvoir rendre disponible la définition de tous les concepts qui permettront d'interpréter les instances extraites

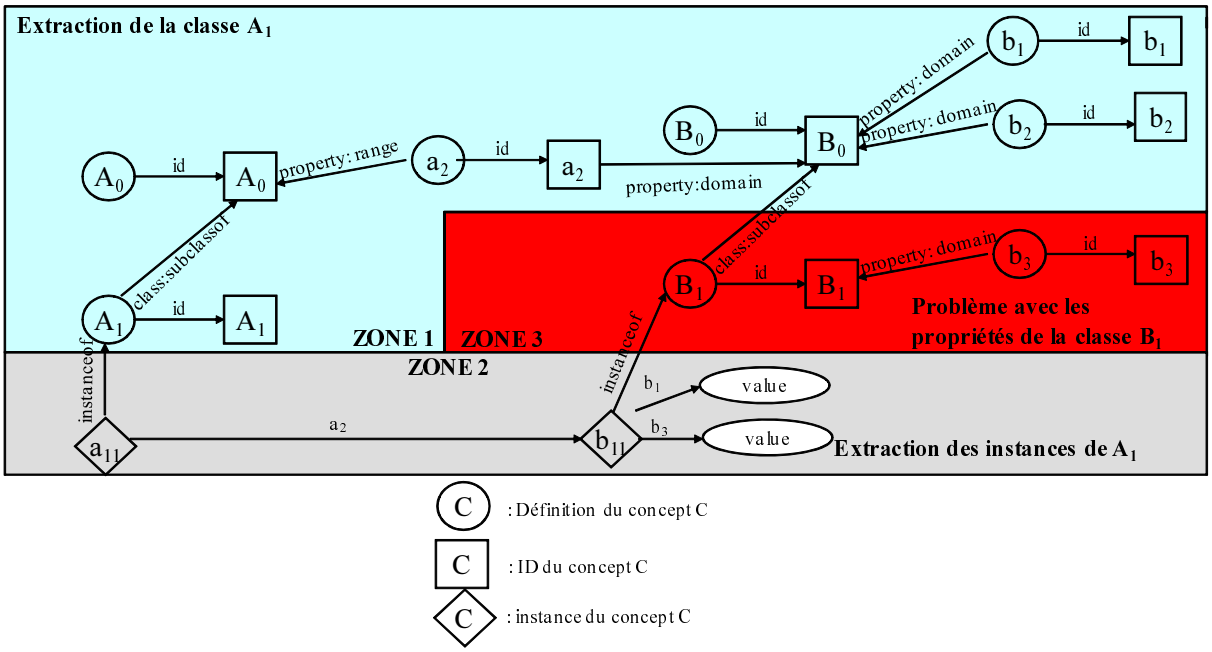


FIG. 3.12 – Extraction des instances des classes

de la BDBO.

3.3.2 Proposition d'un nouvel algorithme

Nous donnons ici un nouvel algorithme qui permet d'assurer la complétude sémantique d'un ensemble de données à base ontologique. Soit R un ensemble de requêtes pouvant s'appliquer sur les instances de la BDBO. Les requêtes sont des prédicats qui permettent de filtrer les instances des classes. Nommons $extraireInstance : R \rightarrow I$[17], la fonction d'extraction des instances et $propriété\_valué : I \rightarrow 2^P$ la fonction qui associe à chaque instance les propriétés qui sont valuées pour cette instance. Soit $Pop^{-1} : I \rightarrow 2^C$ l'inverse à la fonction Pop qui permet d'associer à chaque instance la (ou les) classe(s) à laquelle elle appartient[18]. L'algorithme 13 que nous proposons extrait en plus des instances tous les concepts nécessaires à leurs interprétations.

Algorithme 13 Algorithme d'extraction des instances des classes.

$\forall r \in R$
  $\forall i \in extraireInstance(r)$
    $\forall c \in Pop^{-1}(i) \Rightarrow extraire(c_{def})$
    $\forall p \in propriété\_valué(i) \Rightarrow extraire(p_{def})$

A l'issue cette discussion sur les objectifs et les exigences que nous nous sommes fixés d'at-

[17] I = l'ensemble des instances de la base de données.
[18] Notons dans une ontologie PLIB, toute instance n'appartient qu'à une et une seule classe

teindre en plus de la représentation des données à base ontologique, nous faisons dans la section suivante une synthèse des différentes propositions faites sous forme d'une architecture de bases de données à base ontologique.

4 Notre proposition de modèle d'architecture de BDBO : OntoDB

Nous pouvons maintenant synthétiser l'ensemble des propositions présentées dans ce chapitre sous forme d'un modèle d'architecture pour les BDBOs que nous avons conçu pour répondre aux besoins identifiées dans la section 2 et que nous proposons dans le cadre de notre thèse. Cette architecture répond à toutes les objectifs (O_1 - O_4) que nous nous sommes fixées dans la section 1.1.

Cette architecture vise à faire la synthèse de deux univers.

- L'univers des bases de données, où les données sont associées à des schémas de données qui permettent de traiter de façon efficace de très gros volume de données.
- L'univers des données à base ontologique, issus du Web sémantique, qui permettent de réunir, au sein d'un même système, des données, appelées individus et des ontologies qui en donnent le sens. Dans ces systèmes, chaque individu est porteur de sa propre structure et la notion de schéma de données au sens usuel des bases de données n'est pas présent : chaque individu est décrit par des ensembles de triplets spécifiques.

L'idée centrale de notre synthèse est d'introduire dans les BDBOs le niveau schéma de données qui constitue la structure commune, éventuellement au prix d'un certain nombre de valeurs nulles, de l'ensemble des instances d'une classe de l'ontologie. Cette idée s'applique très naturellement au premier domaine d'application dont nous sommes portés : les catalogues de composants industriels. Après une étude des autres domaines, elle semble en fait également applicable à beaucoup d'autres domaines tels que l'annotation des documents ou les catalogues des portails du Web sémantique [6].

La figure 3.13 présente les quatre parties qui composent notre proposition d'architecture, appelée *OntoDB*, pour les bases de données à base ontologique :

1. La **partie *méta-base.*** La *méta-base*, souvent appelée '*system catalog*', est une partie traditionnelle des bases de données classiques. Elle est constituée de l'ensemble des tables système. Ces tables sont celles dont le SGBD se sert pour gérer et assurer le fonctionnement de l'ensemble des données contenues dans la base de données. Dans une BDBO, toutes les tables et les attributs définis dans les trois autres parties sont documentés dans la *méta-base.*

2. La **partie *données.*** Elle représente les objets du domaine. Ceux-ci sont décrits en termes d'une classe d'appartenance et d'un ensemble de valeurs de propriétés applicables à cette classe. C'est ce que nous appelons les *données à base ontologique.* Les propriétés des classes effectivement représentées (dites *propriétés utilisées*) sont celles qu'au moins une instance

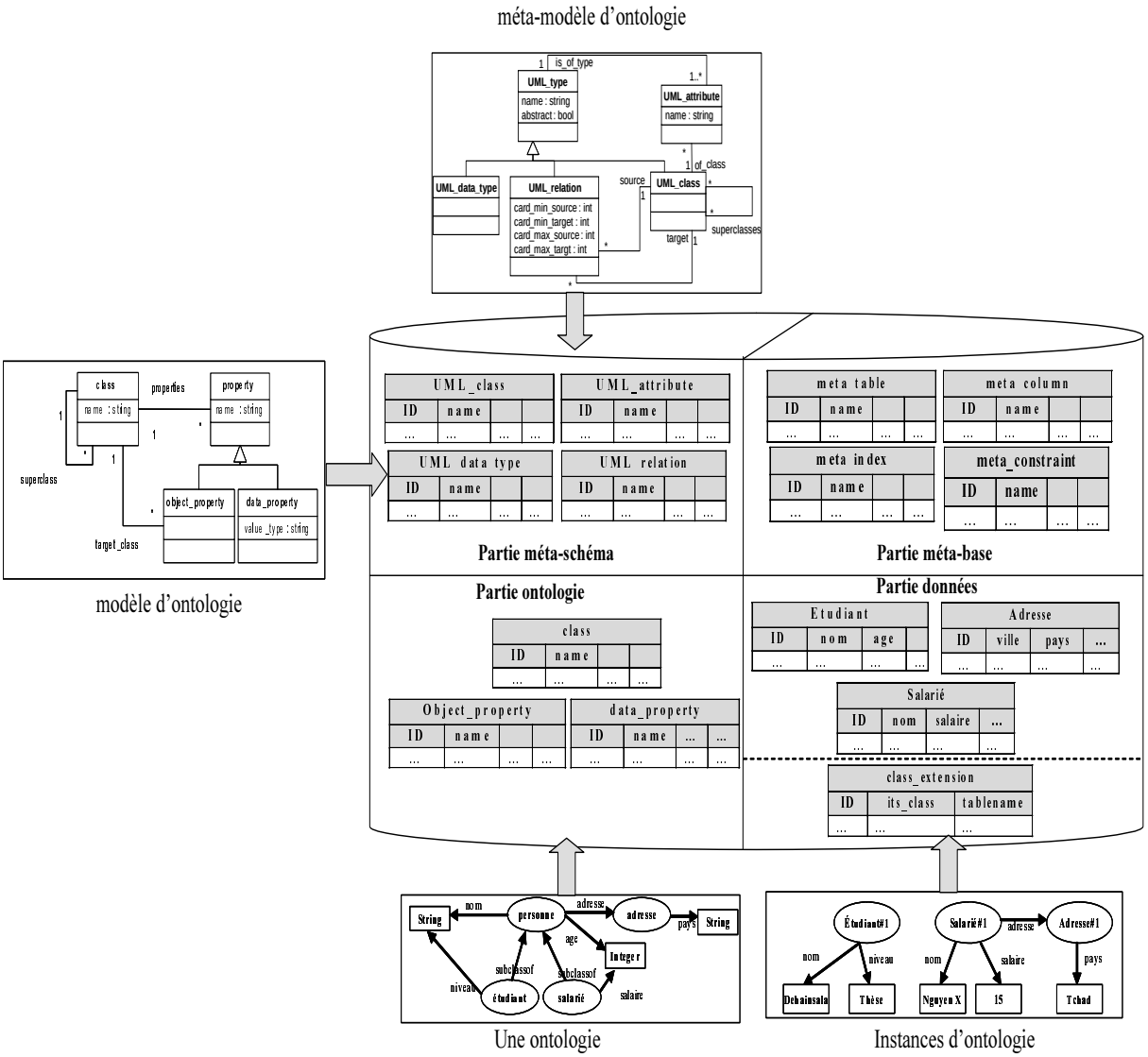


FIG. 3.13 – Architecture OntoDB

de la classe initialise. Une spécificité de la partie *données* d'*OntoDB* est de permettre de gérer le cycle de vie des objets du domaine.

3. La **partie *ontologie.*** Elle contient les ontologies définissant la sémantique des différents domaines couverts par la base de données, ainsi, éventuellement, que l'articulation, représentée par des relations subsomption, de ces ontologies locales avec des ontologies externes (par exemple normalisées). Les ontologies représentées dans cette partie peuvent être versionnées et archivées. L'accès aux données de cette partie est possible grâce à deux types d'interface de programmation : une *API à liaison tardive* et deux APIs *API à liaison préalable*. Toutes ces APIs permettent l'accès aux concepts des ontologies, et ce, en différentes versions.

4. La **partie *méta-schéma.*** Elle représente, au sein d'un modèle réflexif, à la fois le modèle d'ontologie utilisé et le méta-schéma lui-même. Le *méta-schéma* est pour la partie *ontologie*, ce qui est la *méta-base* pour le système. Il permet, au moyen d'une API qui exploite son contenu, de gérer de façon générique les concepts des ontologies représentés dans la partie *ontologie*. Le schéma de données de cette dernière étant susceptible d'évoluer ou d'être modifié, l'autoreprésentation du méta-modèle du modèle d'ontologie permet de rendre certains traitements génériques et/ou indépendamment du modèle d'ontologie utilisé. Enfin, l'existence du méta-schéma permet d'intégrer dans une même base de données des ontologies issues de modèles d'ontologies différents sous réserve bien sûr que ceux-ci soient définis dans un même formalisme de modélisation.

5 Conclusion

Nous avons présenté dans ce chapitre un nouveau modèle d'architecture pour les BDBOs appelé *OntoDB*. Le modèle d'architecture *OntoDB* se compose en quatre parties qui sont les parties *Ontologie*, *Méta-schéma*, *Données* et *Méta-base*. Les trois premières parties servent respectivement à représenter les ontologies, les modèles d'ontologies, les instances des ontologies et la dernière partie, encore appelée *system catalog*, est la partie des SGBDs qui contient la description des tables et des vues de la base de données. L'architecture *OntoDB* se distingue par rapport aux autres architectures sur les points suivants.

– **Représentation des données à base ontologique.** Les architectures existantes dans la littérature représentent toutes les instances, soit sous forme de triplet dans une table à trois colonnes (sujet, prédicat, objet), soit dans des tables binaires (id, valeur) pour chacune des propriétés des classes. Dans *OntoDB*, l'hypothèse de *typage fort* des propriétés, et de *mono-instanciation* permet d'associer à chaque classe un schéma de données constitué d'un *sous-ensemble* des propriétés applicables de la classe. Les instances des classes sont alors représentées dans *OntoDB* au sein d'une relation unique représentée dans des tables spécifiques créées pour chaque classe de la base.

  Cette approche de représentation des instances impose deux restrictions : (1) tout objet du

domaine est décrit par son appartenance à une classe de base, qui est la borne inférieure unique, pour la relation de subsomption, de l'ensemble des classes auquel il appartient (mono-instanciation). (2) Toute propriété doit être typée : elle doit être associée à un domaine et un co-domaine. La dernière hypothèse est très souvent utilisée dans d'autres systèmes [6]. Concernant la première hypothèse, d'une part elle est toujours effectuée dans le domaine des bases de données, et, d'autre part, elle n'interdit pas qu'un objet du monde réel soit décrit par plusieurs instances de classes. On peut en effet utiliser la technique dite de l'agrégat d'instances qui permet de représenter la même information que la technique de multi-instanciation. Cette représentation est d'ailleurs autorisée par la plupart des modèles d'ontologies, y compris le modèle PLIB qui nous intéresse particulièrement. Ces restrictions permettent par contre de gérer de façon efficace de très grands ensembles de données à base ontologie ce que ne permettait pas les méthodes antérieures [47].

– **Représentation du modèle conceptuel des données dans la base de données.** L'architecture *OntoDB* permet également de représenter explicitement le modèle conceptuel des données dans la base de données. Le modèle conceptuel est un sous-ensemble cohérent des concepts des ontologies sélectionnés par le concepteur. Il définit à la fois la *structure* et par sa référence à l'ontologie, la *sémantique* des données à base ontologique.

– **Versionnement des concepts des ontologies.** L'architecture *OntoDB* permet d'archiver et de manipuler les différentes versions des concepts des ontologies. La technique utilisée pour sa mise en œuvre nous a conduit à définir la notion de version courante. Cette approche représente la dernière version disponible de chaque concept et permet de consolider toutes les relations entre les différents concepts. Les *versions historiques* des concepts sont archivées dans la base de données et leurs relations avec les autres concepts sont maintenues pour conserver toute la sémantique des différentes versions des ontologies.

– **Versionnement des instances et gestion du cycle de vie des instances.** Comme les ontologies, les instances des classes sont aussi versionnées dans la base de données lorsqu'il y a des changements. Pour ce faire, chaque instance de la base de données est associée à une liste de numéros de version de la classe à laquelle elle appartient. Son modèle conceptuel est aussi également historisé.

– **Généricité par rapport aux modèles d'ontologies.** *OntoDB* a été défini de façon à supporter toute évolution et/ou changement du modèle d'ontologie utilisé. La réalisation de cette caractéristique, nous a conduit (1) à représenter dans la base de données au sein d'un méta-modèle réflexif à la fois les modèles d'ontologies ainsi que le méta-modèle de ces modèles ontologies et (2) à proposer une interface d'accès à liaison tardive pour accéder aux concepts de ces ontologies, indépendant donc de tout modèle particulier d'ontologie.

Notons que ce modèle d'architecture est un modèle logique qui n'est pas nécessairement implanté sur un unique système.

– Partie *ontologie* et partie *données* peuvent éventuellement être représentées dans deux

systèmes différents, coordonnés par une interface d'accès commune. Nous avons également réalisé un premier prototype selon cette architecture.

– Si le niveau *méta-schéma* est nécessaire en génération, ce niveau n'est réellement indispensable que lors d'un changement de modèle d'ontologie. On peut donc éventuellement effectuer les générations correspondant aux fonctions F_1, F_2, F_3 et F_4 à l'aide d'un système externe dans lequel la partie *méta-schéma* serait représentée. Notons cependant que, dans ce cas, il faudrait régénérer également l'*API à liaison différée* à chaque évolution du modèle d'ontologie, alors que ceci n'est pas nécessaire si elle peut accéder au *méta-schéma*.

Dans le reste de cette thèse, nous nous intéressons désormais au cas où les quatre parties sont effectivement représentées sur le même système, constituant un réel système de gestion de BDBO.

On peut constater que notre architecture de BDBO a de fortes similitudes avec l'architecture metadata du MOF (Meta Object Facility) [63]. Notre architecture est constituée des mêmes quatre couches superposées. La couche modèle M_1 de l'architecture MOF correspond à notre modèle conceptuel, sous-ensemble de l'ontologie. Ce niveau contient lui-même les instances M_0. La couche méta-modèle M_2 correspond au (méta-) modèle d'ontologie (figure 3.1), la couche méta-méta-modèle M_3 (MOF model) correspond au méta-modèle, lui-même réflexif, du langage de définition du modèle d'ontologie.

Notons que l'architecture *OntoDB* est parfaitement adaptée à la nouvelle méthode de conception des bases de données que nous avons proposée au chapitre 1. En effet, elle permet de représenter et de gérer les trois étapes de notre méthode de conception : l'ontologie, le modèle conceptuel et les données. Les bases de données conformes à notre architecture contiendront la sémantique des données qui y sont stockées. Celles-ci sont donc préparées pour d'éventuels processus d'intégration vers d'autres sources de données. Cette intégration peut être réalisée automatiquement comme cela a pu être montré dans ces différents travaux [18, 16, 17, 15] effectués au LISI et entièrement basés sur le modèle OntoDB.

Notons enfin que notre architecture permet également (1) d'exporter, (2) d'importer des instances non nécessairement définies dans le modèle local d'ontologie.

Nous discutons dans le chapitre suivant de la validation de l'architecture *OntoDB* à travers le prototype que nous avons implémenté sur le SGBD PostgreSQL.

Chapitre 4

Implémentation de l'architecture OntoDB

Introduction

Dans le chapitre précédent, nous avons défini et présenté l'architecture de base de données à base ontologique *OntoDB*. Cette architecture est constituée de quatre parties : *méta-base*, *données*, *méta-schéma* et *ontologie*. Nous discutons dans ce chapitre de l'implémentation d'un prototype de système de gestion de base de données à base ontologique basé sur le modèle d'architecture *OntoDB* et dans lequel les quatre parties de l'architecture logique sont implantées dans le même SGBD [85].

L'objectif de ce chapitre est à la fois de montrer la faisabilité du modèle que nous avons défini et de mettre en évidence les services qu'il permet d'offrir à l'utilisateur, en particulier du point de vue de la facilité d'accès aux données. Concernant le premier point, le fait de manipuler des ontologies définies de façon formelle va nous permettre de générer les principaux composants de notre architecture à l'aide des techniques d'ingénierie dirigée par les modèles (IDM). Concernant le deuxième point, nous montrons comment la disponibilité à la fois de l'ontologie et du modèle conceptuel au sein de la base de données permet de transformer les méthodes classiques d'accès aux données, tant interactive que par programme, existant usuellement dans les bases de données.

Le prototype que nous décrivons dans ce chapitre est basé sur le modèle d'ontologie PLIB. Le SGBD support qui servira pour la représentation des ontologies et des données à base ontologique est le SGBD relationnel objet PostgreSQL.

Le chapitre s'organise comme suit. Dans les sections suivantes, nous discutons successivement de la mise en œuvre des différentes parties formant le modèle d'architecture *OntoDB* : *ontologie* (section 1), *méta-schéma* (section 2), et *données* (section 3). Dans chacune de ces sections, nous décrivons les composantes nécessaires pour leur mise en œuvre, nous présentons les différentes alternatives possibles et les solutions que nous avons retenues dans notre implémentation. Dans la section 4, nous discutons des différents modes d'accès au contenu d'une base de données à base ontologique offerts par notre prototype. Nous décrivons successivement (1) les principales

interfaces fonctionnelles (API) que nous avons implémentées pour accéder aux différentes parties des BDBOs, et (2) les interfaces graphiques génériques développées pour permettre la création, l'édition, l'importation, l'exportation et l'interrogation des ontologies et des données à base ontologiques. Ces accès sont offerts au niveau connaissance, et non au niveau logique comme c'est le cas dans les bases de données usuelles. Le chapitre s'achève par une conclusion.

1 Représentation de l'ontologie

Cette section décrit la mise en œuvre des différents composants indispensable pour la représentation des ontologies dans la base de données. Cette représentation nécessite l'implémentation de composants permettant :

1. **la définition de la structure des tables** dans lesquelles les concepts des ontologies seront insérés,
2. **la lecture d'ontologies sous un format d'instances** pour les insérer dans la BDBO,
3. **l'extraction des concepts d'ontologies dans une BDBO** pour les écrire dans des fichiers physique sous un certain format.

Nous discuterons plus en détail, à la section 4, les différences méthodes d'accès aux données des ontologie.

L'implémentation de ces différents composants devra être effectuée de façon à atteindre les principaux objectifs que nous nous sommes fixés au chapitre 3 section 1. Tout particulièrement, elle devra permettre de rendre générique le système par rapport au modèle d'ontologie utilisé même si le premier modèle d'ontologie visé, et implémenté par nous, sera PLIB.

1.1 Utilisation de l'IDM en EXPRESS pour la mise en œuvre de notre architecture

C'est en vue d'atteindre cet objectif que nous avons opté pour l'utilisation des techniques de l'Ingénierie Dirigée par les Modèles (IDM) qui offrent la possibilité de transformer des modèles et de générer des programmes à partir de modèle. Nous utiliserons tout au long de ce chapitre l'environnement d'IDM EXPRESS présenté dans le chapitre 1 section 3.5 [46]. L'environnement d'IDM EXPRESS que nous avons utilisé dans le cadre de notre implémentation est l'environnement ECCO[147].

La structure de l'environnement logiciel que nous serons amenés à mettre en œuvre se présente comme sur la figure 4.1. Sur celle-ci, on peut remarquer que le point départ du traitement est un modèle EXPRESS quelconque fourni à l'environnement. Ce modèle peut, en particulier, être le modèle d'ontologie PLIB. Le modèle EXPRESS constitue donc un paramètre de la chaîne de traitements. Le modèle EXPRESS paramètre est d'abord compilé, puis converti sous forme d'instances d'un méta-modèle EXPRESS (le SDAI-dictionary-Schema [131] par exemple) par l'environnement d'IDM ECCO. Ces instances du méta-modèle peuvent donc être exploitées par des programmes génériques par rapport au modèle particulier traité. Ces programmes, eux-mêmes

écrits en EXPRESS (légèrement étendu) permettent de réaliser les différentes transformations nécessaires (exemple génération de schémas logiques, génération d'APIs, etc.).

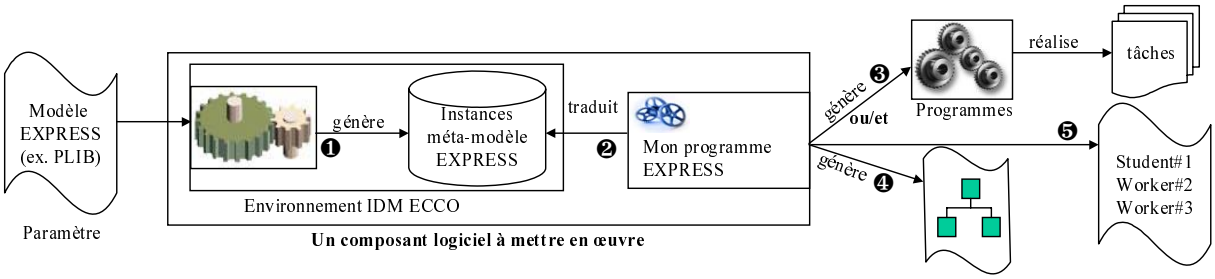


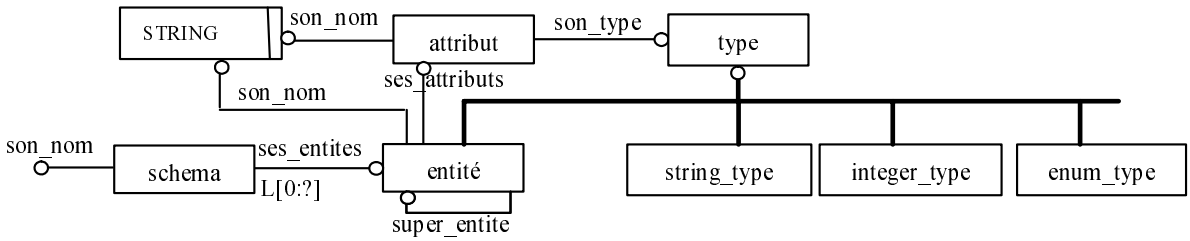
Fig. 4.1 – Environnement d'IDM ECCO

Les figures 4.2a et 4.2b montrent respectivement un exemple d'un méta-modèle (très simplifié) du langage EXPRESS et un exemple de modèle d'ontologie également simplifié. Dans la figure 4.2c, nous avons une instanciation du modèle d'ontologie sous forme d'instances EXPRESS du méta-modèle EXPRESS. La représentation des instances est voisine du format normalisé d'échange d'instances d'un modèle EXPRESS (cf. chapitre 1 section 2.1.6) sauf que pour rendre le contenu plus lisible, nous avons fait apparaître le nom de chaque attribut représenté (par exemple : "son_nom="). Enfin dans la figure 4.2d, nous présentons un algorithme exploitant les instances du méta-modèle.

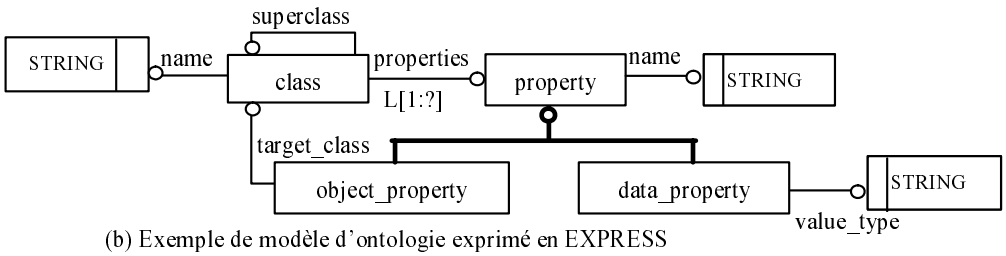
1.2 Définition de la structure des tables

Il s'agit ici de définir le modèle logique permettant le stockage d'ontologies dans les bases de données. Dans le chapitre précédent, nous avons précisé, à travers l'hypothèse **H2** (chapitre 3 section 1.2) que les ontologies auxquelles nous nous intéressons sont celles pouvant s'exprimer sous forme d'instances d'un modèle d'objet. La problématique de la représentation des ontologies dans une base de données revient donc à la représentation de données objets dans une base de données (relationnelle, ou autre). Ce problème a été discuté dans le chapitre 1 où nous avons montré que plusieurs approches existaient pour la représentation des objets dans des bases de données relationnelles ou relationnelles-objets. Toutes ces approches se résument par la définition des règles de correspondances [134, 141, 97, 53, 108, 54, 145] entre les mécanismes objets et ceux des modèles relationnels, ou relationnels-objets.

Chaque formalisme objet disposant de certaines spécificités, il est indispensable de définir également des règles de correspondances de celles-ci vers les mécanismes du modèle logique cible s'il en n'existe pas dans la littérature. Ceci était justement le cas pour le langage EXPRESS et le SGBD que nous avons utilisé (PostgreSQL). Il nous a donc fallu définir les règles de correspondances entre ces deux univers.



(a) Exemple de méta-modèle EXPRESS en EXPRESS



(b) Exemple de modèle d'ontologie exprimé en EXPRESS

```
#1  = schéma (son_nom ="OntoModel", ses_entités=(#2,#3,#4,#5));
#2  = entité (son_nom ="property", super_entité=?, ses_attributs=(#6));
#6  = attribut (son_nom ="name", son_type=#10);
#10 = string_type();
#3  = entité (son_nom ="data_property", super_entité=#2, ses_attributs=(#7));
#7  = attribut (son_nom ="value_type", son_type=#10);
#4  = entité (son_nom ="class", super_entité=?, ses_attributs=(#11,#12,#13));
#5  = entité (son_nom ="object_property", super_entité=#2, ses_attributs=(#8));
#8  = attribut (son_nom =" target_class", son_type=#5);
....
```

(c) Instanciation du modèle d 'ontologie sous forme d 'instances du méta-schéma EXPRESS

```
Soit s ∈ Schema
∀ ent:entité ∈ s.ses_entités;
{ExecutionActionSurEntité(ent);
 ∀ att:attribut ∈ ent.ses_attributs ;
  SWITCH(TYPEOF(att.son_type)) {
      case "String_Type"   : ExecutionAction_String_Type(att);
      case "Integer_Type" :  ExecutionAction_Integer_Type(att);
      case "Entity_Type"   :  ExecutionAction_Entity_Type(att);
      case ...
    }
ExecutionActionSuper_Entité(ent.super_entité);
…
}
```

(d) Exemple de programme exploitatant les instances du méta-schema EXPRESS

FIG. 4.2 – (a) Exemple simplifié de méta-modèle EXPRESS. (b) Exemple simplifié de modèle d'ontologie. (c) Instanciation du méta-modèle EXPRESS avec le modèle d'ontologie. (d) Algorithme exploitant les instances du méta-modèle EXPRESS

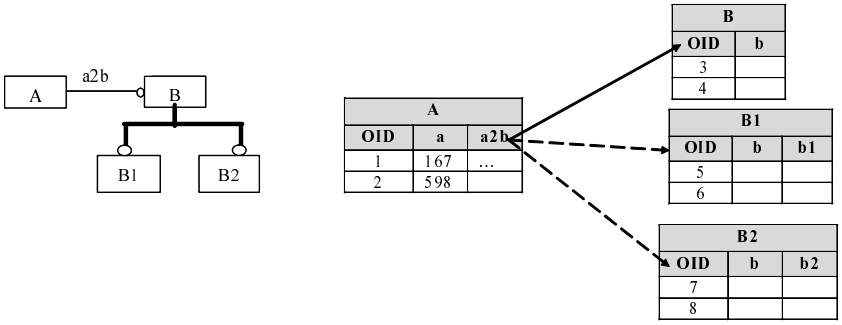


FIG. 4.3 – Représentation PostgreSQL des instances d'une hiérarchie.

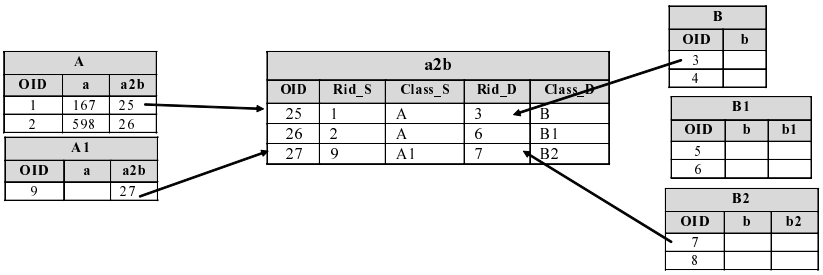


FIG. 4.4 – Un exemple d'une table d'aiguillage.

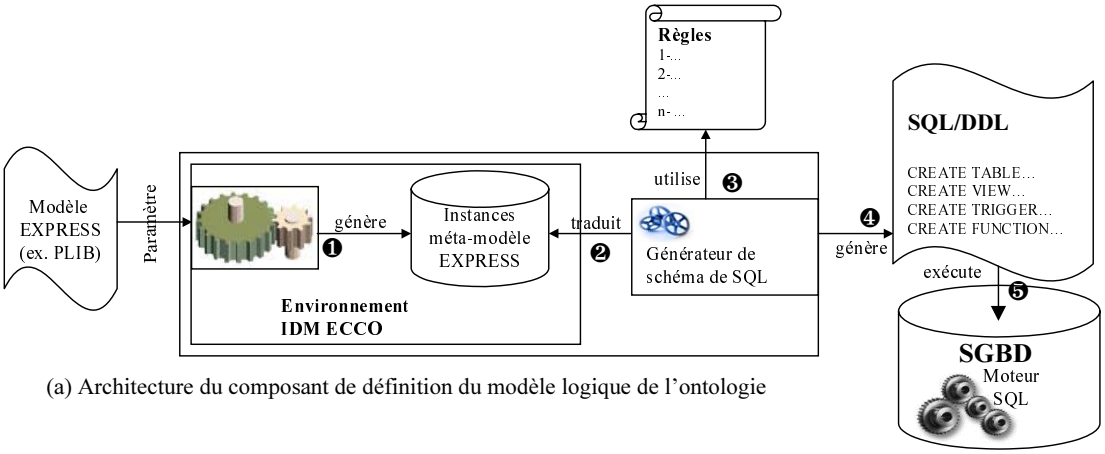
1.2.1 Schéma logique de représentation de l'ontologie dans PostgreSQL

PostgreSQL n'implémente que partiellement l'ensemble des mécanismes relationnel-objet définis dans la norme SQL99. Concernant l'héritage, le seul mécanisme disponible dans la version de PostgreSQL que nous avons utilisé est l'héritage de tables avec représentation globale des instances dans la table correspondante à leur classe effective. Les instances des classes ayant une même super-classe sont donc reparties dans des tables différentes. Cela est illustré à travers l'exemple de la figure 4.3.

Dans la figure 4.3, l'association *a2b* peut référencer des instances appartenant à B, B_1, ou B_2. En PostgreSQL, ceci ne peut être réalisé en utilisant une clé étrangère car elle ne peut référencer qu'une unique table. Pour remédier à ce problème, cela nous a donc été amené à représenter de telles associations polymorphes par l'intermédiaire de tables dites d'"aiguillage" telles que celle représentée à la figure 4.4. Le champ *Table_D*, de la table d'aiguillage, définit la table destination , le champ *rid_D* contient sous forme d'un entier la valeur de la clé de l'enregistrement de destination. Les champs *table_S* et *rid_S* jouent le même rôle pour la (ou la hiérarchie de) table source de façon à rendre l'association traversable dans les deux directions. De telles tables d'aiguillages sont créées pour toutes associations quelques soient leurs cardinalités.

Afin d'accélérer les accès à l'association, la table A contient également un champ *a2b* dont le contenu est une clé étrangère (cas d'une association $1 : m$) ou une collection de clés étrangères (cas association $n : m$) sur la table d'aiguillage.

Remarque :
Générer le schéma logique des tables revient donc à définir de façon systématique comment chaque construction du langage EXPRESS doivent être représentées dans le modèle logique PostgreSQL. Ces règles de correspondances entre le langage EXPRESS et PostgreSQL sont présentées de façon complètes et détaillées dans l'annexe C.



(a) Architecture du composant de définition du modèle logique de l'ontologie

```
Soit s ∈ Schema
∀ ent:entité ∈ s.ses_entités;
{Affiche ("CREATE TABLE "+ent.son_nom+"(")
 Affiche("  OID  INT32,")
 ∀ att:attribut ∈ ent.ses_attributs ;
   SWITCH(TYPEOF(att.son_type)) {
            case "String_Type"   : Affiche(att.son_nom+"   VARCHAR,");
            case "Integer_Type" : Affiche(att.son_nom+"  INTEGER,");
            case "Entity_Type"   : Affiche(att.son_nom+"   INT32 FOREIGN KEY (OID,"+ att.son_type.son_nom+),");
            case ...
      }
}
...
```

(b) Exemple de programme exploitant les méta-schéma EXPRESS pour la génération de schéma de base de données

```
Exemple de résultat :
CREATE TABLE object_property (
  OID INT32,
  target_class INT32
FOREIGN KEY (OID,class),
...
```

(c) Exemple de résultat sur l'entité object_property de la figure 1b

FIG. 4.5 – Générateur de structure de tables : architecture générale

1.2.2 Génération de la structure des tables

L'algorithme 14 présente les grandes lignes du programme de génération des requêtes SQL de création du schéma logique. Ce schéma parcourt chacune des entités du modèle EXPRESS donné en paramètre.

La figure 4.5a présente l'architecture du composant générateur de la structure des tables. Celui-ci reçoit, en paramètre, le modèle d'ontologie PLIB. Ce modèle est alors compilé et traduit

Algorithme 14 Génération des requêtes SQL de création du schéma logique à partir d'un modèle EXPRESS

Pour chaque entité du modèle EXPRESS,
on génère la requête SQL de création de la table correspondante ;
on génère les requêtes de création des tables d'aiguillage des attributs de type entité et de type SELECT ;
on génère les requêtes de création des tables des valeurs d'agrégats ;
on génère les requêtes de création des tables des types construits ;
on génère toutes les contraintes d'intégrités (y compris les PRIMARY KEY et FOREIGN KEY) ;
on génère les fonctions plpgsql des attributs dérivés ;
on génère la requête de création de la vue de l'entité EXPRESS (qui comprend aussi les attributs dérivés et inverses) ;

sous forme d'instances d'un méta-modèle exploitable par un programme générique. Ce dernier se base sur l'ensemble des règles de correspondances que nous avons spécifiées, pour générer les requêtes SQL. La figure 4.5b illustre la logique du programme exploitant les instances du méta-modèle EXPRESS pour générer les requêtes de création de tables. La figure 4.5c donne un exemple simplifié de code produit.

Soulignons qu'une telle approche était indispensable pour traiter le modèle EXPRESS PLIB. En effet, celui-ci est constitué de 218 entités, 5 types énumérés, 40 types SELECT, 61 types nommés autres que des types énumérés et types SELECT. Le modèle relationnel-objet résultant après l'application de nos règles de transformation (voir l'annexe C) est composée de 828 tables et vues dont :

- 218 tables d'entités,
- 48 tables de valeurs d'agrégats,
- 66 tables de types nommés,
- 218 vues d'entités,
- le reste étant les tables d'aiguillages chargées de permettre le polymorphisme et les unions de types.

Il est clair qu'un modèle d'une telle complexité ne pouvait être traité autrement que par des techniques génériques d'IDM.

Dans la section suivante, nous présentons le composant permettant l'importation d'ontologies.

1.3 Importation des ontologies dans la base de données

Cette section décrit le composant permettant le peuplement de la partie *ontologie* de notre architecture, dont nous venons de générer le schéma logique à partir d'ontologies représentées dans des fichiers d'échanges sous un certain format. Rappelons que pour la représentation des ontologies, les objectifs que nous nous sommes fixés pour la définition de l'architecture *OntoDB*

était (1) de rendre le système le plus possible générique par rapport au modèle d'ontologie utilisée, et (2) de pouvoir versionner les concepts (classes et propriétés) des ontologies dans la base de données.

1.3.1 Importation des données de façon générique

Pour atteindre le premier objectif, il est indispensable que le composant d'importation soit programmé de sorte à être indépendant du modèle d'ontologie utilisé dans la base de données, et qu'il ne dépende que des règles génériques définies pour représenter un schéma EXPRESS en PostgreSQL.

Le respect des exigences précédentes est particulièrement important car, si le modèle d'ontologie arrive à être modifié (ce qui était le cas tout au long de notre développement à travers le processus de normalisation du modèle PLIB), les programmes de lecture ne devraient pas nécessiter de modification.

En nous basant toujours sur les techniques de l'Ingénierie de Dirigée par les Modèles en EXPRESS, nous arrivons à répondre à ces deux exigences. En effet, l'environnement d'IDM EXPRESS, en plus de traduire les entités des modèles EXPRESS sous forme d'instances d'un méta-modèle, offre également des primitives pour *la manipulation des instances des entités de ces modèles de façon totalement générique*. Ces primitives permettent en particulier de :

- créer de nouvelles instances d'une entité (dont le modèle est représenté comme une instance d'un méta-modèle),
- supprimer une instance,
- récupérer toute la population d'instance d'une entité,
- accéder à la valeur d'un attribut d'une instance,
- modifier la valeur d'un attribut d'une instance.

Ces primitives peuvent être utilisées pour accéder de façon générique à une population d'instances d'entités EXPRESS qui représentent une ontologie particulière et pour générer des requêtes SQL pour le peuplement de la base de données.

Le composant d'importation (cf. figure 4.6) que nous avons implémenté, reçoit en paramètre le modèle d'ontologie et une population d'instances des entités de ce modèle (par exemple, PLIB). En se basant sur les règles de correspondances définies (cf. annexe C), il accède, au moyen des primitives de l'environnement d'IDM EXPRESS, aux instances des entités du modèle d'ontologie qui représentent l'ontologie. Il génère alors les requêtes SQL permettant le peuplement de la partie *ontologie* de la base de données.

Lors de l'importation des ontologies, il est également nécessaire d'assurer leur versionnement. C'est ce dont nous discutons dans la section suivante.

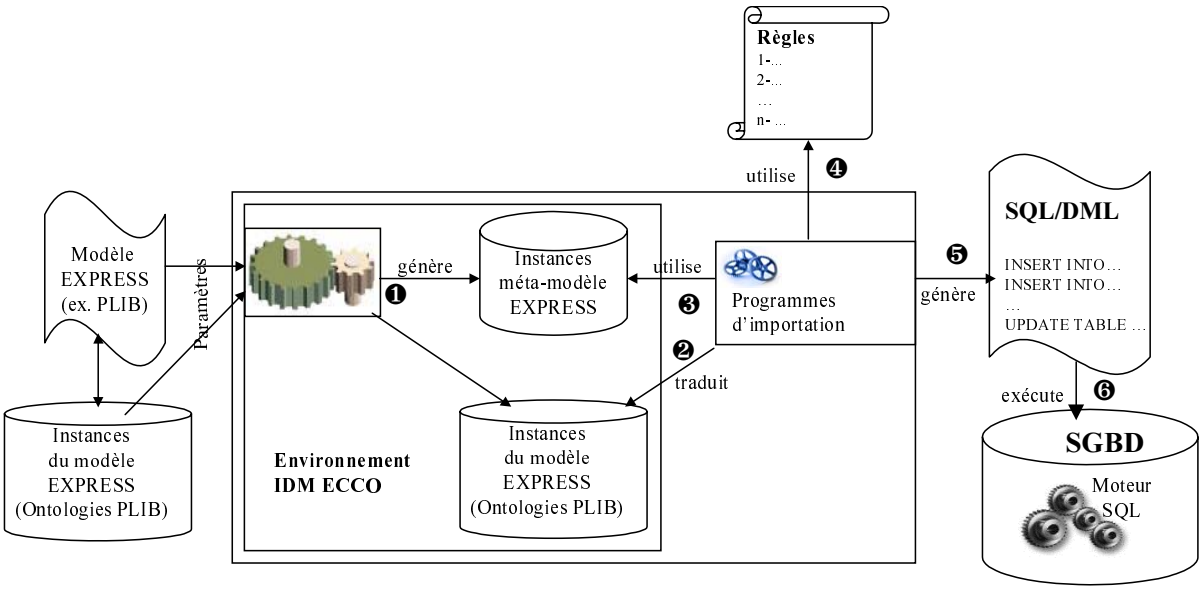


FIG. 4.6 – Importation des ontologies dans la base de données

1.3.2 Gestion du versionnement des concepts des ontologies

La gestion des versions de concepts d'une ontologie dans la base de données est le deuxième objectif que nous nous sommes fixées pour notre BDBO. Cette gestion vise à permettre l'archivage et la récupération des différentes versions de chaque concept à partir de la base de données. Dans la section 2.4.2 du chapitre 3, nous avons proposé une approche se basant sur le principe de *version courante*. Ici, nous montrons comment ce principe peut être implémenté dans le contexte particulier des ontologies PLIB et de PostgreSQL.

Rappelons d'abord qu'en PLIB, un concept est constitué de son identifiant (BSU) et de sa définition (DEF) (i.e. un concept = BSU+DEF). Le *BSU* (*Basic_ Semantic_ Unit*) (cf. chapitre 2 section 2) est composé d'un *code* et d'un numéro de *version*. Toutes les définitions (DEF) des concepts des ontologies référencent un BSU qui définit donc également la version de la définition. Cette composition nécessite donc de versionner (i.e., gérer plusieurs versions pour) à la fois les BSUs et les DEFs dans la base de données. Nous aurons donc des versions courantes de BSUs et des versions courantes des définitions des concepts. Rappelons enfin pour finir que, dans une ontologie PLIB, toutes les relations/associations entre des concepts sont réalisées en référençant des BSUs. Par exemple, lorsqu'une classe C_1 est super-classe d'une classe C_2, alors la définition de la classe C_2 référencera le BSU de la classe C_1 comme super-classe.

L'approche de versionnement que avons proposée à la section 2.4.2 du chapitre 3 est basée sur le concept de *version courante*. La politique de cette approche se présente comme suit :

– Le schéma de la partie *ontologie* se divise deux parties (logiques) :
  1. la partie *historique* qui enregistre les différentes versions des concepts (BSU et DEF),
  2. la partie *courante* qui consolide les dernières versions des différents concepts (BSU et

DEF).

- un concept courant duplique la plus grande version d'un concept sauf que toutes les références qu'il contient vers d'autres concepts sont modifiées pour référencer la version courante (i.e., la plus grande version) de ces concepts,

- toute nouvelle version d'un concept est automatiquement insérée dans la partie *historique* et, de plus, le concept en version courante est soit créé s'il n'existe pas, soit mis à jour avec les informations de la nouvelle version du concept, s'il existait déjà.

La mise en œuvre de cette politique est effectuée comme suit.
Pour identifier une version *courante* d'un BSU, nous avons ajouté une colonne booléenne *is_ version_ current* à la table *Basic_ Semantic_ Unit* qui indiquera si le BSU d'un concept en constitue la version courante ou non.

Soit alors **C**, un concept à introduire dans la BDBO et **n** son numéro de version. L'algorithme 15 définit la procédure de compilation du concept **C** dans la base de données.

Remarque
Notre mécanisme de *version courante* permet, dans tous les cas, de conserver une ontologie *courante* cohérente consolidant toutes les relations entre concepts même dans le cas de mise à jour asynchrone. Ceci est également vrai lorsqu'il n'est pas nécessaire de conserver l'historique des ontologies et que l'administrateur demande au système de ne pas créer de partie *historique*.

1.4 Extraction d'ontologies d'une BDBO

Il s'agit ici de décrire les programmes qui permettent l'extraction d'un sous-ensemble des concepts d'une ou plusieurs ontologies, avec éventuellement leurs instances pour les exporter dans des fichiers physiques sous certains formats. Dans notre prototype, il s'agira du format d'instances EXPRESS.

Dans la section 2.5 du chapitre précédent, nous avons distingué deux types d'extraction des objets de la base de données selon les besoins des utilisateurs. Le premier vise à assurer une *complétude syntaxique* des objets extraits et le deuxième assure une *complétude sémantique*. L'objectif de la *complétude syntaxique* vise à assurer la cohérence syntaxique de la population des objets extraits de la base de données, c'est-à-dire l'absence de pointeurs incohérents (intégrité référentielle). L'idée ici est d'extraire pour chaque objet, tous les objets qu'il référence. Celui de la *complétude sémantique* vise à extraire un sous-ensemble d'objets *pertinents* pour les utilisateurs (ou applications) qui seront amenés à manipuler ces objets pour leur permettre l'interprétation des concepts extraits. L'idée ici est d'extraire de la base de données un sous-ensemble *minimal* d'objets permettant l'interprétation (ou la manipulation) autonome des concepts extraits par une application donnée (ceci ne pouvant être défini dans l'absolu, le système permet à l'utilisateur de préciser ses besoins sémantiques).

Algorithme 15 Algorithme de versionnement de concepts PLIB.

IF aucune version du concept **C** n'existe pas dans la base de données **THEN**
  Créer le BSU du concept **C** dans la base de données. On initialise sa version à **n** dans la base de données et son attribut **is_version_current** à **FALSE** ;
  Créer le BSU du concept en version courante (**is_version_current**=TRUE)
  Créer la définition du concept en version n (i.e. référençant le BSU en version **n**)
  Créer la définition en version courante, i.e. référençant le BSU en version courante, tous les attributs de type BSU ou agrégats de BSU sont modifiés pour référencer des versions courantes.
ELSIF la version de la définition du concept **C** est supérieure à celle existante dans la BD **THEN**
  Créer le BSU du concept en version **n** (**is_version_current**=FALSE) ;
  Ré-initialiser le BSU du concept en version courante de la base de données (qui correspond à une version $\leq n$) avec le BSU du concept **C** ;
  Créer la définition du concept en version **n** ;
  Supprimer la définition du concept en version courante dans la base de données ;
  Créer la définition du concept en version courante (tous les attributs de type BSU ou agrégats de BSU sont modifiés pour référencer des versions courantes.
ELSIF la version de la définition du concept est inférieure à la version du concept dans la BD **THEN**
  Créer le BSU du concept en version **n** (**is_version_current**=FALSE) s'il n'existe pas dans la base de données ;
  Créer la définition version **n** ;
ELSIF la version de la définition du concept **C** est égale à la version du concept dans la BD **THEN**
  IF la révision de la définition du concept **C** est supérieure à celle de la BD **THEN**
    Supprimer la définition du concept en version courante,
    Supprimer la définition du concept en version **n**,
    Créer la définition du concept en version courante à partir de la nouvelle définition,
    Créer la définition du concept en version **n** à partir de la nouvelle définition,
  END IF
END IF

Par exemple, lorsqu'on souhaite extraire une classe C_i,

- dans le cadre d'une *complétude syntaxique*, tous objets référencés par C_i seront extraits,
- dans le cadre d'une *complétude sémantique*, on pourrait vouloir récupérer en même temps la définition de sa superclasse et de celles de ses propriétés applicables mais on pourrait également ne pas vouloir extraire les objets provenant des attributs *icon*, *figure*, *simplified_ drawing* qui sont des références à des fichiers physiques externes et rendent donc plus complexe à manipuler le résultat de l'extraction (cf. section 3.2.2 dans l'annexe A).

Notons que, dans le cas particulier de PLIB, les concepts ne référencent d'autres concepts uniquement qu'au moyen de leur identifiant universel (BSU). Pour accéder à la définition d'un concept à partir de son BSU, il faut calculer l'inverse de la relation qui lie sa définition à son BSU. Les définitions ne sont donc pas extraites si l'on s'en tient à la seule complétude syntaxique.

Dans les deux sous-sections suivantes, nous présentons les algorithmes des fonctions d'extraction dans le cas d'une complétude syntaxique et une complétude sémantique.

1.4.1 Algorithme d'extraction pour complétude syntaxique

L'extraction d'un objet exige de parcourir tous ses attributs pour récupérer les objets qu'il référence et pour chacun des ces objets d'extraire également de façon récursive tous les objets qu'il référence. L'ensemble des objets ainsi récupérés constitue ce que nous appellerons *la grappe* de l'objet de départ. Lorsque tous les attributs des objets de la grappe sont parcourus, alors, la grappe est *syntaxiquement* correcte (complétude syntaxique).

Pour la récupération des objets de la grappe à partir des valeurs des attributs qui référencent des objets, deux solutions sont possibles :

- (1) on code en dur dans les programmes les appels permettant d'accéder aux différents attributs de chaque entité,
- (2) on utilise la représentation du modèle d'ontologie dans la partie *méta-schéma* de notre architecture, pour parcourir automatiquement les attributs de chaque entité.

Les exigences de programmation de l'extraction des objets, étant les mêmes que dans la section précédente à savoir son indépendance vis à vis du modèle d'ontologie utilisé (pour pouvoir supporter tout changement et/ou modification du modèle d'ontologie et du schéma logique des tables de la partie *ontologie*), nous avons opté pour la deuxième approche.

Le principe de l'algorithme pour une complétude syntaxique est présenté dans l'algorithme 16. Celui-ci consiste (1) à partir de l'instance d'entité représentant l'objet à extraire de la base de données, (2) puis à parcourir tous les attributs de l'entité. (3) Si l'attribut est de type association ou agrégation, à extraire récursivement le ou les objets référencées par l'attribut.

Algorithme 16 Algorithme d'extraction syntaxique des concepts de la base de données.

```
FUNCTION export(O : OBJECT) → SET OF OBJECT
  ent :entité ;
  result := {} ;
  //On récupère l'entité de l'objet O à partir méta-modèle.
  ent :=get_type_Of(O) ;
  result := result ∪ {O} ;
  // On parcourt tous les attributs de l'entité de l'objet.
  ∀ att_i ∈ ent.ses_attributs ;
    // Si le type de l'attribut est un type entité (association) ou un type agrégat
    SI(1) att_i.son_type = entité OU att_i.son_type = aggregate_type ALORS
        //On ajoute à la variable result sur les objets référencés de l'objet O
              par l'attribut att.
        result := result ∪ get_value_attribute(O, ent, att) ;
        // Appel recursif de la fonction export par les objets référencés par O.
        ∀ O_i ∈get_value_attribute(O,ent,att) ;
          result := result ∪ export(O_i) ;
  FINSI(1)
  RETURN result ;
END FUNCTION ;
```

1.4.2 Algorithme d'extraction pour une complétude sémantique

Avant la mise en œuvre de l'algorithme d'extraction pour une complétude sémantique, il est d'abord nécessaire d'identifier les attributs des entités qui feront objets d'un traitement particulier. Nous avons identifié trois traitements différents :

- $traitement_1$: on n'extrait pas l'objet référencé par l'attribut (le pointeur correspondant à l'attribut est mise à NULL dans le fichier généré),
- $traitement_2$: on extrait l'objet et sa grappe référencé par l'attribut,
- $traitement_3$: on extrait l'objet et sa grappe référencé par l'attribut ainsi que le ou les objets d'un type qui le référence par un attribut donné.

Le principe général de l'algorithme d'extraction pour une complétude sémantique est le même que l'algorithme précédent à la différence que des tests sont effectués sur les attributs identifiés pour faire l'objet d'un traitement particulier. L'algorithme est présenté dans l'algorithme 17. Nous désignerons par $Trait_1$, l'ensemble des couples (ent_i,att_i) dont la valeur de l'attribut att_i ne doit pas être extraite d'une base de données ($traitement_1$), par $Trait_2$, l'ensemble des couples (ent_i,att_i) dont la valeur de l'attribut att_i doit être être extraite avec sa grappe de la BDBO ($traitement_2$) et par $Trait_3$, l'ensemble des quadruplets (ent_i,att_i,ent_j,att_j) dont la valeur de l'attribut att_i doit être extraite ainsi que les objets de la BDBO de type ent_j qui référencent sa valeur par l'attribut att_j ($traitement_3$). L'algorithme 17 présente une ébauche du programme mettant œuvre le principe d'extraction des concepts pour une complétude sémantique.

Exemple :

- Si on ne souhaite pas extraire l'image graphique des classes (représentée par l'attribut *figure* dans le modèle d'ontologie PLIB), alors on aura $Trait_1 = \{(class,figure)\}$.

– Si souhaite extraire la description textuelle des classes (représentée par l'attribut *names* dans le modèle d'ontologie PLIB), alors on aura $Trait_2 = \{(class,names)\}$.
– Si on souhaite extraire la définition des *Property_ BSU* de l'attribut *described_ by* de l'entité *class* de PLIB, on aura $Trait_3$={(*class*,*described_ by*,*property*,*identified_ by*)}.

1.4.3 Contexte d'exécution

La figure 4.7, présente l'architecture du programme d'extraction des instances. Le programme d'extraction utilise (1) les instances du méta-modèle pour accéder au modèle des entités à extraire et (2) l'interface de programmation à liaison différée (cf. chapitre 3 section 2) que nous avons implémentée pour l'accès aux ontologies ainsi qu'aux données à base ontologique, (3) la table de traitement particulier des attributs, et (4) dans le cadre d'une extraction sémantique, les ensembles $Trait_1$, $Trait_2$ et $Trait_3$ des attributs qui feront l'objet d'un traitement particulier.

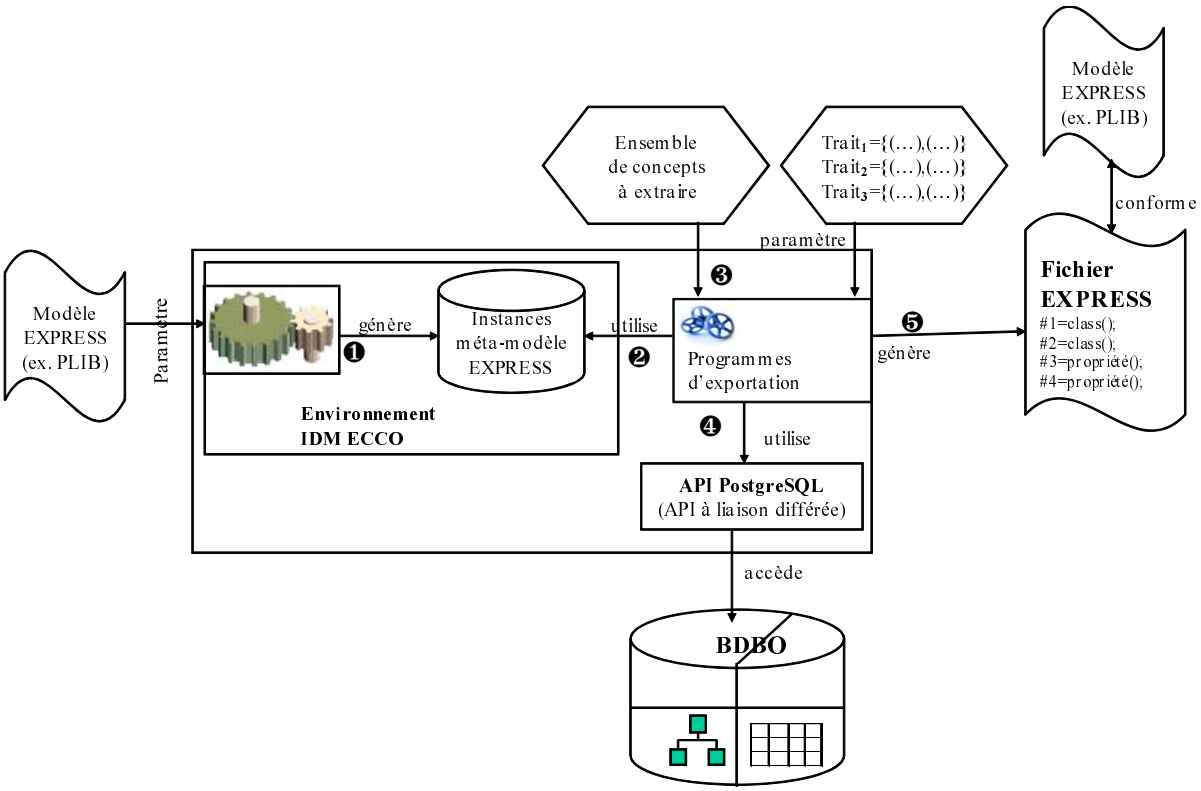


FIG. 4.7 – Extraction de concepts des ontologies stockées dans la BDBO

2 Représentation du méta-modèle d'ontologie

Cette section décrit la mise en œuvre des différents composants nécessaires pour la représentation dans la partie *méta-schéma* de notre architecture *OntoDB* du modèle d'ontologie et du méta-modèle du modèle d'ontologie. Notre prototype se basant sur le modèle d'ontologie PLIB définit dans le langage EXPRESS, il s'agira pour nous de le représenter dans la base de données

Algorithme 17 Algorithme d'extraction sémantique des concepts de la base de données.

```
FUNCTION export(O : OBJECT ;
                Trait1 :SET OF (entité × attribut)
                Trait2 :SET OF (entité × attribut)
                Trait3 :SET OF (entité × attribut × entité × attribut)
                                                    ) → SET OF OBJECT
DEBUT   ent_i :entité ;
  result := {} ;
  //On récupère l'entité de l'objet O à partir méta-modèle.
  ent_i :=get_type_Of(O) ;
  result := result ∪ {O} ;
// On parcourt tous les attributs de l'entité de l'objet.
  ∀ att_i ∈ ent_i.ses_attributs ;
    // Si le type de l'attribut est un type entité (association) ou un type agrégat
    IF att_i.son_type = entité OR att_i.son_type = aggregate_type THEN
       // Cas TRAITEMENT 1 :
       // Si on ne veut pas extraire les objets référencés par l'attribut att_i
       IF (ent_i,att_i) ∈ Trait1 THEN
         result := result ∪ get_value_attribute(O, ent_i, att_i) ;
         O.attt_i := NULL ;          DoNothing() ;
       END IF
       // Cas TRAITEMENT 2 :
       // Si on veut extraire la grappe de l'objet référencé par l'attribut att_i
       IF (ent_i, att_i) ∈ Trait2 THEN
            //On ajoute à la variable result sur les objets référencés par l'objet O
            au moyen de l'attribut att.
             result := result ∪ get_value_attribute(O, ent_i, att_i) ;
             // Appel recursif de la fonction export sur les objets référencés par O.
             ∀ O_i ∈get_value_attribute(O,ent_i,att_i) ;
             result := result ∪ export(O_i) ;
       END IF
         // Cas TRAITEMENT 3 :
       // Si on veut extraire la grappe de ou des objets référencés par l'attribut att_i
        ainsi que les objets qui les référencent par un attribut att_3 d'une entité ent_3
          ∀(ent_0, att_0, ent_3, att_3) ∈ Trait3 ;
          IF ent_i = ent_0 AND att_i=att_0 THEN
            O_att=get_value_attribute(O, ent_i, att_i) ;
            ∀ O_i ∈ used_in(O_att, ent_3, att_3) ;
            result := result ∪ export(O_i, Trait1, Trait2, Trait3) ;
            result := result ∪ get_value_attribute(O, ent_i, att_i) ;
            // Appel recursif de la fonction export sur les objets référencés par O.
            ∀ O_i ∈get_value_attribute(O,ent_i,att_i) ;
            result := result ∪ export(O_i, Trait1, Trait2, Trait3) ;
           //On ajoute à la variable result les objets référencés par l'objet O
              au moyen de l'attribut att_i.
             result := result ∪ get_value_attribute(O, ent_i, att_i) ;
         END IF
    END IF
  RETURN result ;
END FUNCTION ;
```

comme instance d'un méta-modèle EXPRESS définit dans le langage EXPRESS. Comme pour la représentation de l'ontologie dans la base de données, la mise en œuvre de la partie *méta-schéma*, nécessite la définition de composants permettant :

- **la définition de la structure des tables** dans lesquelles les modèles EXPRESS seront stockés ;
- **la lecture des modèles d'ontologies** pour leur insertion dans la base de données ;
- **la lecture d'un méta-modèle EXPRESS** pour son insertion dans la base de données.

2.1 Définition de la structure des tables

Le premier composant consiste à définir la structure de tables dans laquelle seront représentés le ou les modèles d'ontologies et le méta-modèle des modèles d'ontologies utilisés. Les modèles d'ontologies auxquels nous nous intéressons s'expriment tous sous forme d'instances d'un méta-modèle. Représenter les modèles d'ontologies consiste donc à représenter un méta-modèle réflexif de l'ensemble des modèles d'ontologies directement dans la base de données sous forme d'une structure de tables et ensuite à peupler cette structure de tables avec les modèles d'ontologie et le méta-modèle lui-même.

Dans le cadre de notre implémentation, le modèle d'ontologie PLIB que nous utilisons étant défini en EXPRESS, il existe dans la littérature des méta-modèles du langage EXPRESS exprimés en eux-mêmes dans le langage EXPRESS. Nous avons en particulier les méta-modèles EXPRESS des environnements d'IDM ECCO [147] et SDAI [148]. Ceux-ci peuvent être utilisés comme modèle source pour la définition de la structure des tables de la partie *méta-schéma* de notre architecture. Nous avons choisi d'utiliser le méta-modèle défini dans l'environnement ECCO (légèrement modifié) car il est beaucoup plus compacte et efficace que le méta-modèle SDAI (Standard Data Access Interface [131]). En effet, le méta-modèle de l'environnement ECCO comporte 0 type SELECT, 20 types d'entités alors que celui de SDAI comporte 10 type SELECT, 84 types d'entités. Ce méta-modèle n'étant pas disponible sous une forme EXPRESS éditable, nous en avons d'abord créée une forme éditable.

Le méta-modèle EXPRESS étant modélisé dans le langage EXPRESS sous une forme classique, il peut alors être traduit en relationnel de la même façon que nous avons traduit le modèle EXPRESS du modèle d'ontologie PLIB en relationnel pour la représentation des ontologies (cf annexe C et section 1.2). Compte tenu du fait que le composant permettant la *création de la structure de table* de la partie *ontologie* est générique pour tout modèle EXPRESS, celui-ci peut être une fois de plus utilisé pour la définition de la structure des tables du *méta-schéma* (cf. figure 4.8). Le méta-modèle EXPRESS sera cette fois-ci fournit comme paramètre au programme du générateur qui générera les requêtes SQL pour la création de la structure des tables de la partie *méta-schéma*.

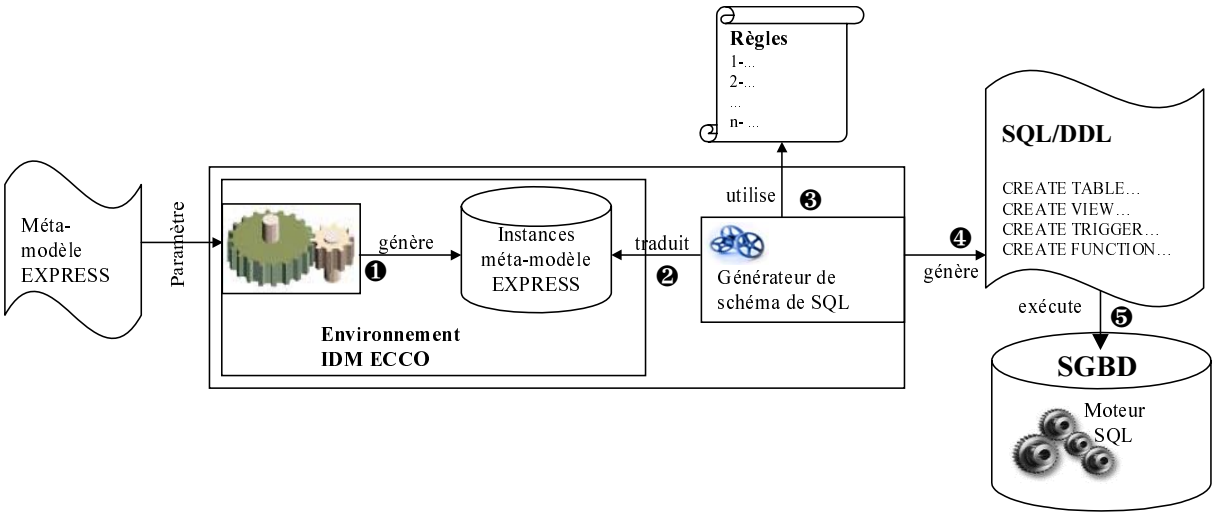


FIG. 4.8 – Génération de la structure des tables (schéma DDL) de la partie *méta-schéma*

2.2 Peuplement de la partie *méta-schéma*

Ce composant doit permettre à partir d'un modèle EXPRESS, de le représenter par un ensemble d'instances dans la partie *méta-schéma*. Ce composant permettra donc de lire successivement le modèle EXPRESS du modèle d'ontologie PLIB et le méta-modèle du langage EXPRESS en EXPRESS (réflexivité) pour les insérer dans la base de données. La réalisation de cette tâche peut être faite en *réutilisant le composant d'importation d'ontologies dans la base de données* définit dans la section 1.3 pour la lecture des ontologies. Ce composant a, en effet, été défini de sorte à être générique, c'est-à-dire indépendant de tout modèle EXPRESS. Il génère alors des requêtes d'insertion dans la base de données pour toute population d'instances du modèle EXPRESS en paramètre.

L'utilisation de ce composant exige que nous lui fournissions d'une part, le méta-modèle d'EXPRESS et, d'autre part, un ensemble d'instances de ce méta-modèle représentant respectivement le modèle d'ontologie PLIB et le méta-modèle EXPRESS lui-même. Pour pouvoir utiliser ce composant, il nous faut donc être capable d'exprimer ces deux modèles sous-formes d'instances du méta-modèle EXPRESS qui a permis de définir la structure des tables. Cette opération est possible grâce à l'environnement d'IDM EXPRESS qui offre la possibilité de compiler (cf. section 4.1) tout modèle EXPRESS et de l'exprimer sous forme d'instances du méta-modèle. Ces instances du méta-modèle EXPRESS générées par l'environnement d'IDM peuvent alors être fournies en paramètre au programme d'*importation d'instances*. La figure 4.9, présente l'architecture du composant de peuplement de la partie *méta-schéma* se basant sur le programme d'importation d'instances présenté à la figure pour générer la représentation du modèle d'ontologie et la représentation du méta-modèle EXPRESS dans lui-même.

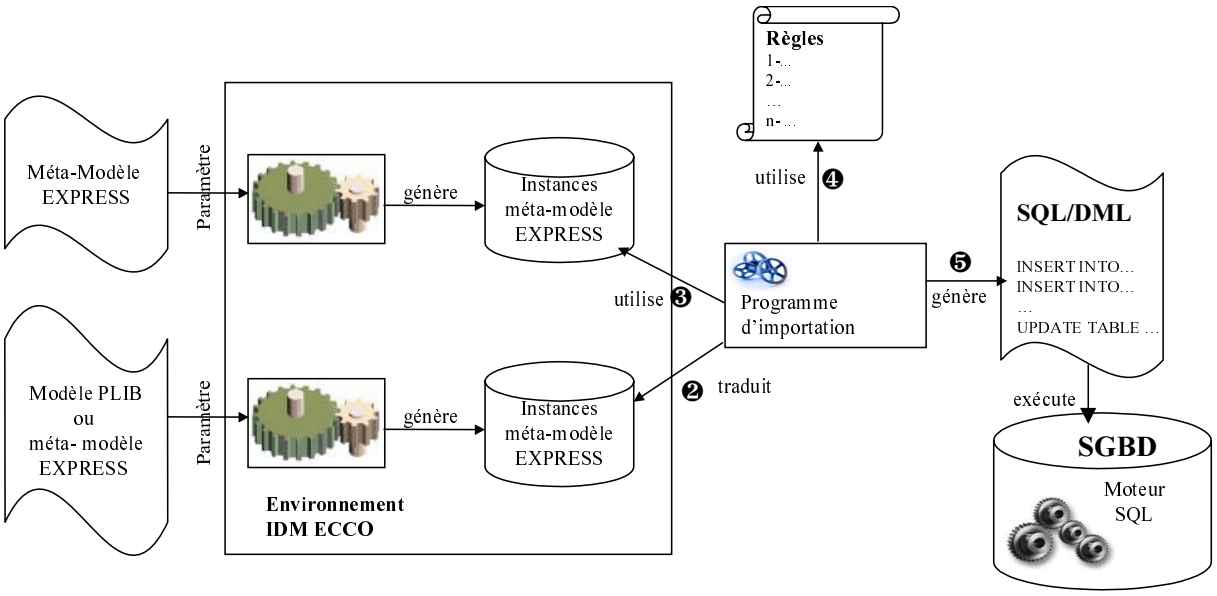


FIG. 4.9 – Peuplement de la partie *méta-schéma* de l'architecture OntoDB par le modèle d'ontologie et le méta-modèle.

3 Représentation des données à base ontologique dans la partie *données*

Nous discutons dans cette section la représentation des données à base ontologique qui sont des instances des classes des ontologies, dans la base de données. Le déploiement de la partie *données* exige de réaliser quatre tâches :

- Définir des règles de correspondances entre les mécanismes existants dans le modèle d'ontologies et les mécanismes du modèle relationnel en vue de générer la structure, c'est-à-dire le schéma relationnel de données, pour la représentation des instances.
- Représenter explicitement le schéma des instances de chaque classe.
- Gérer le versionnement et le cycle de vie des instances dans la base de données pour permettre l'archivage et la récupération des données à base ontologique.
- Permettre l'accès bilatère entre les concepts des ontologies et les données correspondantes qui sont représentées séparément dans l'architecture *OntoDB*. Ceci en vue de pouvoir accéder aux données via l'ontologie et réciproquement.

Dans la section 3.1, nous présentons les règles de correspondances que nous avons définies entre les concepts des ontologies et le modèle relationnel. Dans la section 3.2, nous présentons la solution que nous proposons pour la représentation explicite du modèle conceptuel des instances. La section 3.3 décrit la façon dont nous avons mis en œuvre la méthode de gestion du cycle de vie des instances proposée dans la section 3.2 du chapitre précédent. Enfin la section 3.4 décrit la façon dont nous avons réalisé les fonctions *Nomination* et *Abstraction* pour établir le lien entre ontologie et données.

3.1 Règles de correspondances entre les concepts des ontologies PLIB et le relationnel

Un modèle d'ontologie est un véritable formalisme objet qui, outre les mécanismes usuels (e.g., classe et héritage) définit toujours un certain nombre de mécanismes spécifiques (e.g., *propriétés dépendantes du contexte en PLIB*, concepts *définis* en OWL).

Compte tenu du fait que (1) les classes des ontologies sont organisées en hiérarchies d'héritage, et (2) les propriétés des classes peuvent avoir pour co-domaine un grand nombre de types n'ayant pas tous des équivalents dans l'univers relationnel, alors la définition de mécanismes de correspondances (mapping) pour leur représentation en relationnel s'impose. Le modèle d'ontologie utilisé dans notre cas étant PLIB, notre mapping consistera donc à traduire les principaux concepts de PLIB en relationnel.

Nous présentons dans cette section les règles de correspondances que nous avons définies entre les concepts du modèle d'ontologie PLIB et le modèle relationnel. Dans les sous-sections 3.1.1 et 3.1.2, nous discutons respectivement de la représentation relationnelle des classes et des types PLIB. La section 3.1.3 décrit la représentation des propriétés.

3.1.1 Connaissance structurelle : les classes

Une grande différence entre une ontologie et le modèle conceptuel est que, si ce dernier *prescrit* les propriétés qui *doivent* être utilisées pour décrire les instances de chaque classe, une ontologie ne fait que *décrire* les propriétés qui *peuvent* être utilisées. Si l'on fait l'hypothèse, ce qui est le cas général, que seules certaines classes et propriétés seront valuées, on ne peut donc définir directement une structure relationnelle de représentation des données à base ontologique sans savoir quelles classes et propriétés seront effectivement valuées au moins par une instance. Cet ensemble de classes et de propriétés, qui est un sous-ensemble (éventuellement maximum) de l'ontologie, constitue le *modèle conceptuel* des données à base ontologique. Quelque soit la façon dont ce modèle conceptuel est défini (e.g., choix de l'administrateur, construction dynamique lorsque des ensembles d'instances sont introduits), le problème qui se pose à nous est de définir comment un modèle conceptuel donné doit être représenté. Soulignons que le sous-ensemble extrait *n'a aucune raison de respecter les règles de l'héritage*. Certaines propriétés héritées en tant qu'applicables par plusieurs sous-classes d'une même super-classe pourront ainsi être valuées dans certaines de ces sous-classes, et non valuées dans d'autres. Ceci interdit, pratiquement, d'utiliser la représentation verticale de l'héritage (cf. section 3.4.1.1 du chapitre 1)

Ainsi, le modèle conceptuel est construit par une *sélection*, selon le besoin de l'application à laquelle elle est destinée, d'un quelconque sous-ensemble des classes et des propriétés de l'ontologie représentée dans la BDBO.

Exemple

La figure 4.10 montre un exemple de la définition d'un modèle conceptuel à partir d'une ontologie composée de six classes, chacune définissant un certain nombre de propriétés applicables : C[a_1, a_2, a_3], C1[a1_1, a1_2], C2[a2_1, a2_2], C11[a11_1, a11_2, a11_3], C21[a21_1], C22[a22_1, a22_2]. On peut remarquer sur la figure que le modèle conceptuel est composé *uniquement* de quatre classes (C2, C11, C21 et C22) possédant respectivement les propriétés suivantes : C2[a_3, a2_2] et C11[a_1, a_2, a11_1], C21[a_1, a1_2, a22_1], C22[a_1, a1_3, a22_1]. La propriété *a_1* héritée en tant qu'applicable par toutes les classes n'est effectivement utilisée que dans C11, C21 et C22. La propriété a2_2 utilisée pour les instances de C2 n'est plus utilisée pour les instances de sa sous-classe C21.

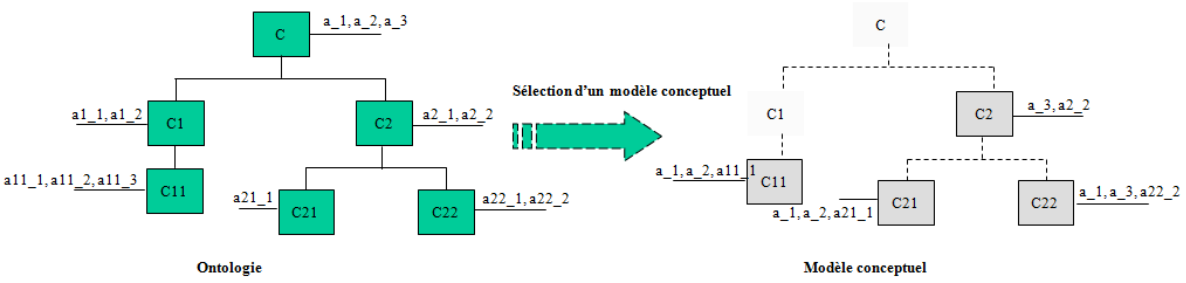


FIG. 4.10 – Extraction d'un modèle conceptuel à partir d'une ontologie.

L'approche de représentation des instances que nous avons proposée dans le chapitre précédent (cf. section 3.1) pour l'architecture *OntoDB*, consiste à définir une table spécifique pour chaque classe de la base de données, où seules sont représentées les propriétés valuées par au moins une instance admettant cette classe comme classe de base. En appliquant cette approche au modèle conceptuel de la figure 4.10, on obtient les tables de la figure 4.11.

C11			
OID	a_1	a_2	a11_1
1	-	-	-
2	-	-	-

C2		
OID	a_3	a2_2
3	-	-
4	-	-

C21				
OID	a_1	a_2	a2_1	a21_1
5	-	-	-	-
6	-	-	-	-

C22			
OID	a_1	a_3	a22_2
7	-	-	-
8	-	-	-

FIG. 4.11 – Approche de représentation horizontale.

3.1.2 Connaissance structurelle : les types

Dans cette section, nous donnons les règles des correspondances en PostgreSQL des types de valeurs utilisés PLIB. Dans PLIB, les types sont repartis en quatre groupes :

1. les types simples,
2. les types complexes,
3. les types nommées,
4. les types collections.

3.1.2.1 Les types simples

La table suivante, présente les correspondances entre les types simples de PLIB et les types du modèle relationnel.

Type PLIB	**Type PostgreSQL**
int_type int_measure_type non_quantitative_int_type	INT8
real_type real_currency_type real_meseaure_type	FLOAT8
boolean_type	BOOLEAN
string_type non_quantitative_code_type	VARCHAR

3.1.2.2 Les types complexes

Les types complexes de PLIB comportent trois familles de type :

- le type *level_type*,
- le type *class_instance_type*,
- le type *entity_instance_type*.

3.1.2.2.1 level_type

Le *level_type* de PLIB permet de qualifier des valeurs d'un type quantitatif. Une valeur de *level_type* contient quatre valeurs caractérisées par les qualificatifs : min (minimal), nom (nominatif), typ (typical), max (maximal) (cf. section 3.2.1 du chapitre 2). Deux solutions sont envisageables pour la représentation de ce type en PostgreSQL.

1. une valeur de type *level_type* pourrait être représentée dans une colonne dont le domaine serait un tableau de quatre éléments INT8[4] ou FLOAT[4] suivant le type de base du type *level_type*. Ainsi, on pourrait affecter la valeur du *min* au première indice du tableau, *nom* la deuxième, *typ* la troisième et *max* la quatrième.
2. une autre solution consisterait à éclater dans des colonnes différentes chacune des valeurs du type *level_type* (min, nom, typ, max). Le nom de chaque colonne est suffixé par chacun des éléments du type *level_type*. On ne crée des colonnes que pour des qualificatifs du type *level_type* effectivement initialisés par au moins une instance de la classe.

Nous avons retenu la deuxième solution qui offre une plus grande facilité pour la manipulation des valeurs du type *level_type* par rapport à la première approche utilisant les tableaux PostgreSQL.

3.1.2.2.2 class_instance_type.

Le type *class_instance_type* est un type de PLIB qui exprime que le domaine de valeur d'une propriété est constitué par l'ensemble des instances d'une classe. Ce type représente l'association entre classes. Il est utilisé, en particulier, pour représenter la relation de compositions entre objets (Ex. vis-écrou). Un type *class_instance_type* dispose d'un seul attribut : *domain* (voir figure A.4 de l'annexe A). Cet attribut référence la *class_bsu* de la classe qui définit le domaine de valeurs du type.

Le problème qui se pose pour la représentation des associations entre objets en PLIB est la mise en œuvre du polymorphisme. En effet, lorsqu'une propriété a pour type *class_instance_type*, elle peut référencer des instances de n'importe quelles sous-classes de la classe déclarée dans l'ontologie. Ceci pose un problème essentiel dans la représentation que nous avons choisi pour les hiérarchies de l'héritage.

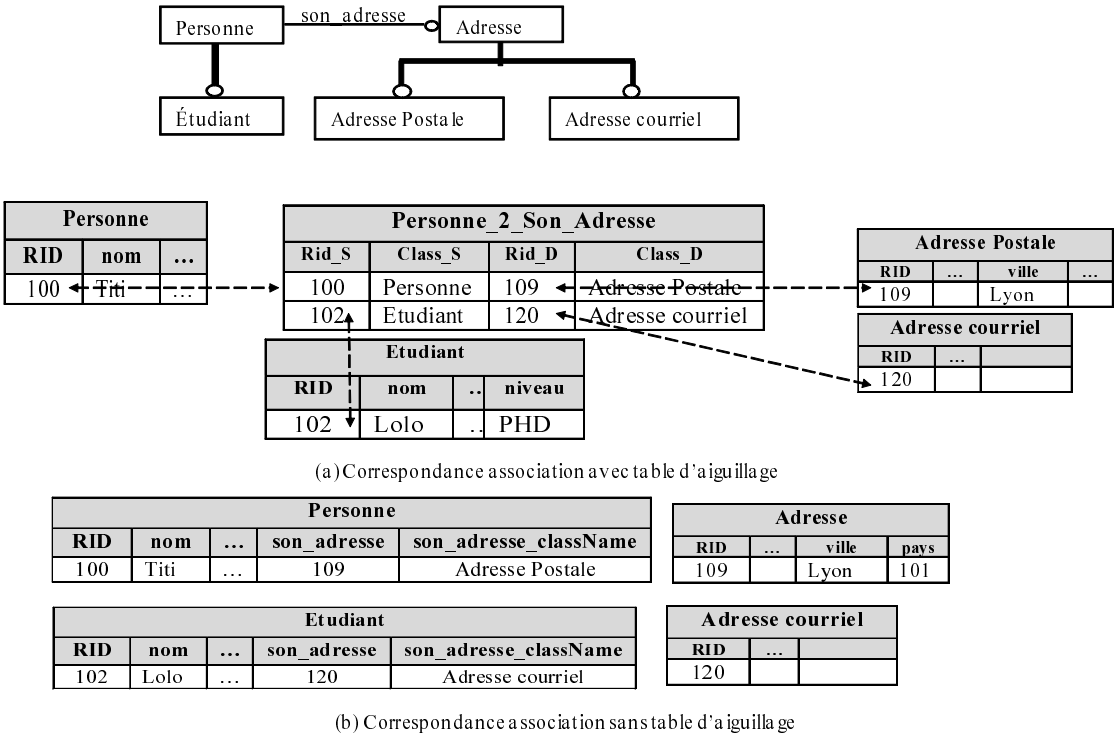


FIG. 4.12 – Les différentes approches de représentation des associations en relationnel.

Pour faire face au problème du polymorphisme plusieurs solutions sont envisageables :

- la première consisterait à représenter dans la colonne l'identifiant (rid) de l'instance cible, sans préciser la classe à laquelle appartient cette instance. Pour évaluer le lien, le système devrait alors interroger l'ontologie pour connaître la hiérarchie de classes, puis chercher la cible toutes les sous-classes pour savoir quelle est la classe de base de l'instance.
- Une deuxième solution consisterait à créer une table d'aiguillage analogue à celle utilisée dans la partie *ontologie* qui ferait le lien entre les deux tables des classes de composants

(cf. section 1.2.1). La colonne de la propriété sera une clé étrangère vers l'identifiant (*rid*) de la table d'aiguillage. La table d'aiguillage serait constituée de quatre colonnes comme sur la figure 4.12a.

1. la colonne *Rid_S* (Rid-Source) permet de contenir l'identifiant (rid) de l'instance de la classe source,
2. la colonne *Class_S* (Class-Source) permet comme le nom l'indique à mémoriser le nom de la classe (source) de l'association,
3. la colonne *Rid_D* (Rid-Destination) permet de contenir l'identifiant (rid) de l'instance de la classe référencée de l'association,
4. la colonne *Class_D* (Class-Destination) permet comme le nom l'indique à mémoriser le nom de la classe référencée de l'association.

– La troisième solution consisterait à associer deux colonnes à toute propriété de type *class_instance_type*. Une colonne pour l'identifiant (rid) de l'instance et la deuxième pour le nom de la table associée à la classe de l'instance. Le nom de la première colonne est de type INT8 et son nom est suffixé par la chaîne 'rid'. La deuxième colonne est de type VARCHAR et son nom est suffixé de la chaîne '*tablename*' (cf. figure 4.12b).

La première solution nécessiterait un nombre d'accès considérable à la base de données. Celle-ci doit donc être évitée pour des raisons de performances. La deuxième solution exige de faire une jointure pour retrouver l'identifiant de l'instance référencée par la valeur du type. La troisième permet de retrouver l'instance cible en un seul accès. Nous avons donc retenu la troisième approche lorsque la cardinalité est de type 0 : 1. Nous proposons de retenir la deuxième solution lorsque la cardinalité est $n : m$.

3.1.2.2.3 entity_instance_type

La problématique de la représentation du type *entity_instance_type* est similaire à celle du type *class_instance_type*. Tous deux permettent de stocker des références à des instances d'un modèle objet. Alors que pour une *class_instance_type*, la structure de la table (ou des tables) cible est définie pour l'ontologie, pour l'*entity_instance_type*, elle est définie par une entité EXPRESS, ou une hiérarchie d'entités EXPRESS. Nous proposons de représenter le type *entity_instance_type* de la même manière que le type *class_instance_type*. On crée donc deux colonnes l'une pour le *rid* de l'objet référencé et l'autre pour le nom de la table de l'entité si la cardinalité est 1. On passe par une table d'aiguillage si la cardinalité est supérieure à 1.

3.1.2.3 Les types nommés

Lorsque le type d'une propriété est de type *named_type* (cf. annexe A section 3.2.1), ses valeurs sont représentées en appliquant les règles de représentations que nous avons définies dans les sections précédentes au type de base de ce type nommé.

3.1.2.4 Les types collections

Pour la représentation des types collections (cf. annexe A section 3.2.1), nous distinguerons trois cas :

- les collections de type simple,
- les collections de type énuméré (*level_ type*),
- les collections de type complexe (*class_ instance_ type*, *entity_ instance_ type*).

Avant de décrire les correspondances pour chacun de ces types, notons que notre proposition se base uniquement que sur les collections à une seule dimension. Les collections à plusieurs niveaux de profondeur ne sont pas traitées et sont en perspectives. Compte tenu de la disponibilité du type collection dans PostgreSQL, nous avons opté pour l'utilisation du type tableau (ARRAY) de PostgreSQL, sauf pour les associations que l'on veut pouvoir parcourir dans les deux sens.

- **les collections de type simple.** Pour les collections (LIST, ARRAY, BAG, SET) de type de simple, nous proposons de les traduire en tableaux. Le type des éléments du tableau sera le type de base de la collection PLIB. Par exemple, les collections de type INTEGER sont traduits en **INT8[]**, et les collections de type STRING sont traduits en **VARCHAR[]**. Suivant la nature de la collection (LIST ou ARRAY ou BAG ou SET), nous associons à sa colonne une contrainte *CHECK* qui contrôlera les éléments du tableau (exemple pour éviter tous doublons dans les collections SET, etc.)
- **les collections de type *level_ type*.** Les types énumérés sont une collection de quatre valeurs. On peut donc voir les agrégats de type *level_ type* comme une collection de données à deux dimensions. Pour leur représentation, nous utilisons les tableaux multi-dimensionnels de PostgreSQL. Par exemple, un agrégat de type énuméré est traduite par **INT8[][4]**.

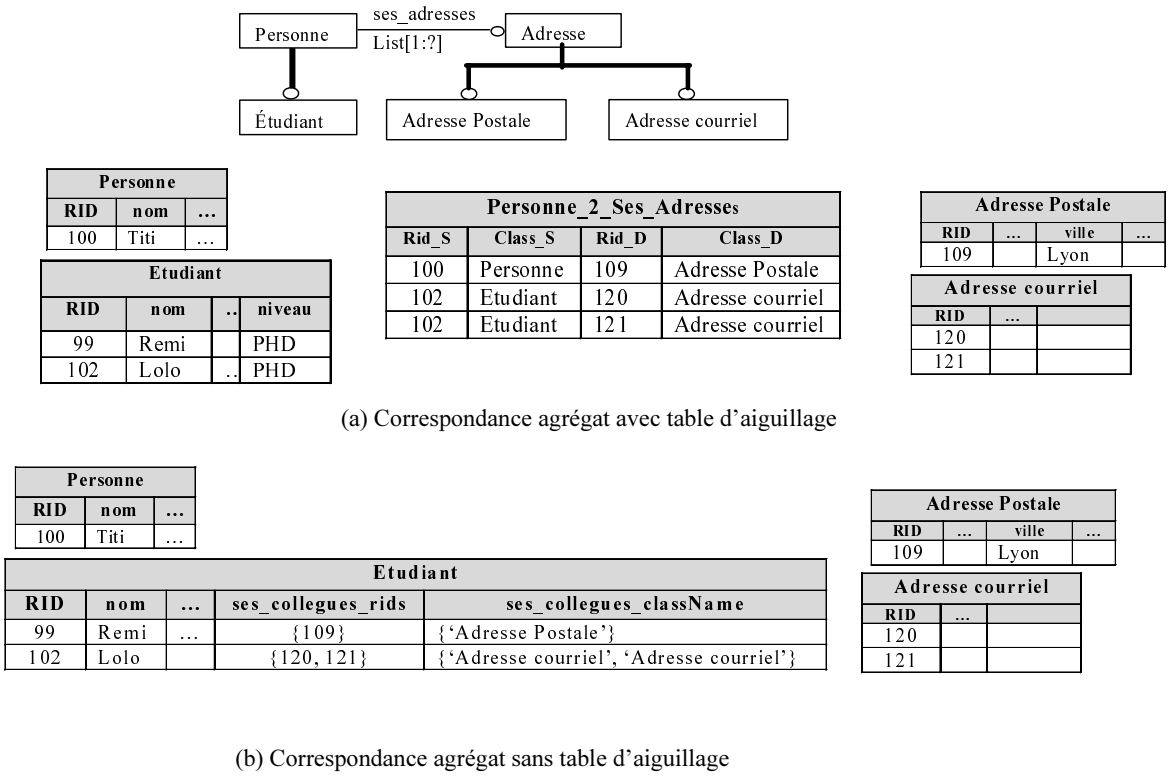


FIG. 4.13 – Les différentes approches de représentation des collections en relationnel.

– **les collections de type complexe.** Comme pour la représentation des types *class_ instance _ type* et *entity_ instance_ type*, les exigences que doivent remplir de la représentation des collections portent la gestion du polymorphisme et du calcul des inverses.
Deux solutions sont possibles pour la représentation des collections :

1. une première solution consiste à utiliser une **table d'aiguillage** entre les tables des classes en relation (comme sur la figure 4.13a). Dans cette table, toutes les instances qui ont le même *rid* (i.e. celle de l'instance référençant) constituent les éléments d'un agrégat. Dans cette approche, la récupération d'un agrégat nécessite une requête sur la table, de plus cette table peut être parcourue dans les deux directions. La figure 4.13a présente un exemple illustre cette approche de représentation.
2. la deuxième solution consiste à définir deux colonnes pour les propriétés de type agrégat. Une colonne pour l'identifiant (rid) des instances contenues dans l'agrégat. Le nom de la colonne est suffixé par la chaîne '_rids' et son co-domaine est un tableau d'entier (**INT8[]**). La deuxième colonne, de co-domaine **VARCHAR[]**, pour enregistrer le nom des tables d'où proviennent les IDs des instances de la première colonne dans le même ordre, c'est-à-dire que le nom de la table à l'indice i dans le tableau, est la table de l'instance à l'indice i dans la première colonne.

Pour les mêmes raisons que pour la représentation des types *class_ instance_ type* et *entity_ instance_ type*, notre choix de représentation s'est porté sur l'approche à deux colonnes lorsque le calcul des inverses n'est pas nécessaire et sur l'approche par table d'aiguillage dans le cas inverse.

3.1.3 Connaissance descriptive : les propriétés

Nous discutons dans cette sous-section de la représentation en relationnel des différentes catégories de propriétés que propose le modèle d'ontologie PLIB (cf. section 3.2 de l'annexe A) :
– propriétés caractéristiques,
– paramètres du contexte,
– propriétés dépendantes du contexte.

3.1.3.1 Les propriétés caractéristiques

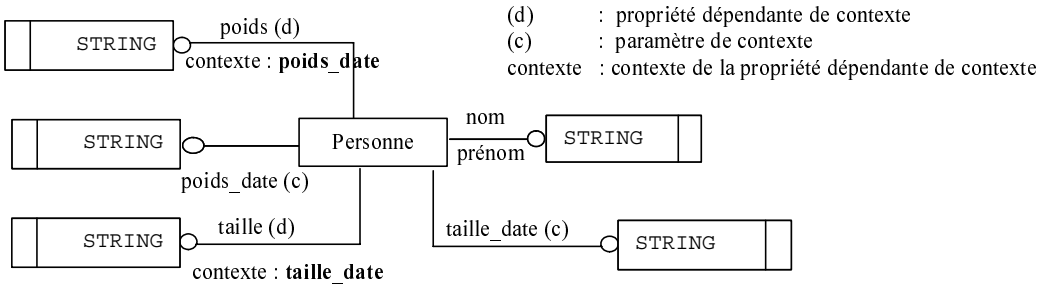
Les propriétés représentent les propriétés intrinsèques des instances des classes. Chaque propriété caractéristique est traduite par une colonne pour les propriétés de types simples et deux colonnes ou une table d'aiguillage pour les propriétés de type association ou collection (cf. figure 4.14).

3.1.3.2 Propriétés dépendantes du contexte et paramètres du contexte

Les propriétés dépendantes du contexte (PDC) sont les propriétés ayant une dépendance fonctionnelle avec au moins un paramètre de contexte. Chaque propriété dépendante du contexte (PDC) est traduite en une colonne (cf. propriétés *poids* et *taille* sur la figure 4.14). Les paramètres de contexte, associés à des PDC, sont aussi traduits par des colonnes. Un paramètre de contexte pouvant être utilisé dans plusieurs PDC (exemple la propriété *date* pour les propriétés

poids et *taille*), pour pouvoir les distinguer les unes des autres, on créera autant de colonnes qu'il y a de propriétés dépendantes du contexte qui utilisent la même propriété de contexte. Les noms des colonnes des paramètres de contextes sont préfixés par celui de la colonne de la propriété dont elle dépend (cf. les colonnes *taille_ date* et *poids_ date* sur la figure 4.14).

Remarque :
Notons que c'est par souci de clarté de compréhension que nous avons représenté deux propriétés de contexte distinctes *date* (*taille_ date* et *poids_ date*) dans la figure 4.14. En réalité dans PLIB, nous pouvons n'avoir qu'une seule propriété *date* qui serait utilisée à la fois par les propriétés dépendantes de contextes *poids* et *taille*.



Personne						
RID	**nom**	**prénom**	**poids**	**poids_date**	**taille**	**taille_date**
10	Dupont	Jack	60kg	12-06-06	180cm	12-06-06
11	Duroc	Sylviane	40kg	01-10-04	168cm	01-10-04

FIG. 4.14 – Représentation des différents types de propriétés PLIB en relationnel.

Les règles de correspondances que nous venons de définir entre le modèle d'ontologie PLIB et le relationnel sont utilisées par les programmes de génération de la structure de tables des classes des ontologies, et par les différentes APIs que nous avons développées pour fournir l'accès à cette structure.

Dans la section suivante, nous présentons comment les modèles conceptuels (schémas des instances) sont représentés dans notre prototype de BDBO.

3.2 Représentation du modèle conceptuel des instances

Dans la section 3.2 du chapitre 3, nous avons proposé de représenter explicitement le modèle conceptuel des instances pour faciliter l'évaluation des requêtes polymorphes et pour faciliter la gestion du cycle de vie des instances. Nous présentons dans cette section la structure de tables que nous avons définie pour la représentation du modèle conceptuel des instances dans la base de données.

Deux solutions sont envisageables pour la définition de la structure des tables pour le stockage du modèle conceptuel :

- une première approche *ad hoc* dans laquelle nous définissons *manuellement* un schéma logique des tables comme dans l'exemple de la figure 3.10 du chapitre précédent.
- une deuxième approche *systématique* qui se base sur le modèle d'extension d'une classe PLIB définit dans l'ISO13584-25 [80] (cf. figure A.7 dans l'annexe A).

  Rappelons que dans le modèle d'extension d'une classe PLIB, les données à base ontologique sont représentées sous forme de listes de couples propriété-valeur au travers l'attribut *population* de l'entité *class_extension*. Chaque instance est identifiée par un sous-ensemble de propriétés spécifiées par l'attribut *instance_identification*. Ces propriétés forment donc la clé primaire des instances (clé sémantique). Pour la gestion des versions et révisions des populations des instances des classes, le modèle d'extension définit les attributs *content_version* et *content_revision* dans l'entité *class_extension*.

Les avantages de l'utilisation de la deuxième approche par rapport à la première sont les suivants :

- le modèle d'extension de PLIB étant défini en EXPRESS, sa traduction en modèle logique peut être réalisée *automatiquement* par le composant de notre prototype qui permet la génération la structure des tables que nous avons utilisé pour les parties *ontologie* et *méta-schéma* de notre architecture.
- Lors de la lecture des populations d'instances de classes PLIB, le peuplement de la structure des tables peut être réalisé *automatiquement* par le composant d'importation d'instances que nous avons utilisé pour peupler la partie *ontologie* et la partie *méta-schéma* de notre architecture.
- Enfin, l'accès aux données peut être réalisée en utilisant les APIs que nous avons définies pour la partie *ontologie* (cf. section 4.1).

Au contraire, toutes les tâches indispensables pour la représentation des modèles conceptuels dans l'approche *ad hoc* doivent être réalisées *manuellement*. Pour des raisons de simplicité et d'efficacité notre choix s'est donc porté sur la deuxième solution.

Notons toutefois que l'utilisation de la deuxième approche a nécessité de faire quelques modifications pour améliorer les performances d'accès aux données. En effet, le modèle d'extension de PLIB est défini de sorte que chaque instance soit porteuse de sa propre structure, autrement dit l'extension de PLIB ne permet pas de spécifier explicitement l'ensemble des propriétés qu'utilisent les instances des classes. Pour le savoir, il est nécessaire de faire l'union des propriétés utilisées par toutes les instances de la classe. Cette opération peut être coûteuse lorsque les classes ont beaucoup d'instances et/ou initialisent beaucoup de propriétés. Dans le cas où toutes les instances initialisent les mêmes propriétés, le modèle d'extension PLIB définit un attribut de type booléen nommé *table_like* qui permet de spécifier si toutes les instances utilisent ou non les mêmes propriétés. Dans le cas où les instances utilisent les mêmes propriétés (*table_like*= TRUE), alors la première instance est utilisée pour déterminer les propriétés utilisées par la classe. Sinon l'algorithme de chargement d'une extension de classe dans la base de données calcule l'union des propriétés utilisées par au moins une instance. C'est le résultat qui est représenté

en tant que modèle conceptuel.

L'utilisation du modèle d'extension PLIB pour la représentation des MCs, amène à stocker une instance *factice* (sous forme de liste de couples de propriété-valeur=null) dans la base de données pour chaque version de classe. Néanmoins, afin d'accélérer l'accès au modèle conceptuel correspond à la version courante, nous avons décidé d'offrir un accès direct aux propriétés utilisées dans celle-ci. Cela est réalisé en ajoutant un attribut (*properties*) de type collection dans l'entité *class_ extension*. Toute opération (ajout de propriétés, suppression de propriétés, etc.) sur la structure des instances sera systématiquement répercutée sur cet attribut (cf. figure 4.15).

3.3 Gestion du cycle de vie des données à base ontologique

Cette gestion correspond à une des exigences que nous sommes fixés dans la définition de notre architecture *OntoDB*. Cette exigence est en particulier motivée par le fait que dans un contexte de base de données entre plusieurs composants industriels, il est essentiel de pouvoir tracer les instances qui étaient valide à une certaine date car ceux-ci peuvent avoir été utilisé dans les produits fabriqués alors et donc la maintenance doit continuer à être assurée. Dans la section 3.2 du chapitre 3, nous avons proposé des solutions aux quatre problèmes à résoudre pour la gestion du cycle de vie des données à base ontologique :

(1) pour la duplication de l'instance, nous avons proposé d'associer à chaque classe la définition d'une ou plusieurs contraintes d'unicité permettant une identification sémantique unique des instances,

(2) pour le calcul du schéma des instances, nous avons proposé d'archiver le schéma des instances de classe pour chaque version,

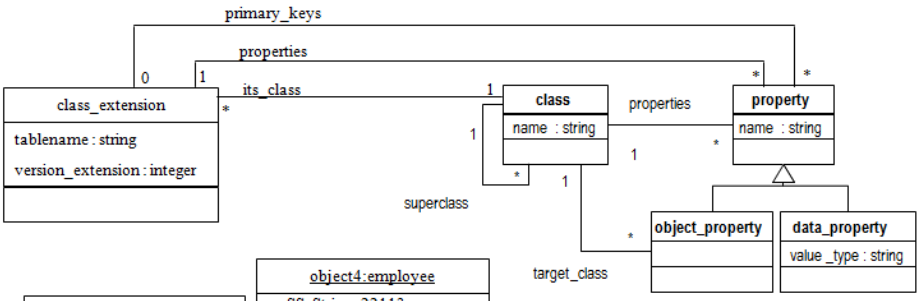
(3) pour le tracé du cycle de vie des instances, nous avons proposé d'ajouter à chaque instance la liste des versions de classes pour lesquelles cette instance était valide, enfin,

(4) pour l'ambiguïté de la sémantique de la valeur nulle, nous avons proposé d'exploiter le schéma des instances stockées dans la base de données pour déterminer si une valeur nulle d'une colonne d'une table est due au fait que la propriété n'était pas fournie pour l'instance au fait qu'elle n'appartient pas, alors, ou schéma logique de la classe.

Concernant le premier problème, dans notre implémentation actuelle, la liste de propriétés formant la clé sémantique de l'instance est représentée dans le modèle conceptuel déjà représenté dans la base de données (cf. section 3.2). L'attribut *instance_ identification* de l'entité *class_ extension* est utilisé à cet effet. Dans le modèle UML de la figure 4.15a, la liste de propriétés formant la clé est donnée par la propriété *primarykeys* dans la classe *class_ extension*.

Concernant le deuxième problème, l'archivage du schéma des instances pour chaque version de l'extension d'une classe est réalisé (1) en ajoutant un attribut *version_ extension* qui permettra d'indiquer les numéros des différentes versions de la population d'instances d'une classe, (2) en représentant pour chaque version de l'extension les propriétés fournies (*properties*) et (3) en représentant la clé sémantique (*primary keys*). La figure 4.15a présente le modèle UML du schéma permettant la représentation du schéma des instances que nous avons étendu pour la gestion des versions.

(a)



(b)

object3:student
noSS :String=82984
name:String=Dung
lastname:string=Nguyen

object1:student
noSS :String=02489
name:String=Woppy
lastname:string=Golber
gender:char=F

object4:employee
noSS :String=22113
name:String=Jean
lastname:string=Stephane
salaire:double=1300
gender:char=H
service:string=UNILOG

class_extension

IDClassExtension	its_class	version_extension
1	id_ student	001
2	id_employee	001
3	id_ student	002
4	id_employee	002

properties

IDClass_extension	IDProperty
1	id_noSS
1	id_name
1	id_lastname
1	id_gender
2	id_noSS
2	id_name
2	id_lastname
2	Id_salary
3	id_noSS
3	id_name
3	id_lastname
4	id_noSS
4	id_name
4	id_lastname
4	id_salary
4	id_gender

primary_keys

IDClass_extension	IDProperty
1	id_noSS
2	id_noSS
3	id_noSS
4	id_noSS

(c)

id_student

OID	id_noSS	id_name	id_lastname	id_gender	version_min	version_max
100	02489	Woopy	Golber	F	001	NULL
110	82984	Dung	Nguyen	NULL	002	NULL

id_employee

OID	id_noSS	id_name	id_lastname	id_gender	id_service	id_salary	version_min	version_max
103	13457	Jean	Claude	NULL	CRCFAO	1500	001	002
110	22113	Jean	Stephane	H	UNILOG	1300	002	NULL

FIG. 4.15 – Gestion du cycle de vie instances sur l'ontologie simplifiée

Concernant le troisième problème, dans notre implémentation, nous faisons l'hypothèse qu'une instance qui a cessé d'appartenir à une classe ne peut y réapparaître ultérieurement, de sorte que la liste des versions de validité peut se représenter comme un intervalle ($Version_{min}, Version_{max}$). Ces propriétés, ajoutées à chaque instance, permettent de savoir la validité d'une instance donnée. La propriété $Version_{min}$ indique la version de la population des instances à partir de laquelle l'instance a commencée d'exister. $Version_{max}$ indique le numéro de la dernière version de la population à partir de laquelle elle a été supprimée. Lorsqu'une instance est valide dans la dernière version en cours, sa $Version_{max}$ vaut **NULL** (cf. figure 4.15c).

Exemple.
Supposons que nous souhaitons intégrer une nouvelle version des populations d'instances des classes *Student* et *Employee*, les instances de la version précédente (cf. figure 4.15) ayant subit quelques modifications.

- **Classe Student.**
  - la propriété *gender* n'est plus fournie dans les instances des classes,
  - une nouvelle instance ($object_3$) est insérée dans la nouvelle version,
- **Classe Employee.**
  - l'instance $object_2$ de *Employee* de la version précédente n'est plus fournie dans la nouvelle (cf. figure 3.9 du chapitre 3),
  - une nouvelle instance ($object_4$) est insérée dans la nouvelle version.

L'intégration de ces nouvelles versions d'instances, implique automatiquement la représentation du nouveau modèle conceptuel des classes (cf. figure 4.15b). Les lignes en gris indiquent les données correspondantes aux nouvelles versions. On peut notamment remarquer que (1) pour la classe *Student*, la propriété *gender* ne fait pas partie du modèle conceptuel, et (2) pour la classe *Employee*, la propriété *gender* fait partie du modèle conceptuel.

Ces changements sur le modèle conceptuel, impliquent également des modifications (1) au niveau de la structure tables des instances des classes : ajout d'une nouvelle colonne *id_gender* dans la classe des *Employee*, et (2) au niveau des instances :

- les instances des versions précédentes sont archivées (i.e., leur $version_{max}$ initialisé à *002*-le numéro de la dernière version (cf. première instance de la classe *Employee*),
- les instances des classes de la nouvelle version sont introduites dans leurs tables respectives (cf. deuxième instance de la classe *Employee*),
- la $version_{min}$ de ces instances sont initialisées à *002* et leur $version_{max}$ sont à *NULL* indiquant qu'elles sont des instances courantes,
- les colonnes des propriétés des instances qui ne sont plus fournies dans la nouvelle version (cf. propriété *gender* de la classe *Student*) ainsi que les colonnes des propriétés qui n'étaient pas fournies dans la version précédente (cf. propriété *gender* de la classe *Employee*), sont initialisées à *NULL*.

3.4 Lien entre ontologie et données à base ontologique

Dans la section 3.1.4.2 du chapitre précédent, nous avons proposé de matérialiser ce lien en nommant les tables et colonnes de la partie *données* en utilisant l'identifiant (URI, BSU) des classes et propriétés et d'implémenter deux fonctions : une fonction de *nomination* et une fonction d'*abstraction* qui permettront de faire le lien entre les concepts des ontologies et les données.

Suivant les caractéristiques du modèle d'ontologie utilisé, diverses solutions sont envisageables pour matérialiser ce lien.

- Une première approche, fonctionnelle, consisterait à utiliser l'identifiant universel (URI, BSU) des concepts pour former le nom des tables, vues et colonnes, etc. dans la base de données. Ces identifiants doivent évidemment respecter les conditions exigées par le SGBD cible, qui sont entre autres : la longueur des noms, l'absence de caractère spéciaux, etc. Notons que ces exigences constituent l'inconvénient majeur pour cette approche car il n'existe pas forcement dans les ontologies des contraintes contrôlant les identifiants.

- Une deuxième approche, associative, consisterait à représenter directement la correspondance entre les concepts de l'ontologie et les noms de leur représentation : tables et colonnes, sous forme d'une table comme sur la figure 4.15. La colonne *tablename* de la table *class_extension* donne le nom de la table des instances d'une classe. La colonne *columnName* de la table *properties* donne le nom de la colonne d'une propriété. Les fonctions *Nomination* et *Abstraction* sont implémentées en faisant des requêtes sur ces tables. Ce qui nécessite des accès en $O(log_2(n))$.

- Une troisième approche, qui est celle que nous proposons, consiste à utiliser les OIDs de la représentation des concepts dans l'ontologie pour nommer les tables et colonnes dans la partie données. En effet, ainsi que nous l'avons vu dans la figure 3.1, les concepts, classes et propriétés, représentés dans la base de données, sont associés à un OID dans leurs tables respectives (CLASS, DATA_PROPERTY, OBJECT_PROPERTY). Cet identifiant, qui est unique dans la base de données, peut être utilisé pour définir le nom des tables et colonnes de façon unique. L'avantage de cette approche est que les fonctions *Nomination* et *Abstraction* ne nécessitent la réalisation d'aucune requête supplémentaire dans la base de données. Néanmoins, les noms de tables et de colonnes n'étant pas mnémoniques, il sera nécessaire que toutes les interfaces d'accès aux données supportent l'utilisation des fonctions d'*Abstraction* et *Nomination* ce qui permettra, par exemple, au niveau des interfaces d'accès graphiques de représenter les noms de tables et de colonnes par des noms *traduits dans la langage de l'utilisateur.*

Dans notre implémentation, le nom des tables (respectivement des colonnes) est la concaténation de la chaîne "E_" (respectivement "P_") avec l'OID de la classe (respectivement de la propriété). L'algorithme 18, présente une vue simplifiée des fonctions de *nomination* et d'*abstraction.* Dans ces algorithmes, la fonction *getID :* $\times \in C \cup P \rightarrow$ *string* permet de retourner l'id interne à la base de données du concept donné en paramètre, la fonction *substring : integer x string* $\rightarrow$ *string* permet d'extraire une sous-chaîne à partir d'un indice i de la chaîne en paramètre. Les

types *TABLE* et *COLUMN* peuvent être considérés comme des chaînes de caractères (String).

Algorithme 18 Algorithme des fonctions de *Nomination* et d'*Abstraction*.

```
FUNCTION Nomination(x : C ∪ P) → TABLE ∪ COLUMN :
    SI x ∈ P THEN
        RETOURNER "P_" + getID(x) ;
    FINSI ;
    SI x ∈ C THEN
        RETOURNER "E_" + getID(x) ;
    FINSI ;
END FUNCTION ;

FUNCTION Abstraction(x : TABLE ∪ COLUMN) → C ∪ P :
    SI x[1] =="P" THEN
      requête : = SELECT *
                  FROM property
                  WHERE id = substring(2,x) ;
      result := EXECQUERY requête ;
       RETOURNER result ;
    FINSI ;
    SI x[1] =="C" THEN
      requête : = SELECT *
                  FROM class
                  WHERE id = substring(2,x) ;
      result := EXECQUERY requête ;
      RETOURNER result ;
    FINSI ;
END FUNCTION ;
```

4 Méthodes d'accès aux ontologies

Ainsi que nous l'avons déjà souligné, l'intérêt essentiel d'une BDBO est de permettre l'accès aux données "au niveau connaissance", c'est-à-dire en termes des *informations représentées* et non en termes de valeurs enregistrées. Nous présentons dans cette section les principales méthodes d'accès que nous avons implémentées pour la manipulation des ontologies et des données à base ontologique représentées dans une BDBO. Nous présenterons dans la section 4.1, les méthodes d'accès au moyen d'interfaces fonctionnelles (APIs).

Dans la section 4.2, nous présentons l'outil *PLIBEditor* qui offre un accès graphique ergonomique pour les ontologies et données à base ontologique.

4.1 API d'accès

Nous présentons séparément, dans cette section, les APIs pour l'accès aux concepts des ontologies et les APIs d'accès aux données à base ontologique. Ce sont ces APIs qui seront ensuite

utilisées pour permettre le développement des interfaces graphiques ou langagières.

4.1.1 Accès aux ontologies

Pour l'accès aux concepts des ontologies stockées dans la partie *ontologie*, nous avons proposé quatre niveaux d'APIs. D'une part une API indépendante de tout modèle d'ontologie (API dite "à liaison différée"), elle-même, structurée en deux couches. D'autre part deux APIs différentes générées à partir de chaque modèle particulier d'ontologie (API dite "à liaison préalable") et suivant soit une approche fonctionnelle, soit une approche orienté-objet.

4.1.1.1 API à liaison différée (*late binding*). Les APIs à liaison différée, sont celles que nous avons décrites dans le chapitre précédent à la section 2.2) et qui visent à offrir un ensemble de fonctions pour l'accès aux ontologies *indépendamment du modèle d'ontologie particulier*. La mise en œuvre de cette interface nécessite la représentation à la fois du modèle d'ontologie et du méta-modèle du modèle d'ontologie dans la base de données. L'APIs d'accès à liaison différée que nous avons développée, est constitué de deux couches (*API PostgreSQL* et l'*API Java*).

1. **API PostgreSQL.** L'*API PostgreSQL* est une interface fonctionnelle entièrement programmée dans l'un des langages procéduraux offert par le SGBDRO PostgreSQL (plpgsql).
2. **API Java.** L'*API Java* est également une interface de type liaison différée ('*late binding*'), ce qui fait d'elle également une interface générique pour tout modèle d'ontologie.

4.1.1.1.1 API PostgreSQL

API PostgreSQL met en œuvre toutes les primitives de bases pour la gestion des ontologies dans la base de données. L'*API PostgreSQL*, comme son nom l'indique, est implémentée dans le langage de programmation natif (plpgsql) du SGBDRO PostgreSQL et sa réalisation est stockée dans la base de données sous forme d'un ensemble de procédures stockées (stored procedure). La raison d'être de cette API est qu'elle fournit toutes les primitives de base pour les accès aux données et qu'elle peut être appelés dans d'autres langages de programmation évolués (Java, C++, C, etc.) via JDBC ou ODBC, évitant ainsi de re-programmer toutes ces primitives dans chacun de ces langages.

L'*API PostgreSQL*, au stade actuel de notre implémentation, contient environ 170 fonctions qui interviennent dans la mise en œuvre d'une quarantaine fonctions de bases effectivement utilisées par les applications et autres APIs. Ce sont, entre autres, les fonctions définies dans l'algorithme 20 ci-après. Notons que, dans les signatures des fonctions, *oid*, *nom_ entité* et *nom_ attribut* sont de type *chaîne de caractères*, et *value* appartient à une *union de tous les types de bases*.

Comme nous l'avons déjà précisé dans la section 2.2 chapitre précédent, la mise en œuvre des fonctions d'une API à liaison différée de sorte quelle soit indépendante, à la fois, du modèle d'ontologie et de la structure des tables dans laquelle sont représentées les ontologies, nécessite la représentation dans la base de données (1) du modèle d'ontologie, et (2) du méta-modèle du modèle d'ontologie (réflexif). Dans notre prototype, ceux-ci sont justement représentés dans

Algorithme 19 Quelques fonctions de l'*API PostgreSQL*

create_instance : nom_entité → oid :
pour la création d'une instance d'une entité EXPRESS.
set_value_attribute : oid x nom_entité x nom_attribut x $value^n$:
pour l'initialisation de la valeur d'un attribut d'une instance d'une entité EXPRESS.
reset_value_attribute : oid x nom_entité x nom_attribut :
pour la ré-initialisation de la valeur d'un attribut de type collection d'une instance d'une entité EXPRESS.
get_value_attribute : oid x nom_entité x nom_attribut → $value^n$:
pour la récupération de la valeur d'un attribut d'une instance d'une entité EXPRESS.
usedin : oid x nom_entité x nom_attribut_ou_relation → $value^n$:
pour la récupération de toutes les instances qui référencent une instance donnée à partir de l'attribut ou relation *nom_attribut_ou_relation* de l'entité *nom_entité*.
get_final_type : oid x nom_entité → nom_entité :
pour la récupération du type effectif d'une instance

la partie *méta-schéma* de notre architecture. Toutes les fonctions de l'API sont implémentées en faisant des accès à cette partie de notre architecture. Notons aussi que tous les accès à la partie *méta-schéma* elle-même, par les fonctions de l'API, sont faites de sorte à être *le plus possible* indépendant de sa structure des tables. Pour cela, nous avons structuré cet API en deux sous-ensembles. Tous les accès nécessitant d'accéder la structure des tables sont codées dans quelques fonctions (qui font partie de l'*API PostgreSQL*). Le fait qu'il soit programmé ainsi, nous permettra de nous adapter *facilement* à des éventuelles modifications des règles de correspondances (cf. annexe C) que nous avons définies pour transformer les mécanismes du langage EXPRESS en relationnel. Dans le cas d'une modification d'une règle de correspondances, nous aurions qu'à re-coder uniquement le sous-ensembles des fonctions de l'*API PostgreSQL* qui implémentent les règles de correspondances sans aucune incidence sur les autres fonctions de l'API.

4.1.1.1.2 API Java

L'*API Java* est un deuxième niveau d'interface d'accès et de manipulation des ontologies stockées dans la base de données. Cette API est une sur-couche de l'*API PostgreSQL*. Elle fournit en une syntaxe particulière en Java en appelant les fonctions de l'*API PostgreSQL* via JDBC.

Cette API ne présente pas de difficultés techniques majeures pour sa mise en œuvre car toutes les fonctions qu'elle contient se contentent de faire appel aux fonctions équivalentes de l'*API PostgreSQL*. L'algorithme 20, montre un exemple d'appel d'une fonction de l'*API PostgreSQL*.

4.1.1.2 API à liaison préalable (*early binding*)

Une *API à liaison préalable* est une API définie spécifiquement pour un modèle d'ontologie. Les classes et les méthodes de cette API correspondent respectivement aux entités et aux accesseurs (constructeur, lecture, écriture, destructeur, etc.) définis pour chaque attribut du modèle d'ontologie. Deux implémentations sont possibles pour ce type d'APIs objets selon que l'on instancie ou non dans le programme les entités existant dans la base de données. Dans le premier

Algorithme 20 Exemple d'appel d'une fonction de l'*API PostgreSQL* par l'*API Java*

```
public class API_Java{
...
  public static String[] get_value_attribute(String rid, String entity_type, String attribute) {
    String[] res = null ;
    Array tab ;
    if (is_loaded_database()) {
       try {
              Statement st = database.createStatement() ;
              //Appel de la fonction get_attribute de l'API PostgreSQL
              ResultSet rs = st.executeQuery("select get_attribute(" + Rid + ",'"
                                    + entity_type + "','" + attribute + "')") ;
              //On récupère le résultat de la requête
              rs.next() ;
              tab = rs.getArray(1) ;
              if (tab != null) {
                     res = (String[]) tab.getArray() ;
              }
       } catch (SQLException e) {
              // En cas d'erreurs
              Debug.warning("GET_ATTRIBUTE : an occurred error is raised ", DEBUG) ;
              e.printStackTrace() ;
              }
       } else {
              Debug.error("GET_ATTRIBUTE : database is not loaded ") ;
              }
    //On retourne le résultat de l'appel de la fonction get_attribute de l'API PostgreSQL.
    return res ;
  }
...
}
```

cas, l'API est dite "objet" sinon elle est dite "fonctionnelle". Dans l'implémentation de notre prototype, nous proposons ces deux types d'APIs pour l'accès aux ontologies.

- L'**API PLIB** : cette API fonctionnelle est composée de classes Java associées à chaque entité du modèle d'ontologie utilisé et dotées de méthodes *statiques* qui font directement des accès à la base de données lors de chaque appel. Ces méthodes de la classe permettent l'accès, à partir de leur identifiant, aux instances de la base de données, sans les ré-instancier dans le programme exploitant l'API.

- L'**API hibernate** : cette API objet est implémentée en utilisant le framework hibernate et permet de ré-instancier au sein d'un programme java, toutes les instances d'entité existant dans la base de données.

Nous décrivons ci-dessous dans la section 4.1.1.2.1 l'API PLIB et dans la section 4.1.1.2.2 l'*API hibernate*.

4.1.1.2.1 API PLIB

Toutes les méthodes de l'API PLIB étant *statiques* et faisant directement des accès dans la base de données, cette API est greffée sur l'API à liaison différée *API Java*. Toutes les méthodes des classes de l'API font directement des appels aux fonctions de l'*API à liaison différée*, comme illustré dans l'algorithme 21.

Algorithme 21 Exemple d'une classe de l'*API PLIB* faisant appel à l'API Java.

```
package API_PLIB;
 public class PLIB_Class{
    //création d'une instance avec toutes les valeurs des attributs initialisées à nul
  public static int CreateInstance(){
    //appel de la fonction create_instance de l'API à liaison différée.
    return API_Java.create_instance('class');
 }
  // récupération du nom de la classe ayant l'ID donné en paramètre;
  public static String getName(Integer ID){ ... }
  //initialisation du nom de la classe ayant l'ID donné en paramètre.
  public static void setName(Integer ID, String name){...}
  // récupération de la super-classe de la classe ayant l'ID donné en paramètre;
  public static Integer getits_superclass(Integer ID){
    return API_Java.get_value_attribut(ID,'class','its_superclass');
    }
  //initialisation de la super-classe de la classe ayant l'ID donné en paramètre.
  public static void setits_superclass(Integer ID, String value){
    API_Java.set_value_attribut(ID,'class','its_superclass',value);
  }
```

La difficulté de la mise en œuvre de cette API est essentiellement due à la complexité du

modèle d'ontologie PLIB qui est constitué de 218 entités avec un nombre total de 486 attributs. Si on estime qu'on dispose au moins de deux méthodes (get et set) par attributs des entités et que chaque entité disposera d'un constructeur et d'un destructeur, cela reviendra à implémenter au moins 2 * 486 + 2 * 218 méthodes au total (soit 1408). La mise en œuvre d'une telle API manuellement est difficile, et encore plus si le modèle d'ontologie est susceptible d'être modifié.

Pour des raisons de simplicité et surtout dans l'esprit de s'adapter à toute éventuelle modification du modèle d'ontologie (compte tenu de notre objectif O_2), nous avons opté pour une génération automatique de cette API en utilisant les techniques de l'IDM. Le programme du générateur que nous avons implémenté dans le langage EXPRESS, parcourt chaque entité du modèle EXPRESS fourni en paramètre, et à générer une classe java avec ses méthodes (i.e., constructeur, destructeur et accesseurs). Le corps des accesseurs aux attributs des entités est généré de façon à faire des appels aux fonctions de l'*API à liaison différée*. Les appels aux fonctions de l'*API à liaison différée* dans les différentes accesseurs sont initialisés suivant les entités et leurs attributs.

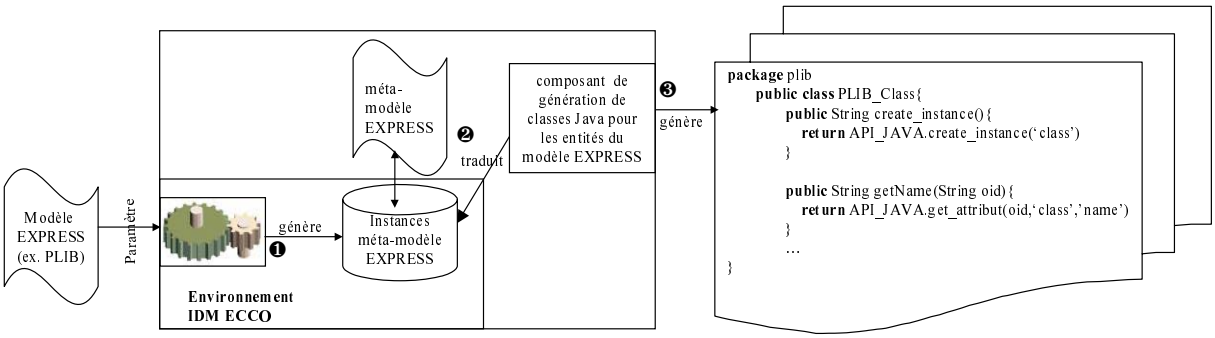


FIG. 4.16 – Architecture du composant de génération de classes de l'API *PLIB API*

4.1.1.2.2 API Hibernate

Cette API est implémentée en utilisant le "framework" hibernate [70]. *Hibernate* permet au moyen de fichiers de configuration XML de définir des correspondances entre des classes Java (appelé POJOs) et le schéma relationnel d'une base de données. Il permet ainsi un accès transparent aux données des bases de données *sans* pour autant écrire une seule ligne de code SQL : Hibernate génère dynamiquement les requêtes SQL d'accès aux données. Un deuxième avantage du framework *hibernate* est qu'il permet de gérer les objets déjà récupérés dans la base de données dans des caches d'objets et d'assurer leur synchronisation avec la base de données. Ce qui permet de limiter les accès à la base de données pour les instances déjà chargées en mémoire centrale.

La mise en œuvre de cette API nécessite donc de définir des classes POJOs pour chacune des entités du modèle d'ontologie PLIB et les fichiers de correspondances XML entre les classes et les tables de la partie *ontologie*. Une façon simple et évidente de définir les classes POJOs et fichiers de correspondances XML à partir des tables de la base de données est d'associer à

chaque table de la base de données une classe Java. Les colonnes des tables sont accédées par des accesseurs définis pour chacun d'elles dans les classes POJOs. Pour les attributs dont les valeurs sont stockées dans des tables auxiliaires (comme les attributs de type association, collection ou union de type, etc.), des méthodes supplémentaires sont définies pour accéder effectivement aux objets référencées en "traversant" les classes POJOs associées aux tables d'aiguillages.

L'algorithme 22 présente un exemple de classes POJOs et de fichiers de configuration XML pour accéder aux concepts des ontologies. L'exemple se base sur le modèle d'ontologie simplifié de la figure 3.1b dans le chapitre 3 pour lequel nous avons créé une classe POJO pour chaque table du modèle logique généré à partir des règles de correspondances en le langage EXPRESS et PostgreSQL (cf. annexe C) du modèle. Rappelons dans ces règles, nous avons proposé de représenter les associations par une table d'aiguillage bilatère entre les tables des entités en relation. Ce qui explique l'existence de la classe *property_2_targetclass* qui lie les tables des entités *Object_Property* et *class*. Afin de rendre transparent notre implémentation (i.e., voiler les tables d'aiguillages), nous proposons de traverser les classes des tables d'aiguillage par des méthodes (cf.*getTargetclass()*). Remarquez, sur la figure 22, que les attributs référençant les instances des classes des tables d'aiguillages sont "*suffixés*" par un "_".

Compte tenu de la complexité du modèle d'ontologie PLIB et du nombre de tables du schéma de la partie *ontologie* (plus de 600 tables), l'implémentation manuelle de cette API est très difficile comme pour l'*API PLIB*. Afin de pouvoir s'adapter à toute modification du modèle d'ontologie, nous avons opté ici encore pour une génération automatique de cette API en utilisant les techniques de l'IDM. L'architecture du générateur est résumée dans la figure 4.17. Elle consiste, à partir du modèle EXPRESS donné en paramètre, à générer (1) les classes POJOs pour l'accès aux ontologies et (2) les fichiers de configuration XML pour la correspondance entre les classes POJOs et les tables de la base de données.

4.1.2 Accès aux instances des classes

Pour l'accès aux instances des ontologies, nous avons implémenté une API (API Extension) qui offre des primitives (1) pour la gestion de l'extension de chaque classe et (2) pour la gestion (création, lecture, modification, suppression) des instances des classes des ontologies. Les règles de correspondances que nous avons précédemment décrites pour la définition des tables des classes d'instances d'une ontologie PLIB (cf. section 3.1), sont *codées* dans l'ensemble des fonctions de l'API.

La mise en œuvre des fonctions de cette API nécessite d'accéder à la définition des concepts (classes et propriétés) stockés dans la partie *ontologie* pour l'application de ces différentes règles. Cet accès à la partie *ontologie* est réalisé au moyen de l'API Java décrit ci-dessus (cf. section 4.1.1.1.2). L'algorithme 23 présente un sous-ensemble significatif des fonctions définies et l'algorithme 24 présente le principe de l'implémentation de ces fonctions.

Algorithme 22 Exemple de classes POJOs et de fichiers de configuration XML de l'*API Hibernate.*

Les classes POJOS

```
package API_Hibernate;
  // classe POJO associée à la table "Object_Property"
  public class Object_Property extends Property{
    private property_2_targetclass targetclass_ ;// on pointe sur la table d'aiguillage.
    private property_2_targetclass getTargetclass_(){
        // on retourne l'instance de la table d'aiguillage.
        return targetclass_ ;
    };
    public class getTargetclass(){
         // on traverse la table d'aiguillage pour retourner l'instance de la classe "class".
        return this.getTargetclass_().getDestination();
    };
...
// classe POJO associée à la table d'aiguillage "property_2_targetclass"
  public class property_2_targetclass{
    private Object_Property source;// qui référence l'instance de la table source (rid_S)
    public Object_Property getSource(){
        return source;
    };
    private class Destination;// qui référence l'instance de la table destination (rid_D)
    public class getDestination(){
        return Destination;
    };
...
    }
```

Les fichiers de configuration XML

```
  <hibernate-mapping>
      <class name="Property" table="Property">
         <id name="id" column="RID"/>
         <property name="name" column="name"/>
         ...
      </class>
      <subclass name="Object_Property" table="Object_Property" extends="Property">
         <many-to-one name="targetclass" column="rid_D" class="property_2_targetclass"/>
      </subclass>
      <class name="property_2_targetclass" table="property_2_targetclass">
         <id name="id" column="ID"/>
         <many-to-one name="Destination" column="rid" class="class"/>
         ...
      </class>
  </hibernate-mapping>
```

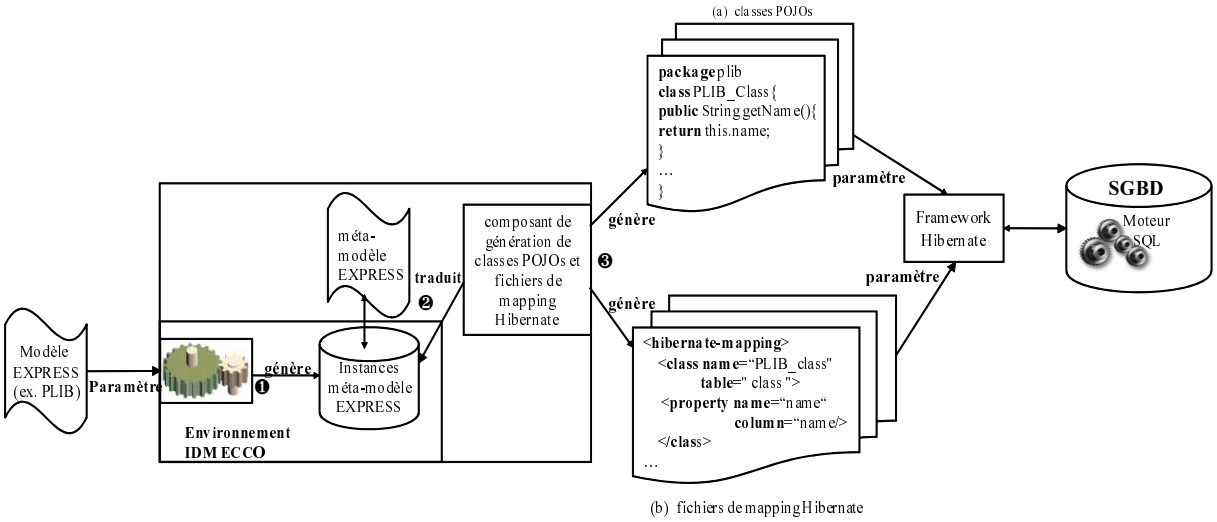


FIG. 4.17 – Architecture du composant de génération de classes POJOs et mapping XML de l'API *API Hibernate*

Algorithme 23 Quelques fonctions de l'API extension.

Gestion de l'extension d'une classe.

- **create_extension** : CLASS -> boolean // crée l'extension d'une classe,
- **drop_extension** : CLASS -> boolean // supprime l'extension d'une classe,
- **addProperty** : CLASS X PROPERTY -> boolean // ajoute une propriété à l'extension d'une classe,
- **dropProperty** : CLASS X PROPERTY -> boolean // supprime une propriété à l'extension d'une classe,

Gestion des instances.

- **create_instance** : CLASS -> oid // crée une instance dans l'extension d'une classe. Toutes les valeurs des propriétés sont nulles,
- **drop_instance** : CLASS X oid // supprime une instance de l'extension d'une classe,
- **getProperty** : CLASS X PROPERTY X oid -> VALUE // renvoie la valeur de la propriété d'une instance,
- **setProperty** : CLASS X PROPERTY X oid X VALUE -> boolean // initialise la valeur de la propriété d'une instance,
- **initProperty** : CLASS X PROPERTY X oid -> boolean // initialise la valeur de la propriété d'une instance à NULL,

Algorithme 24 Exemple de deux fonctions de l'*API Extension* faisant appel à l'API Java.

```
public class API_Extension{
 ...
 //fonction qui permet de créer une extension d'une classe
 public static boolean create_Extension(int UneClasse){//
  if(ExistDefinition(UneClasse)){// s'il existe une définition de la classe
   if(not existExtension(UneClasse)){// s'il n'existe pas encore d'extension dans la base de données
    // créer un instance de class_extension
    int ClassExtension = API_JAVA.create_instance("Class_Extension");
     // on fait pointer la class_extension vers sa classe correspondante
    API_JAVA.set_value_attribute(classExtension,"Class_Extension" ,
                                 "class_definition", UneClasse);
     // on initialise les attributs de type collection properties et instance_identification à vide.
    API_JAVA.reset_value_attribute(classExtension,"Class_Extension" ,
                                   "properties");
    API_JAVA.reset_value_attribute(classExtension,"Class_Extension" ,
                                   "instance_identification");
    ...
   }else{
    Debug.error("CREATE_EXTENSION : IL EXISTE DEJA UNE CLASS_EXTENSION");
    return false;
   }
 }else{
  Debug.error("CREATE_EXTENSION : IL N'EXISTE PAS DE DEFINITION");
  return false;
  }
 }
 ...
 fonction qui permet d'ajouter une propriété à une extension de classe
 public static boolean addProperty(int UneClasse, int UnePropriété){//
  if(existExtension(UneClasse) && ExistDefinition(UneClasse)){
   int ClassExtension = getClassExtension(UneClasse);
   if(ExistDefinition(UnePropriété)){// s'il existe une définition de la propriété
    API_JAVA.insert_value_attribute(classExtension,"Class_Extension" ,
                                    "properties", UnePropriété);
   }else{
    Debug.error("ADDPROPERTY : IL N'EXISTE PAS DE DEFINITION");
    return false;
   }
 }else{
  Debug.error("ADDPROPERTY : IL N'EXISTE PAS UNE CLASS_EXTENSION");
  return false;
 }
 return true;
}
```

4.1.3 Bilan

L'ensemble des APIs que nous avons développées et les relations entre elles, est résumé dans la figure 4.18. On peut remarquer sur la figure les différents niveaux d'APIs. L'*API Java* est basée entièrement sur l'*API PostgreSQL*. L'*API Extension* destinée à accéder aux instances des ontologies, utilise l'API Java pour accéder aux concepts des ontologies et pour établir la correspondance entre ontologie et données. L'API à liaison préalable *API PLIB* est également basée entièrement sur l'*API Java*. L'*API Hibernate* utilise partiellement l'*API Java* pour la gestion de la suppression.

Les différentes APIs que nous venons de décrire constituent les différents points d'accès disponibles pour la gestion d'une BDBO de type OntoDB. Elles permettent ainsi d'éviter d'accéder directement aux schémas logiques des données. Les applications développées à ce jour sur notre système de gestion de BDBO sont entièrement basées sur ces différents niveaux d'APIs. Pour illustrer cela, nous présentons dans la section suivante l'application graphique PLIBEditor (section 4.2) qui offre une interface ergonomique pour la gestion des ontologies et données à base ontologique stockées dans une BDBO.

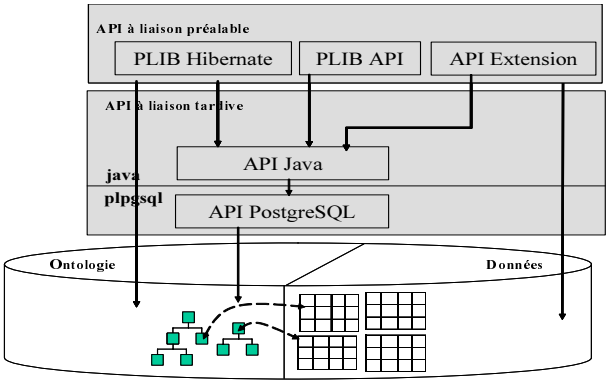


FIG. 4.18 – Architecture des différentes APIs d'accès aux ontologies et aux données à base ontologique.

4.2 Application graphique d'accès : PLIBEditor

Nous présentons l'application *PLIBEditor* implémentée sur le prototype du système de gestion de BDBO que nous avons développé pour la gestion des ontologies et leurs données. *PLIBEditor* est une application graphique implémentée en Java/Swing. *PLIBEditor* est entièrement basé sur les APIs que nous avons décrites précédemment.

Les bases de données usuelles sont définies pour représenter uniquement les données et leur schéma logique, ceci fait que les accès graphiques ergonomiques nécessitent de *coder dans les applications d'accès tout ou partie du modèle conceptuel* spécifique de la base de données. Les BDBOs contiennent non seulement les données mais également leur modèle conceptuel et ontologie. En d'autres termes, elles permettent de représenter explicitement à la fois la structure des données et toute la sémantique des données. Cette faculté des BDBOs offre la possibilité

de programmer une interface *générique* (indépendant du contenu de la base de données) et qui permet néanmoins *l'accès au niveau connaissance* aux données contenues. L'un des buts visés par *PLIBEditor* est justement d'illustrer cette capacité très particulière des BDBOs.

Les principales fonctions offertes par *PLIBEditor* sont :
- gestion d'ontologies,
- gestion des données à base ontologique,
- importation des ontologies et données à base ontologique,
- exportation des ontologies et données à base ontologique,
- exécution de requêtes sur les ontologies et les données à base ontologique.

4.2.1 Gestion des ontologies avec PLIBEditor

Cette fonction PLIBEditor permet la création d'ontologies et l'édition, création, modification et suppression des concepts (source d'information, classe, propriété, type) constituant les ontologies dans une BDBO. L'interface graphique offerte, développées spécifiquement pour la gestion des ontologies PLIB est adaptable facilement à un autre modèle d'ontologie. PLIBEditor est basé sur les APIs : l'API Java et l'API PLIB pour la gestion des ontologies dans la base de données.

La figure 4.19 présente la fenêtre principale de *PLIBEditor*. Celle-ci permet aussi bien de créer et de visualiser la description des classes (cf. zone 1) que des propriétés (cf. zone 2) des ontologies. L'arborescence de la zone 3 présente la hiérarchie des classes stockées dans la BDBO. La description des classes sélectionnées dans la zone 3 est visualisée dans la zone 1. Dans les zones 1a et 1b sont respectivement affichées les propriétés applicables et les propriétés visibles des classes. La sélection d'une propriété d'une de ces deux listes affiche sa description dans la zone 2.

4.2.2 Gestion des instances avec PLIBEditor

Cette fonction de PLIBEditor permet :
- l'édition, la consultation, la modification et la suppression des instances des classes d'ontologies dans la base de données, et,
- la création du schéma de données des instances des classes, ainsi que la manipulation (édition, consultation, modification et suppression) des instances.

La figure 4.20 présente les fenêtres de PLIBEditor permettant d'accomplir ces différentes tâches. Les instances des classes sont affichées sous forme d'une table (cf. zone 1 sur la figure 4.20). Les colonnes de la table sont constituées des propriétés utilisées de la classe. La gestion de la structure des instances est réalisée à partir de la fenêtre imbriquées (zone 2 sur la figure 4.20). Dans la partie gauche (zone 2a), sont listées toutes les propriétés de la classe qui peuvent potentiellement être utilisées par les instances des classes. La sélection d'une propriété dans cette liste, permet de l'ajouter dans la table de la classe courante. Celle-ci apparaîtra dans la liste des propriétés effectivement utilisées par les instances (zone 2b). Les cases à cocher dans la zone 2b permettent de sélectionner les propriétés qui formeront la clé des instances de la classe.

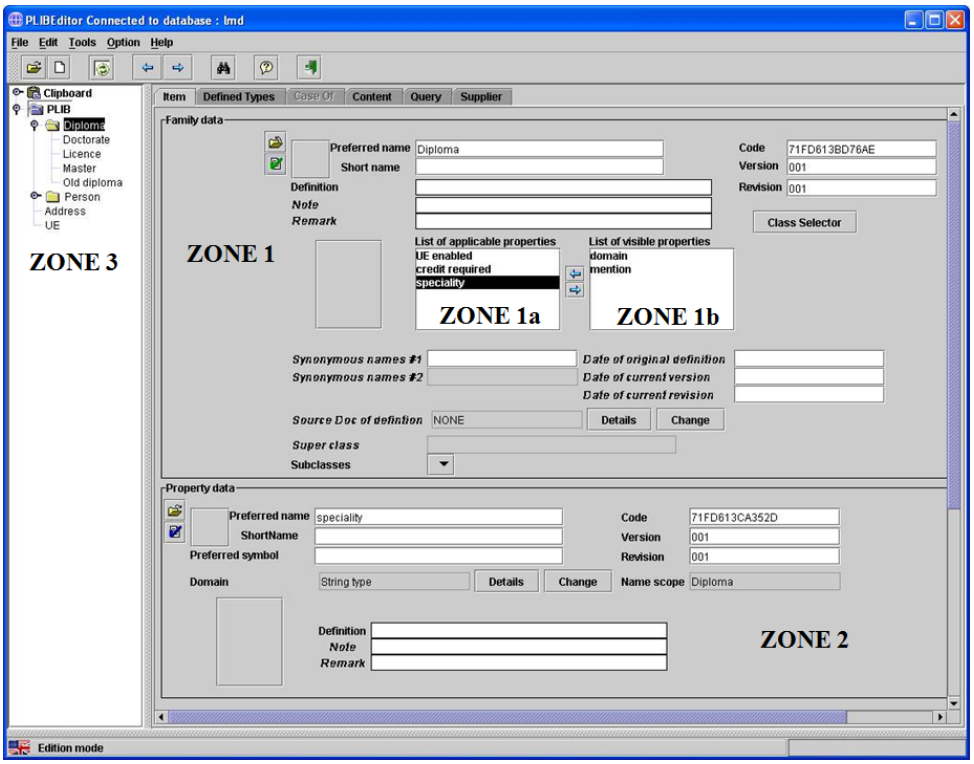


FIG. 4.19 – Gestion des concepts des ontologies dans PLIBEditor

Cette fonction de PLIBEditor se base sur *API Extension* pour réaliser toutes les tâches ci-dessus.

4.2.3 Importation d'ontologies dans la base de données

Cette fonction de PLIBEditor sert à charger des ontologies et/ou des données à base ontologique PLIB représentées dans le format de fichier physique PLIB dans une BDBO. Cette fonction de PLIBEditor fait appel au composant d'importation des ontologies présenté dans la section 1.3. La figure 4.21a montre la capture d'écran de la fenêtre de PLIBEditor permettant l'importation d'ontologies dans la base de données. Cette fonction offre différentes options :

- Importation uniquement les concepts des ontologies ou soit uniquement les instances des concepts et les deux à la fois (cf. zone 1 de la figure 4.21).
- Vérification avant importation dans la BDBO, de la cohérence syntaxique et sémantique des ontologies et données à base ontologique. Ceci en vue d'éviter l'importation dans la BDBO des données erronées (cf. zone 2 de la figure 4.21).
- Importation des données dans une transaction. Cette option permet d'éviter de rendre la BDBO *incohérente* lors du changement des fichiers. Les requêtes SQL, générées pour le peuplement des tables, sont exécutées dans une transaction SQL. En cas d'erreurs, la transaction sera annulée et l'importation est suspendue. Si cette option n'est pas activée, les erreurs qui peuvent survenir ne sont pas prises en compte (cf. zone 2 de la figure 4.21).
- Résolution de façon complètement automatique des références entre ontologie et intégra-

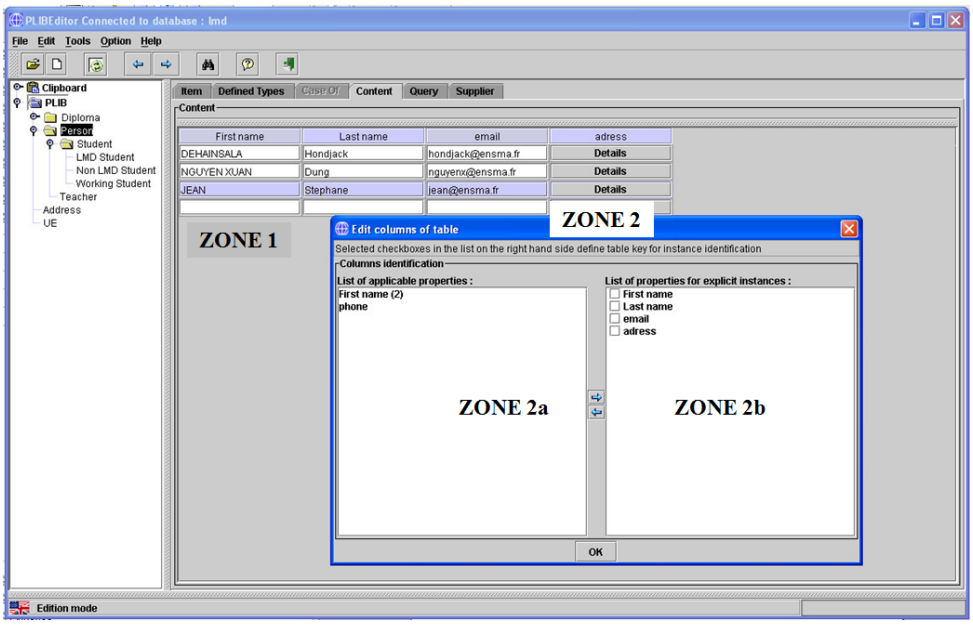


FIG. 4.20 – Gestion des instances des ontologies dans *PLIBEditor*.

tion automatique de l'ensemble des ontologies et les données déjà chargées.

La fenêtre de la figure 4.21b présente les différentes étapes du processus d'importation d'une ontologie dans une BDBO.

4.2.4 Exportation d'ontologies dans la base de données

La fonction d'exportation d'ontologies et de données à base ontologique est réalisée à travers la fenêtre de la figure 4.22. Cette fonction se base sur le composant d'extraction que nous avons implémenté et présenté dans la section 1.4. Cette fonction de PLIBEditor offre une interface pour l'extraction d'un sous-ensemble de concepts (i.e., classes et propriétés) de l'ontologie et/ou de sous-ensembles d'instances de ces concepts. Différents options d'extraction qui sont offertes :

- extraction ou non des éventuelles classes subsumantes par la relation de *case_of* avec les classes à extraire ;
- extraction de la BDBO des classes avec leurs propriétés *applicables*, *visibles*, ou *utilisées* par leurs instances ;
- extraction de la BDBO uniquement d'un sous-ensemble de l'ontologie ou uniquement de données à base ontologique ou les deux en même temps ;
- enfin, extraction de la BDBO des fichiers externes associés à la définition des classes et propriétés ou des valeurs de propriétés de type *external_type* (i.e., constituées d'un fichier externe).

La zone centrale (zone 1) de la fenêtre sert à sélectionner les classes qu'on souhaite extraire de la BDBO. Par défaut, la sélection d'une classe sélectionne également ses super-classes jusqu'à la racine. On peut néanmoins après cela décocher certaines cases.

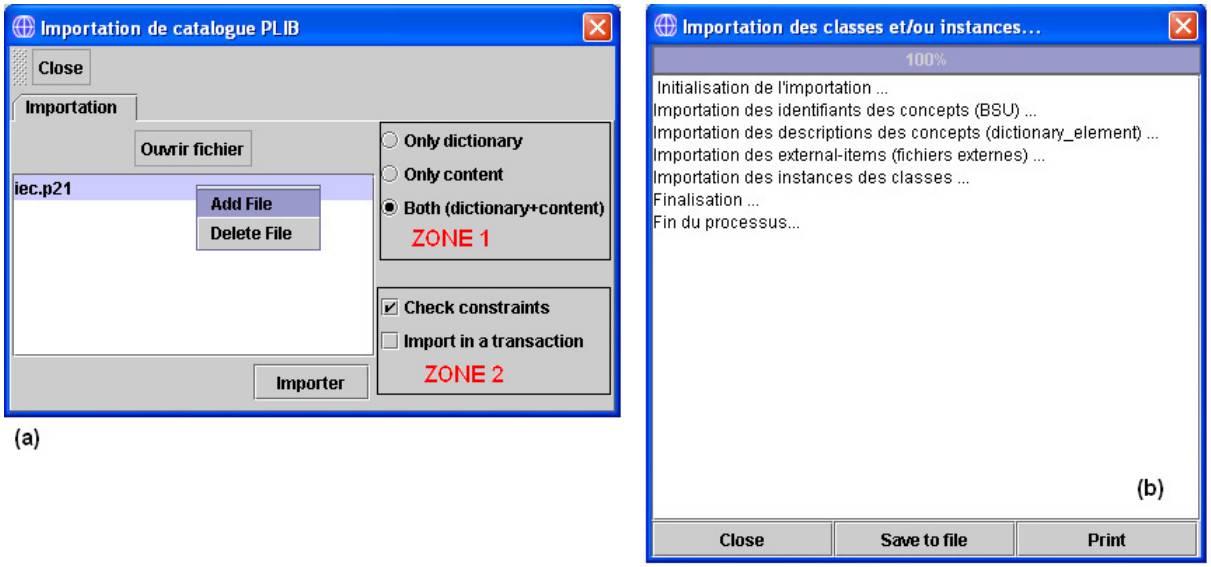


FIG. 4.21 – Importation des classes et des propriétés dans PLIBEditor.

Pour chaque classe, leurs instances peuvent être sélectionnées dans la partie inférieure (zone 2) de la fenêtre pour être extraites. On peut également sélectionner toutes les instances.

4.2.5 Exécution de requêtes OntoQL avec PLIBEditor

A l'image des systèmes de bases de données relationnelles ou objets qui offrent des langages déclaratifs (SQL et OQL) qui permettent la définition des structures des données (DDL - Data Definition Language) et l'interrogation des données (DML - Data Manipulation Language), un travail de thèse est mené en ce moment au sein de notre équipe pour la spécification d'un langage de requêtes pour les bases de données à bases ontologique. Ce langage, nommé OntoQL (**Onto**logy **Q**uery **L**anguage), permet à la fois d'interroger les ontologies, les données à base ontologique et/ou les deux simultanément [86, 87, 84]. Ce langage fait l'objet d'une thèse en cours au laboratoire.

PLIBEditor offre également une interface QBE (cf. figure 4.23) pour l'interrogation des instances des classes d'ontologies implémentée. Celle-ci permet de cacher la complexité de la syntaxe du langage OntoQL ainsi que la structure de table pour ne parler que d'extension de classe. Notons que cette interface permet de faire des requêtes *uniquement* sur les instances des classes. L'interrogation des ontologies pourra se faire avec l'outil *OntoQL Plus** en cours implémentation.

Dans la zone 1, sont affichées les propriétés intervenantes dans la requête conçut par l'interface. Dans la zone 2, nous avons la requête *OntoQL* générée par l'interface QBE (zone 1). Dans la zone 3, nous avons la traduction en SQL de la requête *OntoQL*.

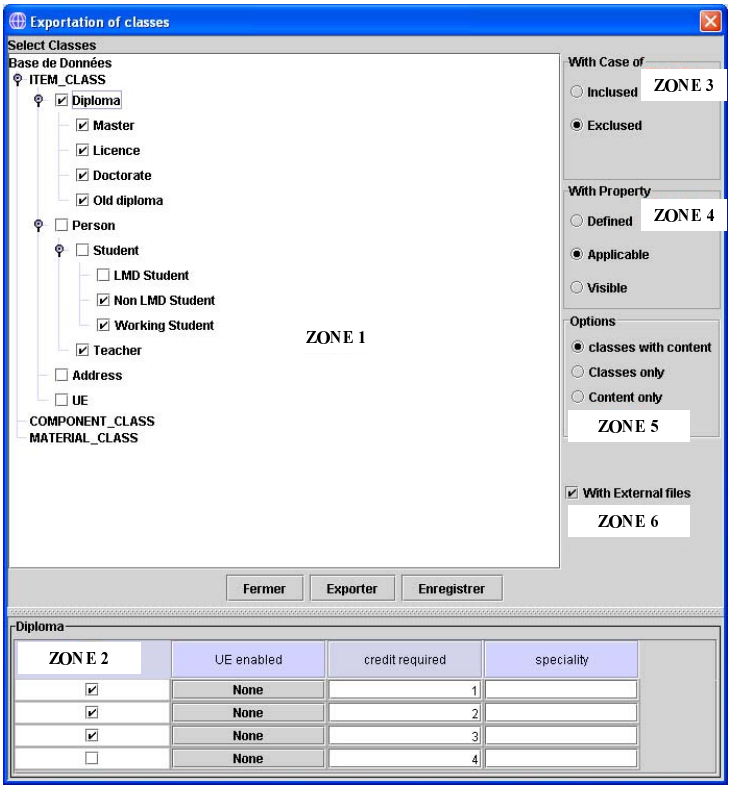


FIG. 4.22 – Exportation des classes et des propriétés dans *PLIBEditor*

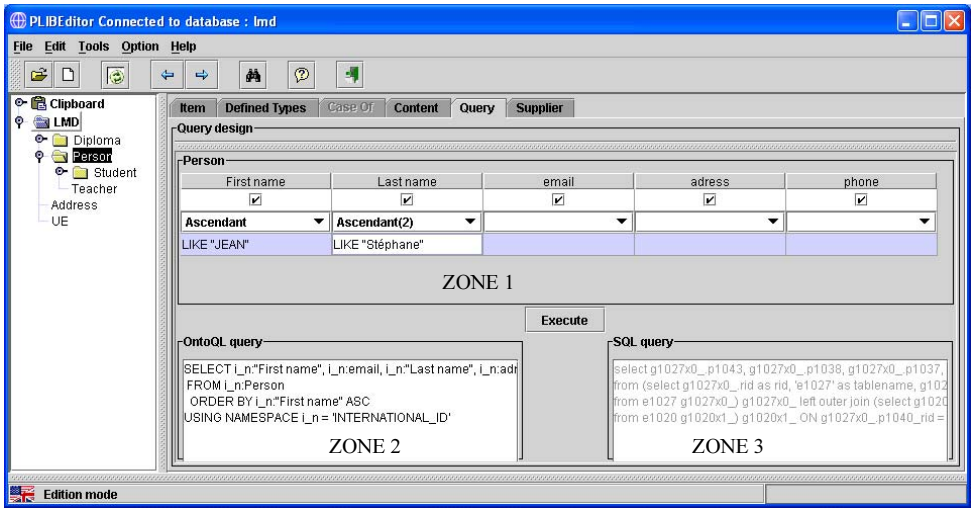


FIG. 4.23 – Interface QBE pour l'interrogation des instances des classes.

4.3 Récapitulatif de l'implémentation de notre prototype du modèle d'architecture OntoDB

Nous avons montré dans la section précédente, les différentes APIs d'accès que nous avons conçues et développées pour la gestion des ontologies et des données à base ontologique stockées dans une BDBO. Ces différentes APIs constituent le noyau fonctionnel de toutes les applications exploitant une BDBO. En effet, toutes les applications implémentées à ce jour au sein de notre équipe, à savoir PLIBEditor, le moteur de requête OntoQL, OntoQL Plus* [84] et OntoWEB (un éditeur de BDBO en mode client-serveur, développé par une autre chercheuse du laboratoiree), se basent entièrement sur elles. La figure 4.24 présente justement comment toutes ses applications utilisent ces APIs.

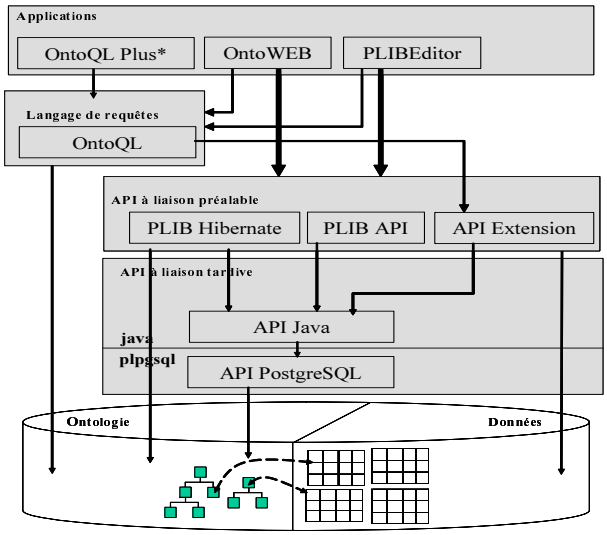


FIG. 4.24 – Utilisation des APIs et langage par les applications.

Dans la figure 4.25, nous présentons, de façon synthétique, l'ensemble des composants que nous avons implémenté pour la mise en œuvre de notre système de gestion de base de données à base ontologique. Chacun de ces composants a été programmé pour permettre à notre système de s'adapter aux évolutions possibles du modèle d'ontologie et/ou au changement du modèle d'ontologie utilisé conformément à l'objectif O_2 que nous nous fixé dans le chapitre 3. Nous avons systématiquement utilisé des techniques d'ingénierie dirigées par les modèles. Une exception existe néanmoins au niveau de la mise en œuvre de la partie *ontologie* (composant "*Définition (manuelle) des attributs dérivés et des contraintes*") dans lequel la connaissance procédurale (attributs dérivés et contraintes (rule local, global, etc.)) du modèle d'ontologie a été codée manuellement par le programmeur. Cela se justifie par le fait que bon nombre des fonctions définissant les attributs dérivés et les contraintes du modèle d'ontologie PLIB ont été définis dans un but de *spécification*, sans *aucune préoccupation* d'optimisation. La traduction automatique de celles-ci dans le langage du SGBD aurait forcement causé de mauvaises performances du système. Il était donc préférable de les ré-écrire à la main.

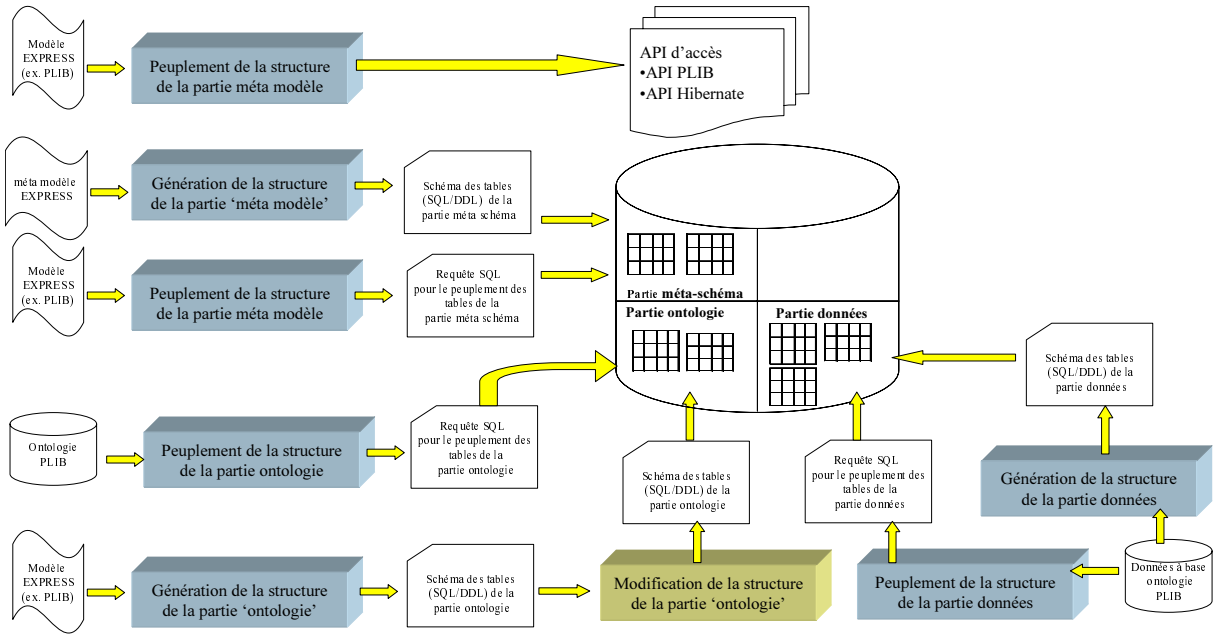


FIG. 4.25 – Architecture des composants du prototype du modèle d'architecture OntoDB

Nous synthétisons dans la table 4.1 les fonctionnalités notre prototype de système de gestion de BDBO comparés aux autres systèmes existants dans la littérature. On peut remarquer les fonctionnalités supplémentaires apportées par notre système :

- **Sécurité de données** : Notre approche de représentation des données à base ontologique dans des tables spécifiques associées à leur classe et dont les colonnes sont de ces propriétés des classes offre la possibilité d'utiliser tous les mécanismes de bases des SGBDs cibles pour assurer la sécurité des données : confidentialité, concurrences d'accès, droit d'accès, etc. Ce qui apparaît impossible dans les approches par *table par propriété*, ou par *triplets*.

- **Gestion des versions et l'évolution des ontologies** : Notre prototype permet, selon les besoins des utilisateurs, de gérer les versions des concepts des ontologies et d'assurer leurs évolutions dans la base de données de façon cohérente malgré la possibilité d'évolution asynchrone des sources de données intégrées. C'est l'une des fonctionnalités n'existant dans aucun des autres systèmes de gestion de BDBOs implémentées à ce jour.

- **Cycle de vie des instances** : Comme la fonctionnalité précédente, notre système permet de versionner également les instances des classes des ontologies pour pouvoir assurer leur cycle de vie, c'est-à-dire, de pouvoir tracer toutes modifications sur sa structure afin de suivre les évolutions des instances pendant toute leur existence dans les différences versions des classes.

Caractéristiques	Sésame	RDFSuite	DLDB	OntoDB
Respect d'un standard	oui	oui	oui	oui (PLIB)
Échange de données	oui	oui	oui	oui
Support linguistiques	oui	oui	oui	oui
Langage de requêtes	oui	oui	oui	oui (OntoQL)
Extraction d'ontologies	oui	oui	oui	oui
Sécurité des données	non	non	non	oui
Gestion des versions et évolutions des ontologies	non	non	non	oui
Cycle de vie des instances	non	non	non	oui

TAB. 4.1 – Comparaison d'OntoDB avec les systèmes de gestion de BDBOs existants dans la littérature

5 Conclusion

Nous avons présenté dans ce chapitre, l'implémentation d'un système de base de données à base ontologique mettant en œuvre l'architecture de BDBO *OntoDB*. L'objectif de cette implémentation était de :

- montrer la faisabilité,
- mettre en évidence les services susceptibles d'être rendus à l'utilisateur du point de vue accès aux données au niveau connaissance,
- évaluer l'efficacité du système de base de données résultat.

Une première difficulté de notre implémentation était de traiter un modèle d'ontologie très complexe (218 entités) et évolutif. Nous avons montré que les techniques d'IDM permettant effectivement le passage à l'échelle sur des modèles de grande taille et qu'elles permettraient également de s'adapter sans trop d'effort aux évolutions. L'IDM en EXPRESS nous a permis d'implémenter des programmes génériques (1) pour la génération de la structure des tables des parties *méta-schéma* et *ontologies*, (2) pour la lecture des ontologies et du modèle d'ontologie respectivement dans la partie *ontologie* et la partie *méta-schéma*, et (3) pour la génération d'interface de programmation de deux APIs à liaison préalable (*API PLIB* et *API Hibernate*).

Une deuxième difficulté était de traiter des ontologies elles-mêmes de grande taille. Ceci nous a mené à étudier et à implanter différentes optimisations qui ont abouti à un fonctionnement acceptable même avec des ontologies de taille très significative correspondant à des ontologies réelles. L'approche *OntoDB* s'avère ainsi susceptible de supporter le passage à l'échelle d'applications industrielles réelles, ce qui n'est pas le cas, d'après nos essais, pour les autres implémentations existantes.

Concernant les services susceptibles d'être rendus, nous avons identifié, puis spécifié, les différents niveaux d'APIs nécessaires pour permettre de développer des programmes de navigation dans les ontologies et des données à base ontologique. Nous avons ensuite implémenté une interface graphique permettant un utilisateur de réaliser graphiquement et dans sa propre langue (si

les ontologies sont traduites) toutes les actions d'exploitation, de modification et d'échange de tout ou partie du contenu de la BDBO. Aucune de ces actions ne sont usuellement possible à un *utilisateur final* dans une base de données ordinaire.

Ce développement nous a amené à faire un certain nombre de choix que l'on peut maintenant remettre en perspectives.

- Notre développement s'est appuyé de façon spécifique sur le SGBD PostgreSQL. Les SGBD relationnels-objets étant tous très particuliers dans leurs implantations partielles et spécifiques de la norme SQL99, il serait intéressant d'étudier la migration de notre prototype sur d'autres plateforme comme Oracle, Sql server.

- Même si notre approche est très générique par rapport au modèle d'ontologie, en fait, certains développements dont l'éditeur graphique PLIBEditor, se sont appuyés sur diverses spécificités du modèle PLIB. Il serait intéressant s'isoler un méta-modèle commun à tous les modèles d'ontologies puis d'offrir des mécanismes permettant de spécialiser ce modèle. Ceci a été fait en particulier au niveau graphique par l'éditeur protégé [61]. Mais celui-ci travaille exclusivement en mémoire centrale.

- Notre implémentation a exploité le modèle PLIB tel qu'il est définit dans sa norme. Vue la complexité de ce modèle, il serait intéressant d'étudier comment ce modèle pourrait être simplifié en conservant son pouvoir d'expression.

- Enfin d'autres systèmes de BDBO ont également été développés parallèlement à notre travail par d'autres chercheurs, il serait intéressant de comparer les performances. Ceci n'est guère possible du point de vue accès aux ontologies puisque les modèles d'ontologies et les ontologies elles-mêmes sont différents et que nous n'avons pas encore écrit de convertisseurs de formats d'ontologies. C'est par contre possible du point de vue données à base ontologique ce qui a été fait dans le chapitre suivant.

Troisième partie

Évaluation des performances

Chapitre 5

Évaluation des performances du modèle d'architecture OntoDB

Introduction

Dans ce chapitre nous évaluons la performance de notre approche de représentation des données à base ontologique et les ontologies au sein du prototype du système de gestion de BDBO présenté dans le chapitre précédent par rapport aux approches existantes. Dans le chapitre 2, nous avons mentionné qu'il existait dans la littérature essentiellement trois autres approches de représentation.

1. La représentation sous forme de *triplets*, également appelée modèle *vertical* dans [5], dans laquelle toutes les données décrivant les propriétés des instances de classe sont stockées dans une unique table à trois colonnes : *sujet*, *prédicat*, *objet*. Ces trois colonnes représentent respectivement l'identifiant des objets, l'identifiant de la propriété valuée et la valeur de la propriété [5, 23, 31, 165].
2. La représentation par *table par propriété*, également appelée *binaire* [5] ou *décomposition* [42], dans laquelle une table spécifique est définie pour chaque propriété. Chacune des tables de propriété est constituée de deux colonnes (*id*,*value*). *id* est l'identifiant de l'objet et *value* pour la valeur de la propriété [6, 24, 30, 42, 58, 121].
3. La représentation dans une unique table dite table *universelle*, dans laquelle toutes les propriétés des classes sont définies comme colonnes d'une unique table [5, 159].

Nous avons proposé une représentation alternative qui consiste à créer une relation pour chaque classe concrète dont les attributs correspondent aux propriétés effectivement valuées pour au moins une instance de la classe [127]. En absence de contraintes de normalisation, nous avons également proposé de représenter cette relation sous forme d'une table unique.

Notre évaluation de performance est réalisée selon le critère de temps de réponse d'exécution d'un ensemble de requêtes types. Notons que l'évaluation de la représentation des ontologies stockées dans la partie *ontologie* de notre architecture peut plus difficilement être effectuée de façon comparative. En effet, la structure de représentation des ontologies que nous avons définie

vise à permettre de représenter des ontologies PLIB, beaucoup plus complètes et complexes que les ontologies usuelles de type RDFS ou OWL, qui sont les seules gérées par les systèmes avec lesquels des comparaisons étaient possibles. Notre évaluation de cet aspect est donc beaucoup plus partielle. Elle vise seulement à vérifier la faisabilité de notre approche en mesurant le temps de réponse des requêtes de navigation dans les ontologies d'OntoDB pour des BDBOs de tailles variables. L'objectif de cette évaluation est de vérifier le comportement de notre prototype et d'étudier son passage à l'échelle pour des ontologies réelles de taille importante.

Le chapitre s'organise en deux parties. Dans la première partie, la plus importante, nous procédons à l'évaluation de notre approche de représentation des données à base ontologie par rapport aux approches existantes en décrivant à la fois les tests réalisés et leurs résultats. Dans la deuxième partie, nous faisons l'évaluation de notre approche de représentation des ontologies en calculant le temps de navigation dans les concepts (classes et propriétés) des ontologies d'OntoDB lorsqu'on les parcours avec l'éditeur PLIBEditor.

1 Évaluation des performances de la partie données

Avant de présenter les résultats des tests réalisés, nous présentons les composantes du banc d'essai que nous avons utilisé : (1) la configuration de la machine des tests, (2) le serveur de bases de données, (3) le générateur d'instances des classes dans les bases de données (4) la structure des schémas de données selon les approches de représentation existantes et (5) les types de requêtes que nous avons considérés pour nos tests sur les différentes approches de représentation.

1.1 Description du banc d'essai

1.1.1 Présentation générale

Comme pour les bancs d'essai antérieurs réalisés sur le sujet [67, 69], nous avons utilisé une approche de génération d'instances de classe d'ontologies ce qui permettait de générer des contenus de taille et de structures diverses.

Par contre nous n'avons pas utilisé de générateurs existants, et ce pour deux raisons. D'abord la plupart des bancs d'essai existants [22, 67, 69] utilisent des ontologies RDF Schéma, DAML+OIL ou OWL et visent tout particulièrement à faire des inférences et à mesurer le degré de complétude des inférences. Compte tenu du fait que nous souhaitons mesurer les performances d'exécution de requêtes et que notre prototype est basé sur le modèle d'ontologie PLIB, l'utilisation de ces bancs d'essai n'était guère adaptée. La deuxième raison est que ceux-ci portent souvent sur des jeux de tests simples sans aucun lien avec la réalité. Or, un de nos domaines cibles est celui de la gestion des composants industriels. Dans ce domaine précisément, il existe des ontologies qui sont à la fois *réelles* et de *grande taille* que nous avons utilisées pour évaluer nos travaux.

Ce sont les deux raisons qui nous ont incitées à utiliser des ontologies PLIB et à programmer notre propre générateur. De plus, définir notre propre générateur, nous a permis de contrôler finement le volume de données générées et la structure des données manipulées. Les bases de

données que nous manipulons sont caractérisées par :

1. un nombre variable de classes,
2. un nombre variable de propriétés par classe,
3. un nombre variable d'instances par classe.

Ceci permettra de couvrir des domaines d'application divers caractérisés par le fait, d'une part, qu'ils manipulent des ontologies ayant des structures diverses, et, d'autre part, que les populations d'instances sont elles aussi diverses. L'avantage de notre approche est de nous permettre d'utiliser une ontologie *réaliste* : l'ontologie de référence *IEC 61360-4* qui décrit l'ensemble des composants électroniques ainsi que leurs propriétés caractéristiques. Elle est normalisée au niveau international à l'IEC[19] (International Electronic Commision).

Dans la littérature, plusieurs critères d'évaluation des approches de représentation ont été proposées [157]. On peut ainsi citer : (1) le temps de chargement des bases de données, (2) la taille des bases de données et (3) le temps de réponse d'exécution de requêtes. Dans notre travail, nous évaluons les performances de ces approches selon le critère le temps de réponse des requêtes.

Remarquons que les résultats obtenus sur nos ontologies PLIB semblent raisonnablement extrapolables à d'autres domaines. En effet, dans [104], on trouve une étude quantitative d'une trentaine d'ontologies réalistes définies en RDF Schéma dans des domaines très variés (e-commerce, géospatial, éducation, biologie, médecine, etc.). Les auteurs ont constaté que ces constructions non supportées par PLIB ou non utilisées dans nos données de test étaient en fait peu utilisées.

- **Subsomption de propriétés.** Très peu d'ontologies hiérarchisent des propriétés (*SubPropertyOf*). Dans les rares cas où cela est utilisé, c'est dans un but très limité afin d'établir une relation sémantique (composition ou agrégation) entre des classes et non dans un but de caractérisation d'instances.
- **La multi-instanciation.** La multi-instanciation n'est pas utilisée dans la majorité des ontologies[20].
- **Co-domaine.** Les types collections (Sequence, Bag, Alternative) non utilisées dans nos données de test ne sont pas non plus utilisés dans la plupart des ontologies étudiées.

D'après cette étude, il semble donc que bon nombre d'ontologies définies dans le cadre du Web Sémantique pourraient être représentées dans *OntoDB* sans perte d'information et que les résultats de nos essais leur seraient également applicables.

1.1.2 Choix des approches alternatives à évaluer

Les approches *table par propriété*, *triplets* et *table universelle* ont déjà fait l'objet d'une évaluation dans d'autres travaux [5, 24, 42, 58, 156, 157]. En particulier, Agrawal et al. [5] ont montré, sur un domaine similaire au notre à savoir celui du commerce électronique que les approches de représentation par *table par propriété* et *triplets* étaient beaucoup plus performantes

[19] `http://dom2.iec.ch/iec61360?OpenFrameset`

[20] Ils précisent toutefois dans leur article qu'ils n'ont pas trouvé une quantité significative de population d'instances d'ontologies.

que l'approche par *table universelle*. Compte tenue de la complétude de ces tests, il nous parait inutile de nous intéresser à nouveau à la *table universelle*.

Theoharis et al. [157] ont montré, en testant différent types de requêtes que l'approche *table par propriété* était plus performante que l'approche *triplets*. Nous confirmerons ce fait sur quelques tests, puis nous nous concentrerons donc sur la comparaison *table par classe* d'une part, et *table par propriété*, d'autre part, puisqu'il s'agit de la plus efficace des approches préexistantes.

1.1.3 Description du générateur d'instances

Notre générateur se base donc sur une ontologie PLIB réelle : l'ontologie IEC 61360-4 :1998. Cette ontologie décrit les composants usuels dans le domaine de la conception électronique ainsi que toutes les propriétés techniques qui les caractérisent. IEC 61360-4 est constituée de 190 classes et 1026 propriétés. La profondeur de la hiérarchie de classes est en moyenne de cinq classes et elle comporte 134 classes feuilles. Pour contrôler plus facilement la taille effective des bases de données, nous avons modifié tous les co-domaines des propriétés de l'ontologie en des chaînes de 255 caractères (soit 255 octets).

Nous avons programmé un générateur (cf. figure 5.1) qui permet de créer des populations d'instances pour les classes de l'ontologie de IEC 61360-4 représentée dans la base de données. Celui-ci est programmé indépendamment d'une ontologie particulière. Il reçoit en entrées une ontologie PLIB quelconque et trois autres paramètres de configuration de la base de données à générer : (1) le nombre de propriétés (NP) à initialiser pour chaque classe, (2) le nombre d'instances (NI) à générer, et (3) le type de représentation choisi pour les instances : table par propriété, triplets ou table par classe. La génération d'instances se réalise en deux étapes :

(1) Une première étape construit le modèle conceptuel (MC) de la base de données, puis génère les schémas de tables en fonction du type de représentation choisi. Pour chaque classe, *NP* propriétés sont sélectionnées de façon aléatoire dans les propriétés applicables de chaque classe.

(2) La deuxième étape génère les *NI* instances de la table de classes.

Nous avons généré différentes bases de données en faisant varier le nombre de propriétés (NP) et le nombre d'instances (NI) par classe. dans le reste du chapitre, nous noterons par :

- *TC* : l'abréviation de l'approche *Table par Classe*,
- *TP* : l'abréviation de l'approche *Table par Propriété*,
- BD_<NombreInstances>K_<NombreProprietes>P : un contenu de base de données générée contenant *NombreInstance*K instances par classe et *NombrePropriété* propriétés pour chacune de ces classes. Par exemple : BD_10K_10P est un contenu de base de données ayant 10K instances et 10 propriétés par classe et qui sera générée pour chacun des types de représentation d'instances que nous souhaitons évaluer.

Comme cela apparaît dans les figures 5.2 et 5.3, nous avons créé 6 populations d'instances différentes se divisant en deux séries.

1. **Serie 1** : La première série (cf. figure 5.2) nous permet d'étudier le passage à l'échelle des

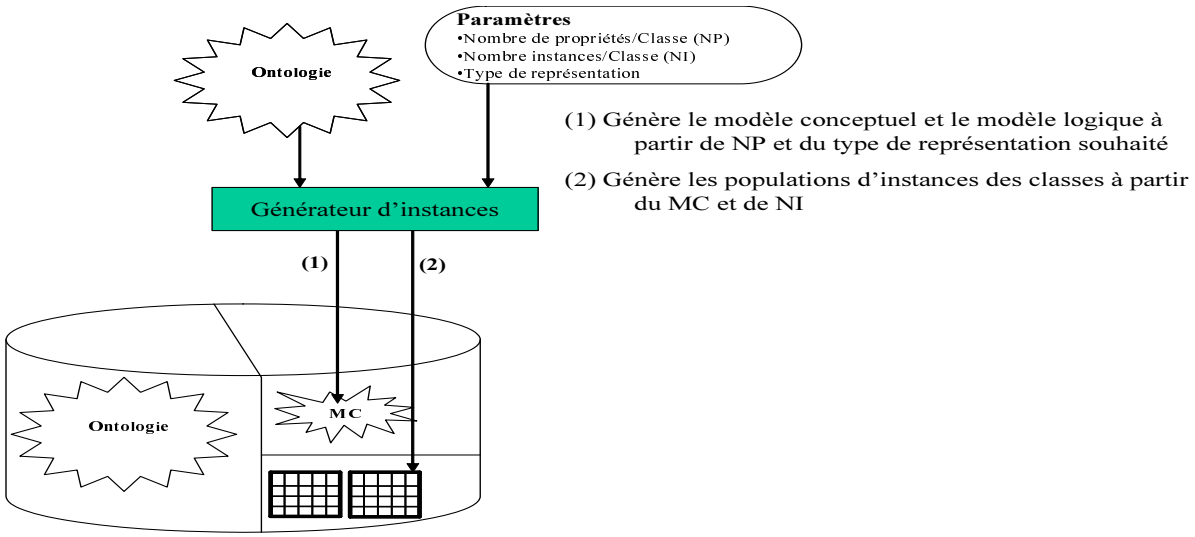


FIG. 5.1 – Générateur d'instances du banc d'essai

différentes représentations. Nous avons les bases de données : BD_1K_10P, BD_1K_25P et BD_1K_50P. Ces bases de données sont caractérisées par un nombre fixe d'instances (1K) et avec un nombre croissant de propriétés dans chaque table de classe.

	BD_1K_10P	BD_1K_25P	BD_1K_50P
Nombre de propriétés par classe feuille (NP)	10	25	50
Nombre d'instances par classe feuille (NI)	1K	1K	1K
Nombre de classe feuille	134	134	134
Nombre total d'instances dans la BD	134K	268K	134K
Volume des bases de données	341 MO	0,84GO	1,68GO

FIG. 5.2 – Caractéristiques des bases de données du banc d'essai pour les tests de passage à l'échelle

2. **Serie 2** : La deuxième série (cf. figure 5.3) nous permet d'étudier l'impact de la variation du nombre de propriétés (NP) et du nombre d'instances (NI) des tables des classes, tel que le produit $NP \times NI$ est constant. Nous avons les bases de données : BD_10K_10P, BD_4K_25P et BD_2K_50P. Le nombre d'instances varie entre 100.000 et 1,5 millions environ.

1.1.4 Configuration de la machine des tests

Nous décrivons le serveur et la machine de nos tests :

1. Serveur de base de données
   - PostgreSQL-7.4 émulé sur cygwin[21]
   - taille du buffer : 50MO

[21] http ://www.cygwin.com/

	BD_10K_10P	BD_4K_25P	BD_2K_50P
Nombre de propriétés par classe feuille (NP)	10	25	50
Nombre d'instances par classe feuille (NI)	10K	4K	2K
Nombre de classe feuille	134	134	134
Nombre total d'instances dans la BD	1340K	536K	268K
Volume des bases de données	3,41GO	3,41GO	3,41GO

FIG. 5.3 – Caractéristiques des bases de données du banc d'essai pour étudier la variation des *NI* et *NP*

2. Machine du serveur
   - DELL SERVER P2600
   - Système d'exploitation Windows 2003
   - 6 GO de RAM
   - 200 GO de HDD
   - 3,7 Ghz de fréquence pour le processeur pentium 4.

1.1.5 Méthodologie d'exécution des requêtes

Les requêtes sont exécutées via JDBC. Le temps d'exécution de la requête est la différence du temps horloge, en millisecondes, avant l'exécution de la requête et juste après son exécution. Chaque requête est exécutée dix fois. Si t_i est le temps d'exécution à l'itération i, le temps effectif (t) retenu pour l'exécution de la requête est $t = moyenne(t_1, ..., t_n)$.

Dans la section suivante, nous présentons brièvement la structure des tables correspondant à chacune des trois approches évaluées à travers un exemple.

1.2 Structures de la partie (*données* des bases de données des tests)

Pour faciliter la compréhension de ces structures, nous considérons l'exemple suivant. Soit le modèle objet de la figure 5.4 (que nous considérons comme une ontologie) constitué de quatre classes (*Personne*, *Étudiant*, *Salarié* et *Adresse*). *Personne* est super-classe des classes *Étudiant* et *Salarié*. Nous nous baserons sur ce modèle pour illustrer la structure de stockage des objets dans chacune des approches de représentation des instances dans des bases de données à base ontologique.

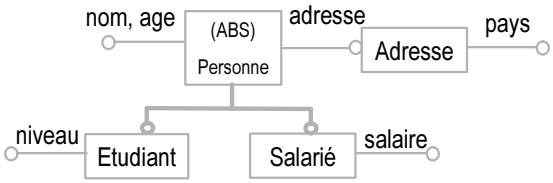


FIG. 5.4 – Exemple de modèle objet pour l'illustration des approches de représentation

1.2.1 Approche de représentation par *table universelle*

Dans cette approche de représentation, une unique table appelée *table universelle* est créée [5, 159]. Les colonnes de cette table sont constituées par l'union des propriétés de toutes les classes de la base de données à base ontologique (cf. figure 5.5).

Personne

ID	nom	age	adresse	niveau	salaire	pays
1	toto	45	5	A2		
2	titi	12		A2		
3	Remi				15	
4	Ali				12	
5						Tchad

FIG. 5.5 – Exemple de représentation par une table universelle

Nous ne testerons pas cette approche car Agrawal et al. [5] ont déjà prouvé qu'elle était généralement moins performante que les approches *table par propriété* et *triplets*.

1.2.2 Approche de représentation *triplets*

Dans l'approche *triplets*, toutes les valeurs des propriétés sont stockées dans une unique table constituée de trois colonnes (sujet, prédicat, objet). La colonne *sujet* permet de représenter l'identifiant des instances, la colonne *prédicat* permet de représenter l'identifiant des propriétés et enfin la colonne *objet* permet de représenter la valeur de la propriété. Concernant le typage des instances, soit la table *triplets* est utilisée, le type de chaque instance étant représenté par un triplet (identifiant instance, *instance-OF*, identifiant classe), soit la population de chaque classe est représentée par une table. Les identifiants des instances sont représentés dans la table associée à leur classe. Ces tables de classes sont constituées d'une seule colonne *id* (cf. figure 5.6). Ainsi, en adoptant la deuxième représentation, l'exemple de la figure 5.4 nécessite les quatre tables : *étudiant*, *personne*, *adresse*, et la grande table *triples*.

Pour optimiser le temps des requêtes, nous avons, comme dans [5, 58, 102], créé des index sur chacune des trois colonnes de la table *triplets*, regroupé la table sur la colonne *prédicat* ("*cluster predicat on triplets*") et défini un index primaire sur la colonne *id* de la table des classes. Le tri de la table *triplets* sur la colonne *prédicat*, comme on peut l'observer sur la figure 5.6, est indispensable pour garder des temps d'accès acceptables pour les bases de données de tailles importantes.

Étudiant

id
1
2

Salarié

id
3
4

Adresse

id
5

Triplets

sujet	predicat	objet
1	nom	toto
3	nom	Remi
4	nom	Ali
1	niveau	A2
3	salaire	15
5	pays	Tchad

FIG. 5.6 – Approche par table triplets

1.2.3 Approche de représentation *table par propriété*

Dans cette approche, implémentée en particulier dans [6, 24, 30, 42, 58, 121], chaque propriété P_i est associée à une table constituée de deux colonnes (*id*, *value*) pour stocker les valeurs de propriétés. Également pour chaque classe de la base de données, une table à une seule colonne (*id*) lui est associée. Cette colonne contient l'identifiant des objets de la classe. Pour optimiser les requêtes, deux index ont été créés sur les colonnes *id* et *value* de chaque table de propriété P_i. Un index a également été créé sur la colonne *id* de chaque table des classes.

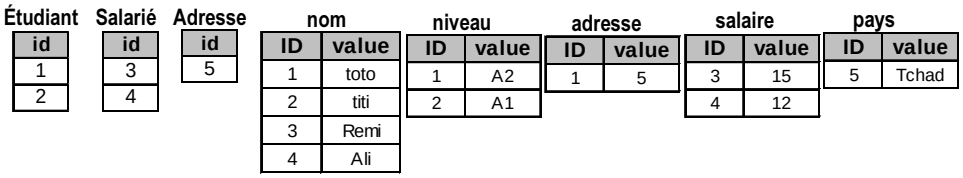


FIG. 5.7 – Approche table par propriété

1.2.4 Approche de représentation par classe concrète ou horizontale

Dans cette approche que nous avons proposé dans notre architecture *OntoDB*, on crée une table pour chaque classe concrète (cf. figure 5.8). La table n'est constituée que des colonnes des propriétés effectivement valuées pour au moins une instance de la classe [127]. C'est l'approche dont nous cherchons à évaluer les performances. Dans cette approche, un index est créé pour toutes les colonnes des propriétés susceptibles d'être impliquées dans des requêtes de jointure ou de sélection.

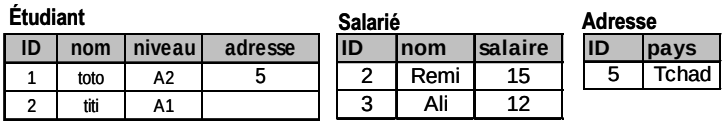


FIG. 5.8 – Approche table par classe concrète

1.2.5 Notations utilisées dans le chapitre

Tout au long du chapitre, lors de nos explications, nous utiliserons les notations formelles suivantes :
Soient :

- C l'ensemble des classes,
- $C_i \in C$ une classe dans la base de données,
- P l'ensemble des propriétés,
- $P_j \in P$ une propriété quelconque,
- P_{C_i} l'ensemble des propriétés de la classe C_i,
- C_{P_j} l'ensemble des classes où la propriété P_j est utilisée et I_{C_i} la population d'instances de la classe

- C_i. I_C représentera le nombre d'instances d'une classe quelconque si toute les classes concrètes ont la même population (ce qui est notre cas).
- $\|E\|$ la cardinalité de l'ensemble E.
- Enfin, dans les notations algébriques des requêtes, les mêmes notations représenteront une classe et la table qui la représente, d'une part, et une propriété et la colonne qui lui correspond, d'autre part. Ainsi C_i et P_j désignent respectivement le table de la classe C_i et la colonne de la propriété P_j.

1.3 Charge de requêtes

Nous proposons de comparer le temps de réponse de chacune des trois approches de représentation (*table par classe*, *table par propriété* et *triplets*) à partir d'une série de requêtes.

Les requêtes que nous considérons dans le cadre de notre évaluation, peuvent être classées en trois familles.

1.3.1 Requêtes typées

Cette famille est constituée de requêtes qui accèdent aux données via des classes. Cela correspond aux requêtes les plus usuelles dans une base de données dans lesquelles l'utilisateur connaît la ou les classes qu'il veut interroger. Les requêtes sont du style : trouver les instances d'une classe C_i ayant des valeurs V_1 et V_2 pour les propriétés P_1 et P_2. Par exemple : trouver toutes les *personnes* de la base de données ayant l'*âge* de *35 ans*. Notons que la classe C_i peut être une classe feuille ou une classe non feuille pour laquelle les propriétés recherchées sont applicables. Dans ce dernier cas, la requête porte sur toutes les sous-classes de la classe non feuille (requête dite "polymorphe" étudiée en particulier dans [157]).
Ces requêtes comportent :
- les requêtes de projection,
- les requêtes de sélection,
- les requêtes de jointure,
- des combinaisons de requêtes de projection, sélection et/ou de jointure.

Ces requêtes peuvent être représentées par la signature suivante :
$CLASS^{+}[\times PROPERTY^{*}[\times VALEUR^{*}]] \longrightarrow VALEUR^{*}$[22].

Exemple :

SELECT S.id, S.nom, S.age
FROM Salarie S

Le traitement de ces requêtes nécessite une connaissance de la (ou les) classe(s) que l'on souhaite interroger et éventuellement les propriétés que l'on souhaite calculer (sinon elles sont

[22] $[], +, *$ sont méta notations usuelles dans les grammaires. [*Expression*] indique que le terme *Expression* est optionnel. $Expression^{+}$ indique que le terme *Expression* apparaît au moins une fois ($n \geq 1$). $Expression^{*}$ indique que le terme *Expression* apparaît zéro ou n fois ($n \geq 0$)

toutes calculées) et les valeurs des prédicats de sélection.

1.3.2 Requêtes non typées

Cette famille est constituée de requêtes qui visent à récupérer des instances dans la base de données qui respectent un ensemble de valeurs de propriétés sans connaître la (ou les) classe(s) auxquelles elles appartiennent. Ces requêtes sont du style : trouver les instances de la base de données et qui ont pour valeurs V_1 et V_2 pour les propriétés P_1 et P_2. Par exemple : trouver toutes les instances de la base de données de la base de données, quelle que soit leur classe, qui ont une valeur *35* pour la propriété *age*.
Ces requêtes correspondent à la signature :
$(PROPERTY^{+}[\times VALEUR^{*}]^{*} \Rightarrow [CLASS^{*}] \times (VALEUR^{*})$

1.3.3 Requêtes de modification

Cette famille est constituée de requêtes d'insertion, de suppression et de mises à jour des instances de la base de données.
Exemple :

INSERT INTO Étudiant (id, nom, prénom) **VALUES**(100,'Toto','Remi') ;
DELETE FROM Étudiant **WHERE** id=100 ;

1.3.4 Quelques définitions

Pour faciliter la compréhension de notre étude expérimentale, nous introduisons quelques définitions.

Définition 1 : Facteur de sélectivité d'une condition de sélection
Une condition de sélection (p) sur une table T est une expression logique de la forme :

$$p : attribut\ \theta\ valeur$$

tels que : attribut $\in$ T, $\theta \in \{<, >, =, \leq, \geq\}$ et valeur $\in$ domaine(attribut).
Un *facteur de sélectivité* d'une condition de sélection (p) est une valeur réelle comprise entre 0 et 1. Elle est définie comme le rapport des instances de T qui satisfont cette condition et celles de la table T. Formellement, ce facteur de sélectivité ρ est défini par l'équation suivante :

$$\rho = \frac{||\sigma_p(T)||}{||T||} \tag{1}$$

où σ représente l'opération algébrique de sélection.

Définition 2 : Facteur de sélectivité d'une condition de jointure
Une condition de jointure (q) sur deux tables R et S est une expression logique de la forme :

$$q : attribut_1\ \theta\ attribut_2$$

,

tels que $attribut_1\ \in R$, $\theta \in \{<, >, =, \leq, \geq\}$ et $attribut_2\ \in$ S.
Un *facteur de sélectivité d'une jointure* est une valeur réelle comprise entre 0 et 1. Elle est

définie comme le rapport des instances du produit cartésien des deux tables (R et S) qui participent à la condition de jointure et des instances du produit cartésien. Formellement, ce facteur de sélectivité χ est défini par l'équation suivante :

$$\chi = \frac{||R \bowtie_q S||}{||R \times S||} \tag{2}$$

où $\bowtie$ et $\times$ représentent les opérations de jointure et du produit cartésien, respectivement.

Dans les sections qui suivent, nous évaluons les temps d'exécution de chacune des trois familles de requêtes sur les trois approches de représentation.

1.4 Évaluation des performances des approches de représentation sur les requêtes typées

Nous effectuons ici des requêtes de projection, de sélection et de jointure. Les requêtes sont effectuées aussi bien au niveau des tables des classes feuilles qu'au niveau des tables correspondant à des classes non feuilles. Nous faisons également varier le nombre de propriétés pour les requêtes de projection et le facteur de sélectivité (ρ) dans les requêtes de sélection et de jointure, afin d'étudier leurs impacts sur chacune des approches de représentation.

Les requêtes typées que nous évaluerons dans cette section sont :

1. Projection dans une classe feuille (section 1.4.1),
2. Sélection dans une classe feuille (section 1.4.2),
3. Jointure sur deux classes feuilles (section 1.4.3),
4. Projection Jointure Sélection (section 1.4.4),
5. Projection sur une classe non feuille (section 1.4.5),
6. Sélection sur une classe racine (section 1.4.5).

Pour chacune des requêtes, nous présenterons successivement :

(1) l'expression des requêtes exécutées pour chacune des trois approches,

(2) les différents tests effectués et les résultats obtenus,

(3) enfin, nous concluons sur l'ensemble des résultats en tentant de les expliquer.

1.4.1 Projection sur une classe feuille

Les requêtes de projection permettent de comparer les performances des différentes approches en fonction du nombre de propriétés retournées par une requête. Voici un exemple d'expression des requêtes selon les différentes approches :

- **Approche *table par classe***
  SELECT S.id, S.nom, S.age
  FROM Salarie S
- **Approche *table par propriété***

```
SELECT   Salarie.id, nom.value, age.value
FROM     Salarie S
         LEFT OUTER JOIN nom USING(id)
         LEFT OUTER JOIN age USING(id)
```

– **Approche *triplets***

```
SELECT   S.id, nom.objet, age.objet
FROM     salarie AS S
         LEFT OUTER JOIN triplets AS nom
         ON (S.id = nom.sujet AND nom.prédicat="nom")
         LEFT OUTER JOIN triplets AS age
         ON(S.id = age.sujet AND age.prédicat="age")
```

Tests

Pour les requêtes de projection, nous ferons trois tests.

– **Test 1 :** Ce test compare les trois approches de représentation (*TC*, *TP* et *triplets*) sur une petite base de données. Nous utiliserons la BD_1K_10P qui contient 134 classes feuilles et dont chaque classe possède 1K instances et décrite par 10 propriétés.

– **Test 2 :** Ce test mesure l'incidence de passage à l'échelle pour les approches de représentation *TC* et *TP*. Ce test portera sur les bases de données de la **Série 1** (cf. figure 5.2). Ce sont les bases de données à volume croissant : BD_1K_10P, BD_1K_25P et BD_1K_50P.

– **Test 3 :** Ce test mesure l'incidence du ratio nombre de propriétés (NP) sur le nombre d'instances par classe pour les approches de représentation *TC*, *TP*. Ce test utilise les bases de données de la **Série 2** (cf. figure 5.3). Ce sont les bases de données à volume constant. Nous utiliserons les bases de données : BD_10K_10P, BD_4K_25P et BD_2K_50P.

Les résultats de nos différents tests sont donnés dans la figure 5.9 pour le **Test 1**, la figure 5.10 pour le **Test 2**, et la figure 5.11 pour le **Test 3**.

1.4.1.1 Test 1 - Requêtes sur la base de données à volume réduit

La figure 5.9, nous montre les performances des requêtes sur une petite base de données : 10 propriétés et 1K instances par classe.

1. On peut remarquer sur la figure que l'approche *table par classe* a des résultats supérieurs ou égaux à ceux des deux autres approches (*table par propriété* et *triplets*). Ils sont supérieurs dès que le nombre de propriétés projetées dans les requêtes est supérieur à 1.
2. Comme l'avait montré Theoharis [157], l'approche *table par propriété* est toujours, au moins aussi performante que l'approche *triplets*.
3. Les ratios de performance de l'approche *table par classe* par rapport aux approches *table par propriété* et *triplets* est d'autant plus grand que le nombre de propriétés retournées dans les requêtes est grand. Il reste néanmoins assez faible pour les petites bases de données.

Discussion :

Les performances de l'approche *TC* sur les approches *triplets* et *table par propriété* s'expliquent par le volume des tables et le coût des jointures et/ou auto-jointure. En effet, le coût des requêtes

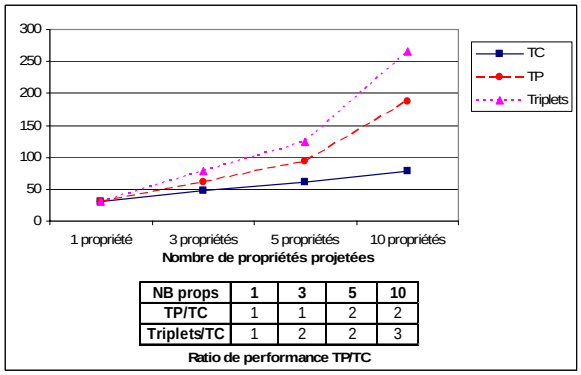


FIG. 5.9 – Projection sur une petite base de données : 10 propriétés et 1K instances par classe

exécutées dans une approche *TC*, se limite à une projection ($\prod_{(P_1,...,P_n)} C_i$) d'une unique table contenant $\|I_{C_i}\|$ instances. Contrairement à l'approche *table par propriété* qui nécessite de faire n jointures entre les tables des propriétés dont chaque table de propriété P_i contient $\|C_{P_i}\| \times \|I_{C_i}\|$ instances ($\prod_{P_1.value,...,P_n.value}(C_i \bowtie P_1 \bowtie ... \bowtie P_n)$).

L'approche *triplets*, elle nécessite $n-1$ auto-jointures sur la table *triples* contenant $\sum_{C_i \in BD}(\|I_{C_i}\| \times \|P_{C_i}\|)$ tuples et une jointure sur la table des *instances* contenant I_{C_i} tuples. La requête est de la forme : $\prod_{t_1.object,...,t_n.object}(C_i \bowtie \sigma_{predicat=P_1} triples \bowtie ... \bowtie \sigma_{predicat=P_n} triples)$, avec n le nombre de propriétés projetées.

On remarque que les tables utilisées dans la jointure de l'approche *triplets* sont de tailles supérieures (trois colonnes au lieu de deux) à celles utilisées dans la jointure de l'approche *TP*. L'approche *triplets* ne pourra donc en aucun cas donner de meilleurs résultats.

L'approche par triplet apparaissant toujours inférieure à l'approche *TP*, nous ne nous intéresserons désormais qu'aux deux approches *TC* et *TP*.

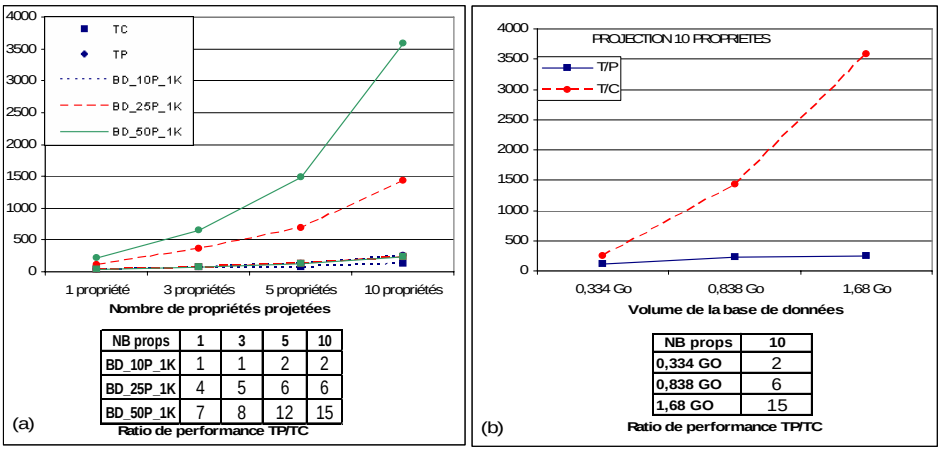
1.4.1.2 Test 2 - Le passage à l'échelle

La figure 5.10 donne les résultats lors du passage à l'échelle des bases de données tests.

1. On peut constater, les meilleures performances de l'approche de *table par classe* sur l'approche *table par propriété*.
2. Le temps de réponse des requêtes (quelques soient l'approche ou la base de données) sont des fonctions croissantes du nombre de propriétés projetées et la taille de la base de données.
3. La performance de l'approche *TC* par rapport à l'approche *TP* *croît fortement* lorsque la taille des bases de données augmente (passage à l'échelle).

Discussion :

– La meilleure performance de l'approche *TC* par rapport à l'approche *TP* se justifie, comme dans les tests précédents, par le coût des jointures et volume des tables des propriétés. La variation du nombre de propriétés projetées n'a pas d'effet sur l'approche *TC* parce que



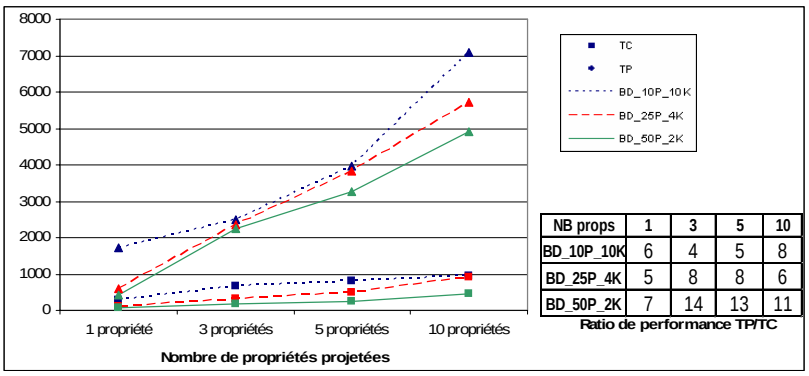
NB props	1	3	5	10
BD_10P_1K	1	1	2	2
BD_25P_1K	4	5	6	6
BD_50P_1K	7	8	12	15

NB props	10
0,334 GO	2
0,838 GO	6
1,68 GO	15

FIG. 5.10 – Performance de la projection lors du passage à l'échelle.

lors d'une requête projection, l'optimiseur de requêtes récupère tous les tuples (toutes les propriétés des instances) avant de tronquer les sous-ensembles de propriétés désirées. D'où les temps de réponse relativement identiques quelque soit le nombre de propriétés projetées dans l'approche *TC*. Tandis que dans l'approche *TP*, la variation de nombre de propriétés projetées ses traduit par des jointures sur des tables propriétés assez volumineuses ($\|C_{P_i}\| \times \|I_{C_i}\|$).

– Le coût de l'approche *TP* croît plus vite que celle de l'approche *TC* lorsque la taille des bases de données augmente parce que, dans le premier cas, les tables de propriétés croissent plus vite que les tables de classe. En effet, valuer plus de propriétés par classe signifie que la même propriété est valuée dans plus de classes.



NB props	1	3	5	10
BD_10P_10K	6	4	5	8
BD_25P_4K	5	8	8	6
BD_50P_2K	7	14	13	11

FIG. 5.11 – Performance projection sur des bases de données à volume constant : variation du nombre d'instances et de propriétés par classe.

1.4.1.3 Test 3 - variation du nombre de propriétés sur le nombre d'instances par classe

La figure 5.11 donne les résultats des tests où, à taille constante de base de données, on fait

varier en sens inverse le nombre de propriétés valuées et le nombre d'instances par classe.

1. Les bases de données ayant une taille significative montrent que l'approche *TC* a largement une meilleure performance par rapport à l'approche *TP*,
2. à taille de base de données et nombre de propriétés projetées constants, les temps de réponse des requêtes sur l'approche *TC* croissent régulièrement avec le nombre d'instances par classe,
3. les temps de réponse des requêtes sur l'approche *TP* évoluent de façon anarchique par rapport au nombre d'instances par classe,
4. les temps de réponse des requêtes croissent selon le nombre d'instances initialisées par classe,
5. les ratios de performances croient avec le nombre de propriétés initialisées par classe.

Discussion

1. Lorsque, dans une classe feuille, le nombre d'instances croit et le nombre propriétés décroît, la taille de la table *TC* reste constante. Par contre la projection ayant lieu sur des propriétés représentées dans la classe, le nombre d'instances ayant cette propriété augmente. Il est donc normal que le temps de réponse de projection dans l'approche *TC* soit une fonction croissante du nombre d'instances. Dans la figure 5.12, bien que les deux tables aient une taille identique, on peut remarquer la nette différence de lignes entre elles. Une projection sur la table *BD_10K_10P* aura un temps de reponse plus long que sur la table *BD_2K_50P*.

Table BD_10K_10P			
Id	P1	...	P10
1			
2			
...			
10 000			

Table BD_2K_50P																			
id	P1	...																...	P50
1																			
2																			
...																			
2000																			

FIG. 5.12 – Projection sur une table de classe à volume constante (a) 10 propriétés, 10K instances, (b) 50 propriétés, 2K instances.

2. Dans l'approche *TP*, lorsque le nombre de propriétés par classe et le nombre d'instances par classe varient de façon inverse, laissant constant, dans chaque classe, le produit : (nombre d'instances par classe) × (nombre de propriétés par classe) à la classe également constant, à l'échelle de la base de données : (nombre total d'instances) × (nombre total de propriétés/instances). Le nombre total d'instances ayant une valeur pour une propriété donnée reste à peu près constant, mais il peut diminuer ou augmenter selon la répartition (aléatoire) des propriétés. La taille des tables *TP* ne peut donc varier de façon imprévisible. Il n'est donc pas étonnant que les temps de réponse aux requêtes de projections dans l'approche *TP* dépendent pas moins régulièrement du nombre d'instances par classe.

1.4.1.4 Conclusion sur les requêtes de projection sur une classe feuille

La conclusion que nous pouvons tirer des tests projection sur les différentes configurations de bases de données est que l'approche *TC* est toujours plus performante que les deux autres approches de représentations (*TP* et *Triplets*), et ce, quelle que soit la taille des bases de données : petites bases de données ou bases de données à grand volume de données. Lors du passage à l'échelle, les coûts de l'approche *TC* restent relativement stables tandis que ceux des approches *TP* et *Triplets* évoluent *très fortement*. Enfin, comme prévu, l'approche *TP* est plus performante que l'approche par *Triplets*.

1.4.2 Sélection sur une classe feuille

Afin de comparer les trois approches de représentation, nous faisons varier, (1) dans un premier test, le facteur de sélectivité (ρ) des requêtes (de 0,1%, 1%, 10%, et 25%) et (2) le nombre de propriétés retournées dans les requêtes et (3) le nombre de propriétés dans les prédicats de sélection.

Requêtes selon les approches

Voici un exemple d'expression des requêtes selon les différentes approches :

1. **Approche table concrète**

```
SELECT  id, nom, age
FROM    Salarie
WHERE   nom = "%toto%"
```

2. **Approche table par propriété**

```
SELECT  S.id, nom.value, age.value
FROM    Salarie S
        LEFT OUTER JOIN nom USING (id)
        LEFT OUTER JOIN age USING(id)
WHERE   nom.value = "toto"
```

3. **Approche triplets**

```
SELECT  S.id, nom.objet, age.objet
FROM    Salarie AS S
        LEFT OUTER JOIN triplets AS nom
        ON (S.id = nom.sujet AND nom.prédicat="nom")
        LEFT OUTER JOIN triplets AS age
        ON (S.id = age.sujet AND age.prédicat = "age")
WHERE   nom.objet = "toto"
```

Tests

Pour les requêtes de sélection, nous ferons quatre tests. Les trois premiers (Test 1, Test 2, Test 3) correspondent à ceux définis pour les requêtes de projection (test sur une petite base de données - BD_1K_10P, test sur l'incidence du passage à l'échelle et le test sur l'incidence du ratio nombre de propriétés / nombre d'instances). Dans ces tests, les prédicats de sélection des requêtes de sélection ne contiendront qu'une seule propriété. Dans le **Test 4**, nous mesurerons l'incidence de la variation du nombre de propriétés dans le prédicat de sélection suivant les approches *TC* et *TP*. Ce test portera sur l'une des bases de données dans la **Série 2** : la base de données BD_2K_50P.

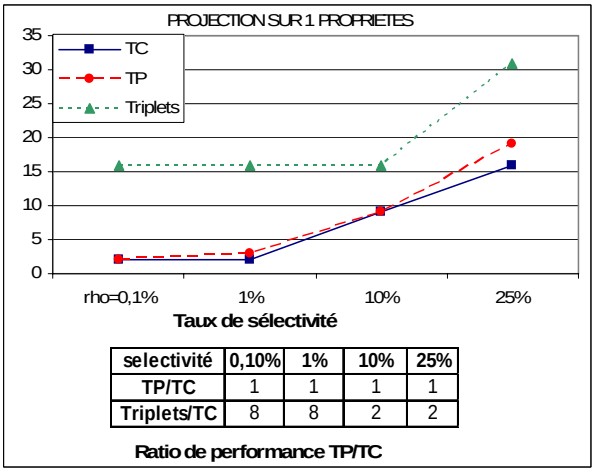


FIG. 5.13 – Sélection sur une petite base de données : 10 propriétés et 1K instances par classe

1.4.2.1 Test 1 - Comparaison des trois approches de représentation

Les résultats sur la petite base de données (BD_1K_10P) sont décrits dans la figure 5.13 qui donne les performances d'une requête de sélection sur une propriété et une projection sur une autre propriété différente de la propriété de sélection.

- On peut constater sur la figure que les temps de réponse des approches *TC* et *TP*, sont quasiment identiques quels que soient les taux de sélectivité, l'approche *TC* étant néanmoins toujours légèrement préférable.
- L'approche *Triplets* est cette fois beaucoup moins intéressante que les deux autres.

  En notation algébrique, une requête de sélection sur l'approche *TC* s'écrira :
  $\prod_{(P_1,...,P_n)} \sigma_{P_s=X} C_i$
  Dans une approche *TP*, elle s'écrit :
  $\prod_{(P_1,...,P_n)} \sigma_{P_s.value=X}(C_i \bowtie P_s \bowtie P_1 \bowtie ... \bowtie P_n)$

Discussion :

- La performance presque équivalente des approches *TC* et *TP* s'explique par le fait que la requête de sélection exécutée sur l'approche *TP* ($\prod_{(P_1)} \sigma_{P_s.value=X}(C_i \bowtie P_s \bowtie P_1)$) nécessite deux jointures sur des tables qui peuvent être logées entièrement en mémoire centrale (évitant ainsi toute opération d'E/S). Ainsi dans les deux cas le coût principal est la représentation du résultat qui dépend du taux de sélectivité.
- La mauvaise performance de l'approche *Triplets* s'explique par les deux auto-jointures de la requête de sélection sur une table qui ne logeant pas en mémoire centrale, nécessitera des paginations.

1.4.2.2 Test 2 - Incidence du passage à l'échelle

Les tests réalisés dans cette section sont effectués sur des bases de données de taille croissante (**Série 1**). Les requêtes portent sur une seule propriété dans le prédicat de sélection et une seule propriété retournée (différente de celle du prédicat).

La figure 5.14a compare les approches *TC* et *TP* pour différents facteurs de sélectivité et pour différentes tailles de base de données. La figure 5.14b compare le passage à l'échelle des approches *TC* et *TP* avec un facteur de sélectivité de 25%. On peut remarquer sur la figure que :

- le ratio de performance TP/TC est très fortement croissant avec la taille de la base de données. l'*approche TP passe très mal à l'échelle.* Pour la plus grosse base de données, les ratios de performance varient entre 15 et 100 selon le facteur de sélectivité.
- le facteur de sélectivité a un impact similaire sur les performances des requêtes quelle que soit l'approche. Il est pratiquement négligeable pour les grosses bases de données dans l'approche *TP*.

Discussion :

- Pour exécuter la requête de sélection dans l'approche *TC*, logiquement l'optimiseur devrait (1) charger, dans un premier temps, en mémoire la table d'index de la colonne de la propriété sur laquelle porte la sélection, (2) puis calculer les instances de la classe qui ont la valeur de la propriété demandée, (3) et enfin, récupérer les instances des classes sélectionnées. Dans l'approche *TP*, (1) la table de la classe C_i sera jointe avec la table de la propriété à sélectionner (P_s) et la table de la propriété à projeter (P_1), et enfin (2) la table de la jointure ainsi construite sera filtrée sur la propriété de sélection (P_s).

  La meilleure performance de l'approche *TC*, observée sur la figure 5.14b, s'explique alors par le fait que seules sont chargées en mémoire les instances de la classe effectivement sélectionnées contrairement à l'approche *TP* où toutes les tables sont chargées en mémoire pour être jointe avant l'application des prédicats de sélection. Le coût d'E/S et des jointures sont donc responsables de la mauvaise performance de l'approche *TP*.

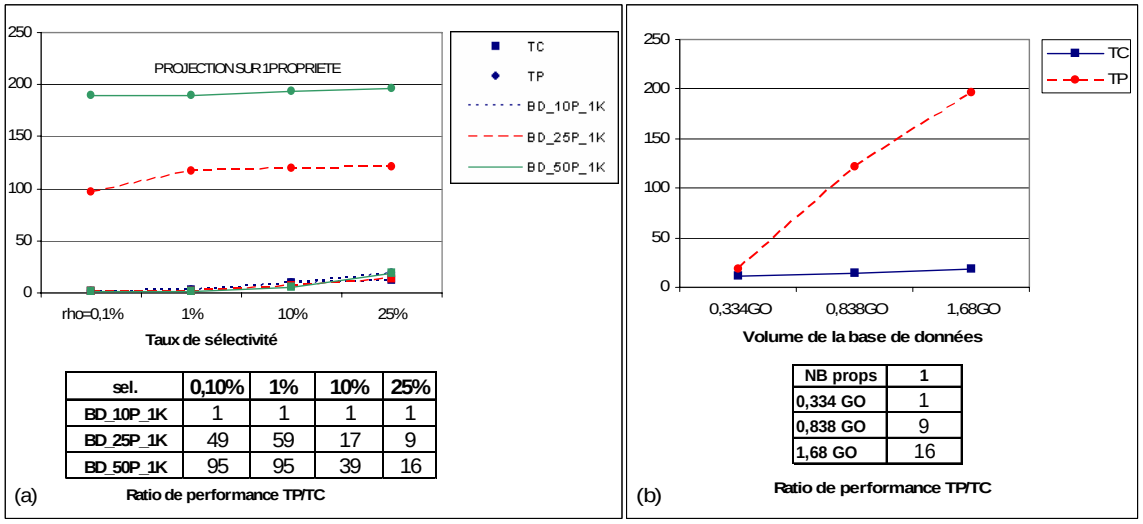
  La supériorité de l'approche *TC* diminue lorsque le facteur de sélectivité augmente puisque dans l'approche *TC*, il faut charger de plus en plus d'instances alors que dans l'approche *TP* toutes les instances sont, de toute façon, chargées.

- Sur la figure 5.14b, la variation de la taille des bases de données qui fait augmenter considérablement les temps de réponse de l'approche *TP* s'explique simplement par le fait que les tables instances et des propriétés augmentant de taille, alors le temps de chargement de celles-ci en mémoire pour être jointes devient plus long. Dans l'approche *TC*, *vu que seules sont chargées en mémoire centrale les instances effectivement sélectionnées issues de la table d'index*, la taille de la base de données, influent moins sur les performances.

1.4.2.3 Test 3 - Incidence du ratio nombre de propriétés sur le nombre d'instances

La figure 5.15 présente les résultats des performances sur différentes structures de bases de données ayant un volume constant (**Série 2**). La figure 5.15 donne les performances des requêtes de sélection sur une propriété avec projection sur une autre propriété. On constate sur la figure que :

- l'approche *TC* offre des temps de réponses *très* inférieurs à l'approche *TP* quelque soit le facteur de sélectivité,



sel.	0,10%	1%	10%	25%
BD_10P_1K	1	1	1	1
BD_25P_1K	49	59	17	9
BD_50P_1K	95	95	39	16

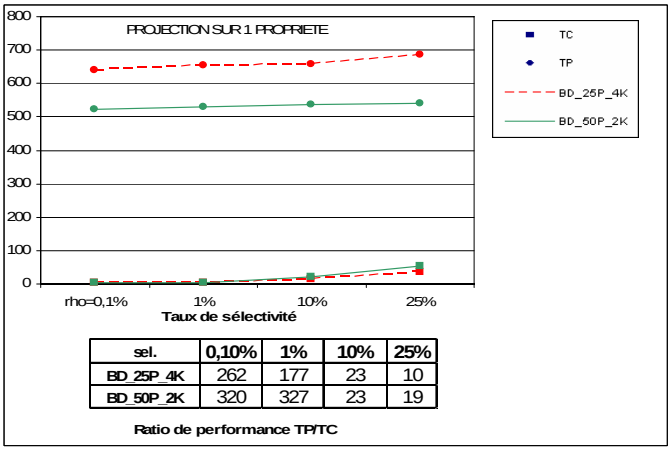
NB props	1
0,334 GO	1
0,838 GO	9
1,68 GO	16

FIG. 5.14 – Performance sélection passage à l'échelle, 134 classes, 1K instances par classe, nombre variable de propriétés.

– les ratios de performance TP/TC des tests sur cette série de bases de données à grand volume sont assez significatifs par rapport aux tests de la section précédente.

Discussion :

Les tailles des bases de données étant encore supérieures à celles des bases de données précédentes, les chiffres de cet essai ne font que prolonger les chiffres de l'essai précédent : plus les bases de données sont grosses, plus l'approche que nous proposons s'avère préférable à l'approche TP.



sel.	0,10%	1%	10%	25%
BD_25P_4K	262	177	23	10
BD_50P_2K	320	327	23	19

FIG. 5.15 – Performance sélection sur des bases de données à volume constant : variation du nombre d'instances et de propriétés par classe.

1.4.2.4 Test 4 - Incidence de la variation du nombre de propriétés dans le prédicat de sélection

La figure 5.16 donne les performances des requêtes sur sélection en faisant varier le nombre de propriétés de 1 à 10 dans le prédicat de sélection des requêtes. Ces requêtes sont exécutées sur une base de données à 50 propriétés et 2K instances par classe. Les propriétés du prédicat de sélection sont combinées par des disjonctions (OU). On constate sur la figure que :

- les temps de réponse des requêtes sur l'approche TP augmentent significativement avec le nombre de propriétés dans le prédicat de sélection tandis que l'approche TC reste stable avec le nombre de propriétés,
- le ratio de performance entre l'approche TC et TP croit aussi significativement avec le nombre de propriétés dans le prédicat de sélection.

Discussion :

Cette meilleure performance s'explique une fois de plus par le nombre de jointures et le volume des tables des propriétés. En effet, la requête de sélection dans l'approche TC s'écrit algébriquement de la forme $\prod_{(P_1,...,P_n)} \sigma_{P_j=X_j\ OR...OR\ P_k=X_k} C_i$. Son exécution consistera à charger la table des index des propriétés du prédicat de sélection en mémoire puis à calculer les instances (tuples) de la table qui répondent à la requête et qui seront enfin chargées en mémoire.

Dans l'approche TP, la requête algébrique s'écrit sous la forme :
$\prod_{(P_1,...,P_n)} \sigma_{P_j.value=X_j\ OR...OR\ P_k.value=X_k}(C_i \bowtie P_1 \bowtie ... \bowtie P_n \bowtie P_j ... \bowtie P_k)$
où chaque table P_i contient $\|C_{P_i}\| \times \|I_C\|$, l'exécution de cette requête nécessite plusieurs opérations de jointure entre les différentes tables. Étant donné que la jointure est une opération coûteuse, cela justifie la mauvaise performance de cette approche.

L'ajout de nouvelles propriétés dans le prédicat de sélection se traduit dans l'approche TP (1) par le chargement des tables de ces propriétés en mémoire, (2) de nouvelles jointures et (3) de nouvelles conditions dans le prédicat de sélection. Dans l'approche TC, elle se traduit simplement par le chargement en mémoire des tables d'index de ces propriétés (qui sont normalement de petite taille) et l'application du prédicat de sélection.

La croissance rapide des temps de réponse de l'approche TP à partir de la 7ième propriété s'explique par le mécanisme de pagination que subit le moteur des requêtes. En effet, à partir d'un certain nombre de propriétés (7 dans notre cas), le *buffer central* (cache de données) se remplit et le SGBD pour pouvoir charger en mémoire de nouveaux blocs mémoires des tables des propriétés, doit "dé-localiser" (renvoyer) certains blocs mémoires sur disque et les recharger ultérieurement en mémoire centrale. Ce processus de va-et-vient de la mémoire centrale vers le disque dur, qui se traduit par de nombreuses E/S, est responsable du pic constaté.

1.4.2.5 Conclusion sur la sélection sur une classe feuille

Pour les tests de sélection, nous pouvons tirer les conclusions suivantes.

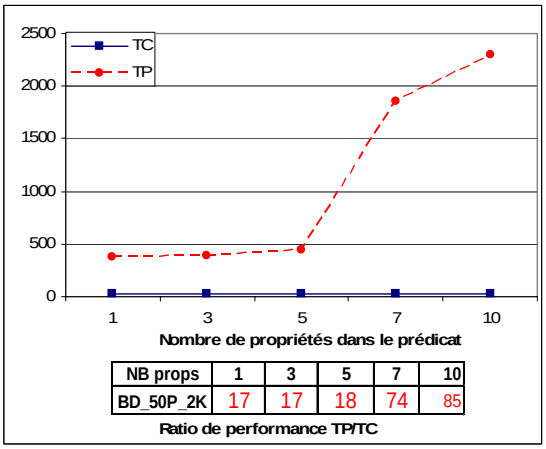


FIG. 5.16 – Sélection variation du nombre de propriétés dans le prédicat de sélection, 134 classes, 50 propriétés et 2K instances par classe

- L'approche *TC* offre des temps de réponses meilleurs par rapport aux deux autres approches de représentations (*TP* et *Triplets*) et ce quelle que soit la taille des bases de données : petite base de données ou bases de données à grand volume de données.
- Même avec une seule propriété de sélection et la projection sur une seule propriété le ratio de performance *TP/TC* croît très fortement lorsque la taille de la base de données augmente. Selon le ratio de sélectivité, le ratio de performance varie de 1 à 20 et 1 à 300. L'approche *TC* passe donc très considérablement mieux à l'échelle que l'approche *TP*.
- Lorsque le nombre de propriétés dans le prédicat de sélection devient supérieur à 1, la supériorité de l'approche *TC* augmente encore.

Dans la section qui suit, nous montrons les performances sur des requêtes de jointures entre deux classes feuilles.

1.4.3 Jointure sur deux classes feuilles

Les tests de requêtes de jointure permettent de comparer les performances des différentes approches en joignant des tables des classes. Le facteur de sélectivité de jointure (ρ) des requêtes est de 0.25%.

Requêtes selon les approches

Voici un exemple de requête de jointure suivant les approches :

1. **Approche table concrète**

   SELECT S.nom, E.nom, E.niveau
   FROM Salarie S, Étudiant E
   WHERE S.adresse = E.adresse

2. **Approche table par propriété**

```
SELECT   Snom.value, Enom.value, Eniveau.value
FROM     Salarie S
         LEFT OUTER JOIN nom AS Snom USING (id)
         LEFT OUTER JOIN adresse AS SA USING(id),
         Etudiant E
         LEFT OUTER JOIN nom AS Enom USING (id)
         LEFT OUTER JOIN adresse AS Eniveau USING(id)
         LEFT OUTER JOIN niveau AS EA USING(id)
WHERE    SA.value = EA.value
```

3. **Approche triplets**

```
SELECT   Snom.objet, Enom.objet,niveau.objet
FROM     Salarié AS S
         LEFT OUTER JOIN triplets AS Snom
         ON (S.id = Snom.sujet AND Snom.prédicat="nom")
         LEFT OUTER JOIN triplets AS SA
         ON (S.id = SA.sujet AND SA.prédicat="adresse"),
         Étudiant AS E
         LEFT OUTER JOIN triplets AS Enom
         ON (E.id = Enom.sujet AND Enom.prédicat = "nom")
         LEFT OUTER JOIN triplets AS niveau
         ON (E.id = niveau.sujet AND niveau.prédicat="niveau")
         LEFT OUTER JOIN triplets AS EA
         ON (E.id = EA.sujet AND EA.prédicat="adresse"),
WHERE    SA.objet = EA.objet
```

Tests

La figure 5.17 donne les performances des requêtes de jointure exécutées sur les bases de données à volume croissant (**Série 1**).

On peut remarquer sur la figure 5.17a que l'approche *TC* est toujours plus performante que l'approche *TP* avec un ratio de 4 à 7 lorsque la taille des bases de données augmente.

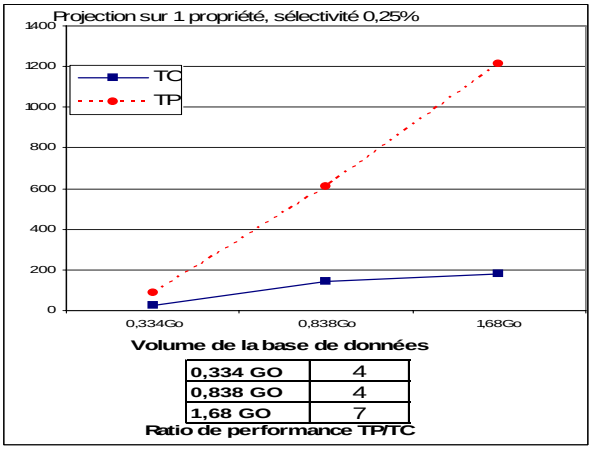


FIG. 5.17 – Jointure de deux classes sur les bases de données à volume constant

Discussion :

1. Dans l'approche *TC*, la réalisation de la jointure se fera en deux étapes : (1) les tables des deux classes sont avant tout réduite par une projection sur la colonne de la propriété à retourner et la colonne de jointure ($\prod_{(rid,P_1,P_j)} C_i$) et (2) puis l'opération de jointure est effectivement effectuée sur les deux tables réduites.

   Dans l'approche *TP*, où la requête de jointure des deux classes qui s'écrit algébriquement : $(\prod_{P_1.value,P_2.value}((C_1 \bowtie_{id} P_1 \bowtie_{id} P_{J1}) \bowtie_{value} (P_{J2} \bowtie_{id} P_2 \bowtie_{id} C_2)))$[23].

   Elle s'exécutera (1) avant tout par les jointures ($C_i \bowtie_{id} P_i \bowtie_{id} P_J$) de la table des instances (C_i) avec la table de la propriété retournée (P_i) et la table de la propriété de jointure (P_J) pour chacune des classes C_1 et C_2, et (2) enfin, les tables résultantes de ces jointures sont de nouveau jointes entre elles sur les colonnes "*value*" des deux tables des propriétés de la jointure.

   Remarquons que quelle soit l'approche *TC* ou *TP*, la première étape de l'exécution de la requête de jointure entre deux classes consiste à faire une projection sur deux propriétés de chacune de deux classes. La deuxième étape consiste à faire une jointure sur la propriété de jointure des deux tables construites précédemment. Le coût de traitement de cette deuxième étape est la même dans l'approche *TC* et *TP* puisque (1) les deux tables sont identiques, et (2) elles sont déjà en mémoire.

   La meilleure performance de l'approche *TC* sur l'approche *TP* se justifie donc par le coût des projections sur les deux propriétés. C'est justement ce qu'on observe en comparant les ratios de performance avec les résultats obtenus dans les requêtes de projections de la figure 5.10 avec ceux que nous avons obtenus pour les requêtes de jointures (cf. figure 5.17).

   Remarquons que les ratios de performance sont quasiment identiques dans les deux tests (environ 4 pour la base de données à 0.838 GO et 7 pour la base de données à 1.68 GO).

2. Les ratios de performance augmentent avec la taille des bases de données, comme nous l'avons déjà souligné lors des requêtes de projection, l'augmentation du volume des bases de données fait augmenter la taille des tables des instances dans l'approche *TC* et la taille des tables des propriétés dans l'approche *TP*. Dans l'approche *TP*, cette augmentation du volume des tailles des tables des propriétés vont engendrer de forts coûts : (1) pour les E/S lors du chargement de celles-ci en mémoire et (2) pour les jointures des tables des propriétés.

1.4.3.1 Conclusion sur la jointure sur deux classes

Les résultats des requêtes de jointure indiquent que l'approche *TC* est plus performante que l'approche *TP* avec un ratio de 4 à 7 lorsque la taille de la base de données augmente. Ce coût est principalement dû au coût de projection, plus grand pour l'approche *TP*.

[23] $R_1 \bowtie_P R_2 \equiv R_{1\,R_1.P \bowtie_= R_2.P} R_2$

1.4.4 Projection Jointure Sélection

Les requêtes de *Projection-Jointure-Sélection* visent à utiliser dans une seule requête les opérateurs de projection, jointure et de sélection. Elles nous permettent de comparer les performances des différentes approches en fonction du nombre de propriétés retournées dans une requête, la sélectivité des données et de la jointure entre plusieurs classes. Les requêtes exécutées portent sur deux classes. Nous effectuons deux tests : le premier retournant une seule propriété, et le deuxième retournant cinq propriétés. Pour chacun de ces tests, nous faisons une sélection sur une seule propriété des deux classes avec un facteur de sélectivité de 0,1%. Les tests ont été réalisés sur des bases de données à volume croissant (**Série 1**). Les résultats de ces requêtes sont reportés dans la figure 5.18.

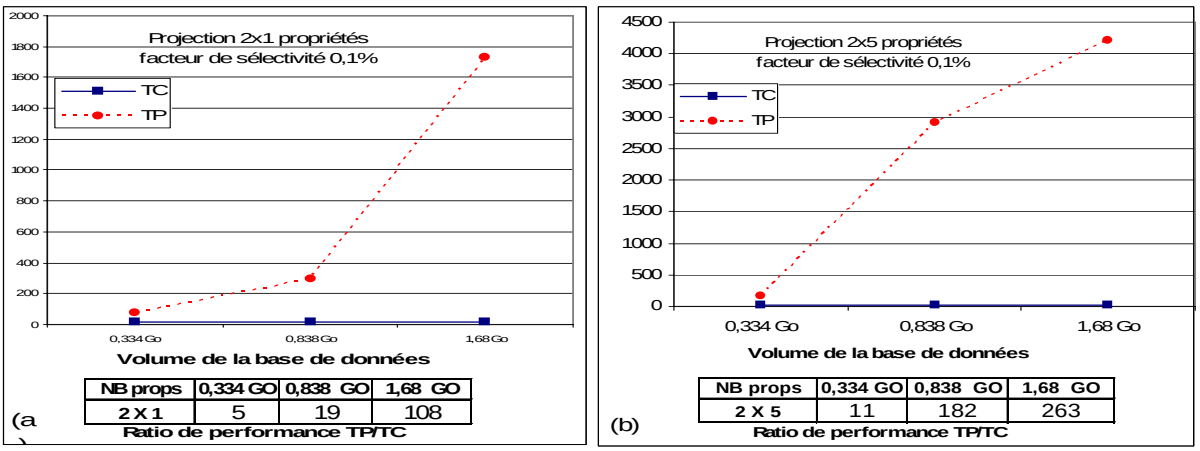


FIG. 5.18 – Projection-Sélection-Jointure sur deux classes feuilles sur des bases de données à volume croissant.

Discussion

Ces résultats confirment les tests réalisés individuellement pour les requêtes de projection (cf. section 1.4.1), de sélection (cf. section 1.4.2)), et de jointure (cf. section 1.4.3). En effet,

- la figure nous montre des temps de réponses de l'approche *TC* relativement stables et avec une croissance linéaire lors du passage à l'échelle, alors que l'approche *TP* connaît une très forte croissance amenant à des ratios de l'ordre de la centaine avec une seule propriété projetée par classe.
- Ce ratio croît encore, comme on peut s'y attendre lorsque le nombre de propriété de projection augmente.

Remarque :

Notons comme Aggrawal et al.[5] dans leur conclusion sur les tests qu'ils ont réalisés que ce test n'était pas indispensable car ce comportement résulte du comportement séparé des trois opérations. Il confirme néanmoins que *l'on peut atteindre des ratios de l'ordre de 100* même lorsque la sélection et la projection portent sur un très petit nombre de propriétés.

Tous les tests précédents ont été réalisés sur des classes feuilles. Dans la section qui suit, nous

montrons les performances des requêtes de projection et de sélection sur des classes non feuilles ayant un certain nombre de classes filles.

1.4.5 Projection et Sélection sur une classe non feuille

Les résultats des performances de la projection et sélection sur une classe non feuille sont donnés respectivement sur la figure 5.19 pour la projection et la figure 5.20 pour la sélection. Pour ces tests, on suppose qu'on dispose de toutes les sous-classes de la classe sous-feuille qu'on interroge. Notons que le temps de calcul de toutes les sous-classes est la même quelle que soit l'approche puisqu'elle consiste à accéder à la partie *ontologie* et interroger la table des classes.

Un moyen pour accélérer les accès à l'ontologie pour récupérer les sous-classes de la classe non feuille à interroger serait d'utiliser l'approche à intervalle ("labelling") présentée dans la section 3.2.2.1 du chapitre 2. Cette approche qui a été testée dans [157], s'est révélée plus performante que l'approche que nous utiliserons pour identifier les classes d'un sous-arbre de classes. L'utilisation de cette approche (que nous prévoyons d'implémenter dans une future version d'OntoDB) ne devrait pas modifier les rapports de performances entre les différentes approches que nous testons (TC et TP) puisqu'elle devrait accélérer les deux de façon similaire.

1.4.5.1 Projection sur une classe non feuille

Nous effectuons ces tests sur des bases de données de taille croissante (**Série 1**) et une base de données de la **Série 2** afin d'étudier l'incidence du passage à l'échelle.

Voici un exemple d'expression des requêtes selon les différentes approches :

- **Approche table par classe :**

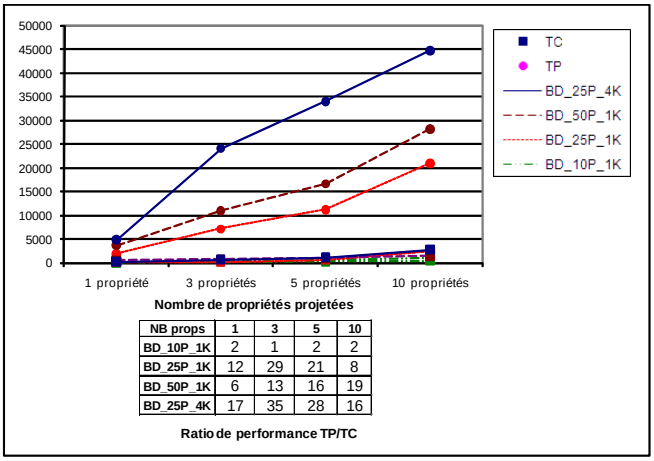
```
SELECT  id, nom, age
FROM    étudiant
        UNION
SELECT  id, nom, age
FROM    salarié
```

- **Approche *table par propriété***

```
SELECT  Étudiant.id, nom.value, age.value
FROM    Étudiant E
        LEFT OUTER JOIN nom USING(id)
        LEFT OUTER JOIN age USING(id)
        UNION
SELECT  Salarie.id, nom.value, age.value
FROM    Salarie S
        LEFT OUTER JOIN nom USING(id)
        LEFT OUTER JOIN age USING(id)
```

Une requête sur une classe non feuille s'écrivant comme une *union* de requêtes réalisées sur chacune de ses classes filles, on devrait pouvoir observer les mêmes phénomènes que ceux rencontrés lors des tests de projection sur une classe feuille dans la section 1.4.1 pour les différents aspects, à savoir :

- les ratios de performance,
- l'allure des courbes,



NB props	1	3	5	10
BD_10P_1K	2	1	2	2
BD_25P_1K	12	29	21	8
BD_50P_1K	6	13	16	19
BD_25P_4K	17	35	28	16

FIG. 5.19 – Projection classe non feuille ayant 7 sous-classes

- les temps de réponse,
- les phénomènes observés lors du passages à l'échelle.

Discussion

- Les ratios de performances des tests de projection sur des classes feuilles (cf. figure 5.10) et ceux que nous avons obtenus sur les classes non feuilles sont quasiment identiques. A titre d'exemple le ratio de performances des tests de projection sur une classe feuille dans la base de données : *BD_50P_1K* varie de 7 à 15. Les ratios des tests de projection sur une classe non feuille dans la même base de données varie de 6 à 19.
- Les allures des courbes sont similaires.
- Les temps de réponse sur les classes non feuilles sont au moins sept fois supérieurs aux temps de réponses sur une classe feuille qui correspond au nombre de sous-classes de la classe non feuille.
- Dans les tests de projection sur les classes feuilles qui comportent sept sous-classes, nous avons constaté que le nombre de propriétés retournées n'influe pas sur les temps de réponse de l'approche *TC*, au contraire de l'approche *TP*.

1.4.5.2 Sélection sur une classe non feuille

Les temps de réponse des requêtes de sélection sur une classe non feuille sont donnés dans la figure 5.20. Nous donnons seulement ici le résultat pour la plus grosse des bases de données considérées à la figure 5.19.

Voici un exemple d'expression des requêtes selon les différentes approches :

- **Approche table par classe**

```
SELECT   id, nom, age
FROM     étudiant
WHERE    nom LIKE "%toto%"
         UNION
SELECT   id, nom, age
FROM     salarié
WHERE    nom LIKE "%toto%"
```

– **Approche *table par propriété***

```
SELECT   Étudiant.id, nom.value, age.value
FROM     Étudiant E
         LEFT OUTER JOIN nom USING(id)
         LEFT OUTER JOIN age USING(id)
WHERE    nom.value = "toto"
         UNION
SELECT   S.id, nom.value, age.value
FROM     Salarie S
         LEFT OUTER JOIN nom USING(id)
         LEFT OUTER JOIN age USING(id)
WHERE    nom.value = "toto"
```

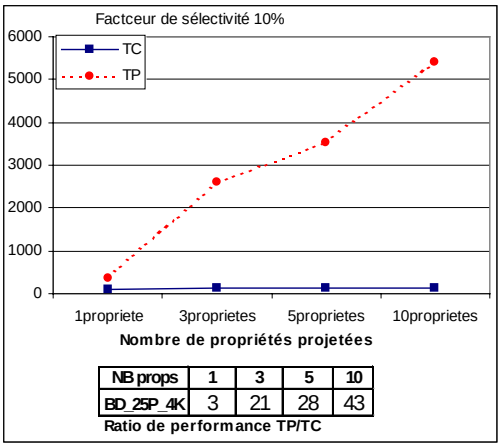


FIG. 5.20 – Sélection sur une classe non feuille ayant sept sous-classes sur une base de données avec 134 classes, 25 propriétés, 4K instances par classe.

Discussion

Les remarques faites précédemment sur les tests des requêtes de projection sur les classes non feuilles s'appliquent aux tests de sélection. En effet, comme pour les requêtes de projection, les requêtes de sélection sont réalisées par une union des requêtes réalisées sur chacune des classes filles. Nous donnons seulement ici le résultat pour la plus grosse des bases de données considérées à la figure 5.19.Les remarques faites précédemment sur les tests des requêtes de projection sur les classes non feuilles s'appliquent aux tests de sélection. En effet, comme pour les requêtes de projection, les requêtes de sélection sont réalisées par une union des requêtes réalisées sur chacune des classes filles.

1.4.5.3 Conclusion sur la projection et sélection sur une classe non feuille

En résumé, les résultats des tests de requêtes sur des classes non feuilles étant réalisée par l'union des résultats sur les différents classes feuilles, leurs résultats ne sont qu'un cumul des performances des tests sur chacune des classes feuilles. Cela se traduit par des ratios de performance qui sont tout à fait similaires.

1.4.6 Conclusion des tests des requêtes typées

A l'issu des tests sur les requêtes typées, il apparaît clairement que :

- l'approche de représentation par *triplets* présente des performances largement inférieures aux approches *TC* et *TP* ;
- sur les bases de données de petites tailles, les ratios de performances entre les approches *TP* et *TC* sont relativement faibles (1 à 4). Par contre sur les bases de données de tailles significatives (en nombre de propriétés valuées et/ou en nombre d'instances par classe), les ratios *TP/TC* croissent assez rapidement atteignant en général des ratios supérieurs à 10 pour une opération pouvant atteindre des ratios supérieurs à 100 dès que plusieurs opérations de jointures sont en jeux.

Pour ce type de requête l'approche *TC* supporte beaucoup mieux le passage à l'échelle par rapport à l'approche *TP*.

Dans la section qui suit, nous montrons les performances sur des requêtes non typées.

1.5 Évaluation des performances des approches de représentation sur les requêtes non typées

Cette deuxième famille est constituée des requêtes qui visent à récupérer des instances dans la bases de données sans connaître les classes auxquelles elles appartiennent. Nous ne nous intéresserons qu'à une seule requête qui consiste à retrouver toutes les instances de la base de données qui ont une valeur V pour la propriété P.

1.5.1 Retrouver toutes les instances de la base de données qui ont une valeur V pour la propriété P

Requêtes selon les approches

Voici un exemple de l'expression des requêtes selon les différentes approches.

Approche table par classe

Dans l'approche de représentation *TC*, la requête nécessite de s'exécuter en deux étapes :

1. Détermination de toutes les classes de la base de données qui utilisent la propriété P. Pour cela, on interroge le modèle conceptuel des données stocké dans la table *ClassExtension* de la partie *données* de l'architecture *OntoDB* (cf. section 3.1.3 du chapitre 3).

   Chaque tuple (ou instance) de cette table, représente la description du schéma d'une classe. La colonne *ClassID* est l'identifiant de la classe. La colonne *properties*, dont le

ClassExtension

ID	...	ClassID	properties
1		etudiant	{id, nom, age, niveau, adresse}
2		salarie	{id,nom,salaire}
3		adresse	{pays}

FIG. 5.21 – Modèle conceptuel de la base de données

domaine est de type collection (ARRAY en PostgreSQL), contient l'ensemble des propriétés utilisées par une classe (cf. figure 5.21).
La requête s'écrit alors :

SELECT ClassExtension.ClassId
FROM ClassExtension
WHERE P <* ClassExtension.properties

2. Exécution de la requête qui fera l'union des requêtes de sélection sur chacune des tables des classes récupérées précédemment.

SELECT id, "Etudiant" AS ClassName
FROM étudiant
WHERE nom **LIKE** "%toto%"
UNION
SELECT id, "Salarié" AS ClassName
FROM salarié
WHERE nom **LIKE** "%toto%"

Approche table par propriété
Dans l'approche *TP*, la requête s'exécutera simplement en interrogeant la table de la propriété.

SELECT id
FROM nom
WHERE nom.value = "toto"

Approche triplets
Dans l'approche *Triplets*, la requête s'exécutera aussi simplement

SELECT sujet
FROM triples
WHERE prédicat = nom **AND** objet = "toto"

Tests
Les figures 5.22a et 5.22b présentent les résultats d'exécution des requêtes précédentes respectivement sur une petite base de données sur laquelle sont comparées les trois approches (triplets, TP, TC), puis sur les bases de données à volume croissant (**Série 1**) sur laquelle des comparaisons sont affinées entre TP et TC.

1. On peut remarquer, pour cette catégorie de requêtes, que lorsque la requête consiste uniquement à retrouver l'identifiant des instances, sans retourner aucune autre propriété de ces instances notre approche *TC* présente de mauvaises performances par rapport aux approches *TP* et *triplets* qui ont quasiment les mêmes performances (cf. figure 5.22).

2. Lorsque l'on fait croître le nombre de propriétés retournées pour chaque instance, les rapports de performance évoluent. Les approches *TP* et *TC* ont un comportement voisin lorsque l'on retrouve environ 4 propriétés par instance. Puis *TC* devient plus efficace que *TP* (cf. figure 5.23)

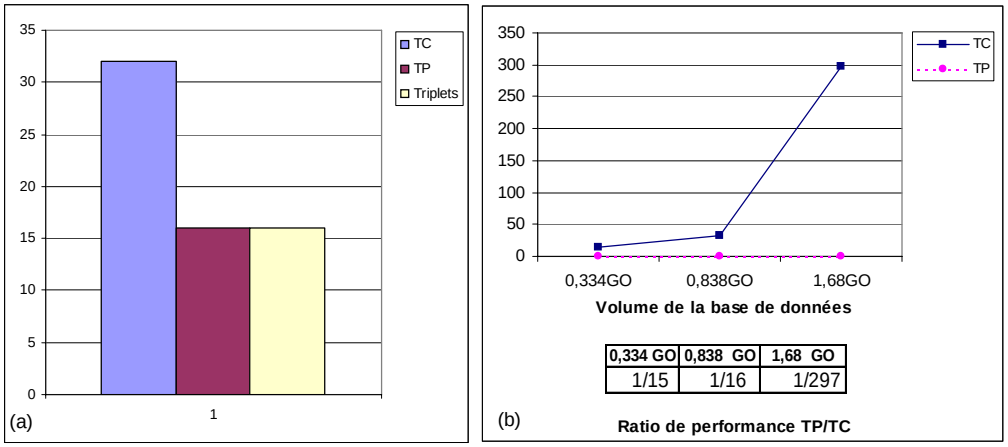


FIG. 5.22 – Temps de calcul de l'identifiant des instances de la base de données qui ont une valeur V pour la propriété P (a) sur une base de données à 10 propriétés et 1K instances par classe, (b) sur les bases de données à volume croissant.

Discussion

1. La meilleure performance de l'approche *TP* sur l'approche *TC* s'explique par le fait que dans l'approche *TC*, l'opération, comme nous l'avons souligné, se réalise en deux étapes : l'étape de récupération de toutes les classes qui utilisent la propriété P, puis l'étape dans laquelle on interroge chacune des tables de ces classes récupérées dans la précédente étape. Dans l'approche *TP*, une *seule* requête de sélection ($\prod_{id} \sigma_{value=X} P$) est exécutée sur la table de la propriété. Dans l'approche *Triplets*, une seule requête de sélection ($\prod_{sujet} \sigma_{objet=X\ AND\ prdicat=P} triples$) est aussi exécutée sur la table *triples*.
2. Notons qu'une des raisons de la croissance du coût de l'approche *TC* lors de l'augmentation de taille des bases de données est dû au temps de calcul des classes de la base de données qui utilisent la propriété. Cette tâche s'effectue en parcourant la liste des propriétés (cf. colonne *properties* de la figure 5.21) utilisées de chaque classe de l'ontologie au moyen de l'opérateur "$<*$". Celui-ci nécessite de parcourir de toute la liste des propriétés et de tester chaque élément de la liste. Compte tenu du fait que la croissance de tailles des bases de données de la **série 1** est effectuée en augmentant le nombre de propriétés initialisées par classe, le nombre d'éléments de la liste des propriétés des classes augmente avec la base de données. Et cela a pour conséquence une augmentation du temps de recherche des classes qui utilisent une propriété. Il n'en reste pas moins vrai que lorsqu'on recherche au moins 3 propriétés les deux approches font jeu égal.

Les requêtes ne renvoient que l'identifiant (OID) des instances. Ces requêtes ne sont donc réalistes, puisqu'en pratique, on veut en plus des identifiants, les valeurs d'un certain nombre de

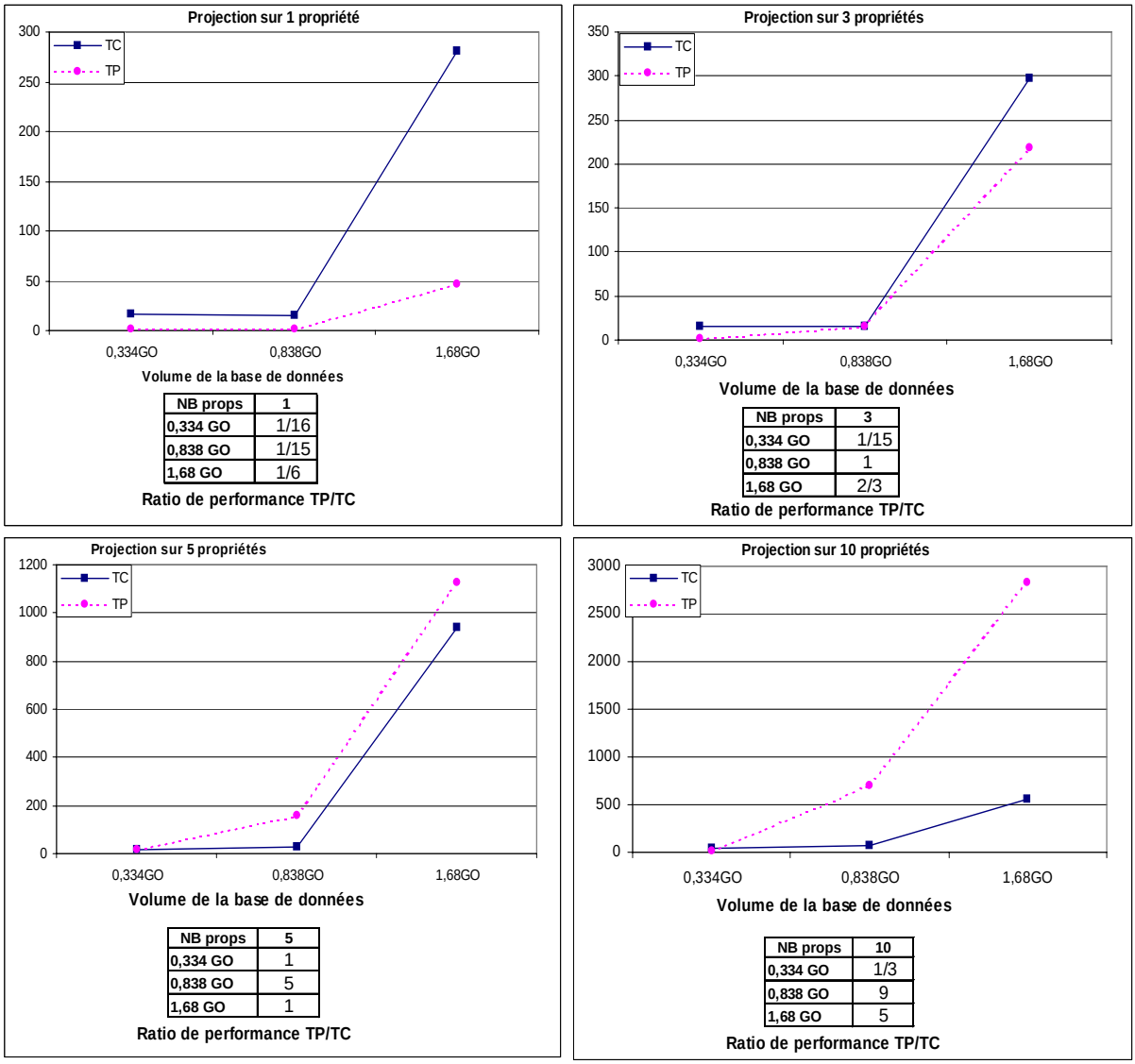


FIG. 5.23 – Performance des requêtes non typées en faisant varier le nombre de propriétés projetées.

propriétés des instances récupérées. Les courbes de la figure 5.23 présentent les performances des approches *TP* et *TC* lorsque l'on se situe dans ce cas. Nous faisons varier le nombre de propriétés retournées de 1 à 10.

On peut remarquer sur les figures que l'approche *TP* est plus performante que l'approche *TC* pour 1 à 3 propriétés. Mais à partir de 5 propriétés projetées, l'approche *TC* devient plus performante. L'explication est simplement dû au fait que chaque propriété retournée dans la requête se traduit par une jointure entre les tables de propriétés. Et plus il y a des propriétés (et donc des jointures) plus la requête est coûteuse. Au contraire, les requêtes de projection sur approche *TC* sont relativement constantes quel que soit le nombre de propriétés retournées. Ceci fait qu'au bout d'un certain nombre de propriétés, l'approche *TC* (5 dans notre cas) devient plus performante que l'approche *TP*.

1.5.2 Conclusion des requêtes non typées

Comme on pouvait s'y attendre, l'approche *TC* s'avère moins efficace que les approches *TP* et *Triplets* sur les requêtes consistant à récupérer l'identifiant des instances à partir d'une sélection sur une seule valeur de propriété, et ce indépendamment des classes concernées. Néanmoins dès que la requête devient plus réaliste, i.e. nécessitant par exemple de récupérer un certain niveau de description des instances sélectionnées, l'approche *TC* fait jeu égal ou devient plus efficace que l'approche *TP*.

Il est clair que pour ce type de requête, peu fréquente en général, une approche *TP* est mieux adaptée. Ainsi par exemple, si l'on souhaite offrir un accès direct aux instances, à partir d'un identifiant, cette propriété particulière devrait faire l'objet d'une représentation particulière sous forme d'une table ternaire (id, oid, classe). Ainsi, dans nos exemples, dès que l'on souhaite récupérer plus de 5 valeurs de propriétés par instance, l'approche *TC* devient plus efficace.

Dans la section qui suit, nous présentons les tests des requêtes de modification des bases de données.

1.6 Évaluation des performances des approches de représentation sur les requêtes de modification

Les requêtes de modification sont les requêtes qui visent à modifier la base de données. Ces requêtes sont constituées des requêtes d'insertion (INSERT INTO), de suppression (DELETE FROM) des instances et de modification de valeurs de propriétés des instances. Nous nous intéresserons uniquement aux requêtes d'insertion et de mises à jour des valeurs des propriétés.

1.6.1 Requête d'insertion d'une instance

Il s'agit donc ici de tester les temps de réponse des requêtes d'insertion des instances dans la base de données. Nous effectuons deux tests :

- un sur des bases de données ayant un même nombre d'instances et un nombre croissant de propriétés par classe, et

– un deuxième sur des bases de données ayant un même nombre de propriétés et un nombre croissant d'instances par classe.

Le premier test nous permettra de mesurer l'incidence du nombre de propriétés sur les différentes approches et le deuxième test nous permettra de mesurer l'incidence du nombre d'instances sur les requêtes d'insertion. Les résultats de ces tests sont donnés respectivement dans les figures 5.24 et 5.25.

Requêtes selon les approches

Voici un exemple de l'expression des requêtes selon les différentes approches :

– **Approche *table par classe*** :
  INSERT INTO Salarie (id,nom,age) **VALUES**(100,"toto",25) ;
– **Approche *table par propriété*** :
  INSERT INTO Salarie (id) **VALUES**(100) ;
  INSERT INTO nom (id,value) **VALUES**(100,"toto") ;
  INSERT INTO age (id,value) **VALUES**(100,25) ;
– **Approche *triplets*** :
  INSERT INTO Salarie (id) **VALUES**(100) ;
  INSERT INTO triplets (sujet,prédicat,objet) **VALUES**(100,"nom","toto") ;
  INSERT INTO triplets (sujet,prédicat,objet) **VALUES**(100,"age",25) ;

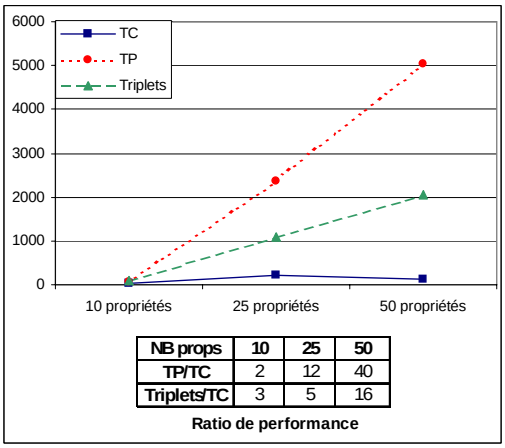


FIG. 5.24 – Insertion d'une instance dans une classe sur des bases de données à nombre de propriétés croissant.

1.6.1.1 Tests d'incidence du nombre de propriétés

Les résultats sont donnés dans la figure 5.24. On peut remarquer sur la figure que :

1. quel que soit le nombre de propriétés, les temps de réponse des requêtes d'insertion sur approche *TC* restent stables et inférieurs aux temps des approches *TP* et *triplets*.
2. les ratios de performances sont assez faibles lorsque les classes contiennent très peu de propriétés et augmentent fortement avec le nombre de propriétés des instances insérées.
3. La mauvaise performance apparente de l'approche *TP* sur l'approche *triplets* se justifie par les n accès aux tables des propriétés nécessaires dans l'approche *TP* tandis que dans

l'approche *triplets*, l'ajout des tuples s'effectue sur une unique table : *triples*. Mais ceci n'est qu'apparent. En effet, après toute insertion de nouvelle instance dans l'approche *triplets*, la table *triples* nécessite *d'être clustérisée sur sa colonne prédicat* si l'on veut que ses performances restent comparables à celles de l'approche *TP*. Cette opération de clustérisation est assez longue. A titre indicatif, la clustérisation de la table *triples* sur la base de données : *BD_10P_1K* s'effectue à environ **3 min 30 s**. Ceci montre que l'approche *triplets* est pratiquement inutilisable si la base de données doit faire l'objet de mises à jour fréquentes.

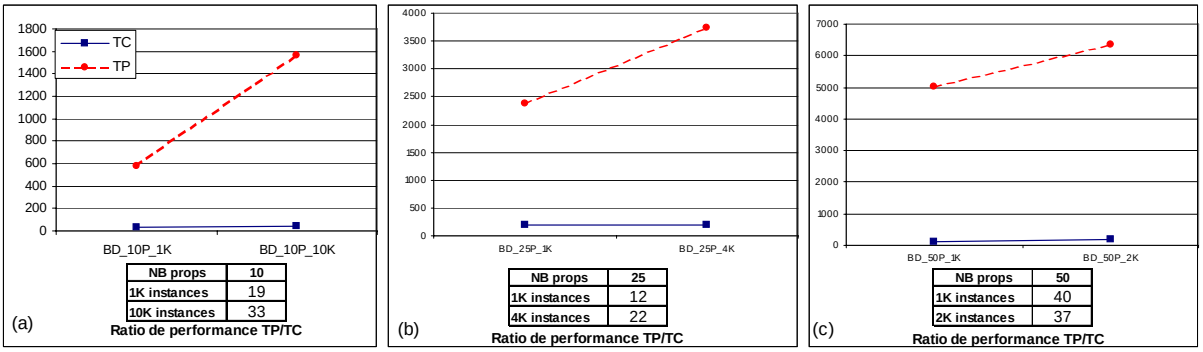


FIG. 5.25 – Insertion d'une instance dans bases de données ayant même nombre de propriétés par classe

Dans la section suivante nous faisons des tests d'insertion en faisant varier cette fois le nombre d'instances et en maintenant constant le nombre de propriétés, pour les seules approches *TP* et *TC*.

1.6.1.2 Tests d'incidence du nombre d'instances

Les résultats de ces tests sont présentées sur les figures 5.25a, 5.25b, 5.25c où nous comparons respectivement les temps de réponse des requêtes d'insertion sur les bases de données (BD_10P_1K, BD_10P_10K), (BD_25P_1K, BD_25P_4K), (BD_50P_1K, BD_50P_2K). On remarque que chaque paire de bases de données initialise le même nombre de propriétés, mais possède un nombre différent d'instances par classe.

1. On constate sur chaque figure que l'approche *TC* est encore beaucoup moins sensible que l'approche *TP* à l'effet de taille et supporte beaucoup mieux le passage à l'échelle.

2. La variation de la taille des bases des données sur chacune des courbes des figures 5.25a, 5.25b, et 5.25c montre que l'approche *TC* n'est que très peu sensible au nombre d'instances par classe dans les bases de données contrairement à l'approche *TP*. La légère croissance des temps d'exécution dans le cas *TC* peut se justifier par le temps de mises à jour des tables des index dans les tables qui ont beaucoup d'instances. En effet, compte tenu de l'augmentation du nombre d'instances par classe dans la base de données, la table des in-

dex des colonnes augmente également.

Après ces tests sur les requêtes d'insertion, nous faisons dans la section suivante des tests sur des requêtes de mises à jour des valeurs des propriétés des instances.

1.6.2 Requêtes de changement de valeurs d'une propriété d'une instance

Nous mesurons le temps de réponses des requêtes de modification de la valeur d'une propriété d'une instance connaissant son *oid*.

Requêtes selon les approches

Voici un exemple de l'expression des requêtes selon les différentes approches :

- **Approche *table par classe***
  UPDATE Salarie **SET** nom = "titi" **WHERE** id =100
- **Approche *table par propriété***
  UPDATE nom **SET** value = "titi" **WHERE** id =100
- **Approche *triplets***
  UPDATE triplets **SET** objet = "titi" **WHERE** prédicat = "nom" **AND** sujet =100

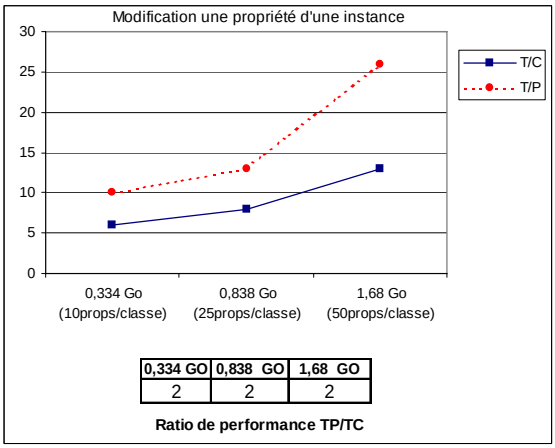


FIG. 5.26 – modification de la valeur de propriété d'une instance

Tests

Les résultats de nos tests sont présentés dans la figure 5.26, on remarque que les temps de réponse des approches *TC* sont légèrement meilleurs par rapport à *TP* avec des ratios de performance assez faibles lorsque l'on ne met à jour qu'une seule propriété.

Ceci s'explique simplement par le fait que les tables des propriétés contiennent beaucoup de tuples ($\|C_{P_i}\| \times \|I_{C_i}\|$) par rapport aux tables des classes ($\|I_{C_i}\|$). Ce qui fait que la recherche du tuple (id, value) de la valeur de la propriété de l'instance à modifier est plus longue dans l'approche *TP* que dans l'approche *TC*.

Nos tests n'ont été réalisés que sur une seule propriété. Il est clair que si on devait modifier plus d'une propriété en même temps (Exemple : UPDATE C_i SET $P_1 = X_1$, ..., $P_i = X_i$),

les temps de réponse de l'approche *TP* seraient plus longs que l'approche *TC*, puisqu'elle nécessiterait de modifier la valeur des propriétés dans différentes tables, alors que dans l'approche *TC*, elle se réaliserait sur la seule table de la classe et sur des valeurs de propriétés qui seraient *potentiellement* stockées dans le même bloc mémoire.

1.6.3 Conclusion des tests de modification

A partir des tests des requêtes de modification, nous pouvons faire les constats suivants :
- l'approche *triplets* est pratiquement inutilisable dès lors que des modifications interviennent puisque cela nécessite de ré-clustériser toute la table *triplets* ce qui exige un temps extrêmement long (3min30s sur la plus petite des bases de données).
- l'approche *TC* offre des performances toujours préférables à l'approche *TP*,
- les ratios de performances *TP/TC* augmentent considérablement lorsque la taille des bases de données augmente, et tout particulièrement lorsque le nombre de propriétés par instances augmente.

Après cette évaluation de la partie *données* par comparaison avec les approches précédemment proposées dans la littérature, nous donnons dans la section suivante, quelques indications sur les performances de la partie *ontologie* de notre architecture.

2 Évaluation des performances de la partie *ontologie*

Nous donnons dans cette section quelques indications sur les performances d'accès aux ontologies représentées dans la partie *ontologie* de notre architecture de BDBO. Les tests que nous avons réalisés consistent à mesurer le temps d'affichage des classes à différents niveaux de la hiérarchie des classes et pour un nombre variable de propriétés décrites. L'objectif de ces tests est de mesurer l'incidence du nombre de classes et de propriétés sur les performances de notre architecture et de vérifier pour quelles tailles d'ontologies un parcours interactif et visuel de la structure ontologique reste possible.

2.1 Description du banc d'essai

Nous effectuons nos tests sur trois ontologies de différentes tailles :
- l'ontologie normalisée *IEC 61360-4* composée de 190 classes et 1026 propriétés. La moyenne des classes des hiérarchies est de 5,
- l'ontologie *LMPR* définie par RENAULT pour sa propre base de données de composants est constituée de 295 classes et 509 propriétés. La profondeur de la hiérarchie des classes varie de 2 à 5,
- l'ontologie *fixation* composée de 36 classes et de 23 propriétés et la moyenne est de 3.

Sur ces trois ontologies stockées dans des bases de données différentes, nous effectuerons les tests suivants :

-Test 1. L'affichage de la description d'une classe (racine ou intermédiaire ou feuille) des ontologies (sans leurs propriétés).

	IEC	LMPR	Fixation
Nombre de classes	190	295	36
Nombre de propriétés	1026	509	23
Profondeur hiérarchie	5	2	3
Nombre propriétés à la classe racine	1026	13	21

TAB. 5.1 – Caractéristiques des ontologies sur lesquelles se baseront les tests.

-Test 2. L'affichage de la description d'une classe racine des ontologies avec toutes leurs propriétés.

-Test 3. La récupération des sous-classes d'une classe

-Test 4. L'affichage des propriétés des classes en faisant varier leur nombre de propriétés caractéristiques.

Pour chacun de ces tests, nous effectuerons deux évaluations : une première *à froid* i.e., buffer de la base de données vidée et *à chaud* i.e., buffer initialisé. Ces deux évaluations confirmeront en particulier l'intérêt d'utiliser un cache pour améliorer les performances des applications qui accèdent aux BDBOs, comme nous l'avons mentionné pour la fonction F_3 (API pour la programmation orienté-objet) dans le chapitre 3 section 2.3.

2.2 Évaluation

Les résultats des différents tests que nous avons effectués sont résumés dans les tables 5.2 et 5.3. Tous les temps reportés sur ces tables sont en millisecondes (ms).

	IEC		LMPR		Fixation	
	à froid	à chaud	à froid	à chaud	à froid	à chaud
Test 1- Affichage de la description d'une classe sans ses propriétés	80	16	80	16	62	16
Test 2- Affichage de la description d'une classe racine avec ses propriétés	25111	1376	346	16	406	16
Test 3- Calcul des sous-classes d'une classe (22)	47	8	47	8	45	7

TAB. 5.2 – Tests sur les classes des BDBOs (Test 1, Test 2, Test 3).

	IEC		LMPR		Fixation	
	à froid	à chaud	à froid	à chaud	à froid	à chaud
1 propriété	172	0	47	0	16	0
5 propriétés	250	16	141	16	125	0
20 propriétés	656	16	453	16	375	16
1026 propriétés	25031	1359	-	-	-	-

TAB. 5.3 – Test 4- Variation du nombre de propriétés à récupérer de la BDBO

Les résultats de tous ces tests, nous montrent que les temps de réponse des tests réalisés *à froid* sont très supérieurs aux tests réalisés *à chaud*. Cela montre l'intérêt des caches de données

(au niveau du SGBD et au niveau de l'outil PLIBEditor) qui conservent le résultat des requêtes précédemment exécutées. Les requêtes ré-exécutées ne nécessitent plus d'accéder à la base de données. D'où les temps largement inférieurs pour les requêtes réalisées *à chaud*.

Les résultats du **Test 1**, nous montrent que quel que soit le nombre de classes dans l'ontologie, le temps de calcul de la description et d'affichage d'une classe (sans ses propriétés) reste à peu près constant, et tout à fait compatible avec les critères d'interactivité. Les résultats du **Test 2** montrent que les propriétés des classes ont un grand impact sur l'affichage des classes. En particulier, pour l'ontologie IEC 61360-4 qui est mal structurée (toutes les propriétés sont définies à la racine, puis rendues applicables de façon sélective et beaucoup plus bas) le temps d'affichage de la racine est extrêmement long. (de 1,3 secondes à chaud et 25 secondes à froid). Par contre, la navigation devient ensuite très fluide. Les résultats du **Test3**, nous montre que quelle que soit la BDBO, le temps de calcul des sous-classes d'une classe reste identique et faible.

Les tests sur les propriétés à afficher (**Test 4**) sur la table 5.3, nous montre que :
- la variation du nombre de propriétés occasionne une augmentation conséquente du temps d'affichage des propriétés.
- Pour un même nombre de propriétés à afficher, le temps d'affichage croît avec le nombre de propriétés définies dans les ontologies.

Cette forte croissance des temps d'accès à l'ontologie pour les ontologies de grandes tailles résulte du fait que la récupération de la description (i.e. l'ensemble des attributs) d'une propriété nécessite d'accéder à plus d'une dizaine de tables et de réaliser de nombreuses jointures. Cela est dû à la structure des tables de la partie *ontologie* qui est directement déduite du modèle d'ontologie PLIB à partir des règles de correspondances EXPRESS que nous avons définies (cf. annexe C). Ce mapping, bien que simple, présente l'inconvénient de générer beaucoup de tables. Aussi l'accès aux données demande beaucoup de jointures et donc un temps important. Ceci milite pour essayer de simplifier le modèle objet PLIB avant de le transformer, de façon automatique en un schéma relationnel.

2.3 Bilan

Nos tests sur la partie *ontologie* nous ont permis de tirer les conclusions suivantes :
- la navigation reste fluide pour un nombre raisonnable de classes et de propriétés stockées dans les BDBOs.
- l'affichage devient néanmoins lent s'il y a un grand nombre de propriétés.

Cela nous a donc conduit à travailler sur une nouvelle approche pour la définition du schéma relationnel de la partie *ontologie* de notre architecture de BDBO. Cette approche développée, dans le cadre d'une collaboration avec d'autres chercheurs du laboratoire visait essentiellement à améliorer l'efficacité du schéma relationnel généré. Cette approche s'est basée sur l'exploitation fine des contraintes d'intégrité existant au niveau des instances de chaque modèle objet pour transformer de façon systématique un modèle objet avant de définir sa représentation relation-

nelle. Par contrainte de niveau instance, nous signifions les contraintes précises que l'on peut associer à des ensembles d'instances mais qui sont en général masquées par les formalismes d'expressions de contraintes utilisés [128].

La nouvelle approche qui a été développée se base essentiellement sur deux types de telles contraintes. La contrainte *used_ once* (qui correspond pour partie, en UML 2.0, au losange noir) permet de mettre en évidence une relation de composition d'une classe à l'intérieur de plusieurs classes composites. Chaque composant peut alors se représenter à l'intérieur du composite pertinent, ce qui supprime des relations de compositions, et donc des jointures ultérieures. La *redéfinition de type* (qui consiste à redéfinir par restriction, lors d'une spécialisation d'une classe, le type d'un attribut hérité) permet de supprimer des classes abstraites tout en conservant la possibilité de représentation du polymorphisme. Cet élagage de la hiérarchie supprime également des jointures ultérieures.

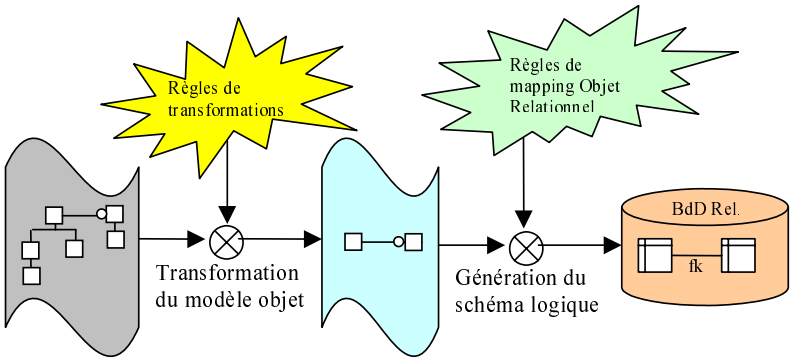


FIG. 5.27 – Nouvelle approche de représentation du schéma logique de la partie ontologie de notre architecture

Cette approche, qui a permis de réduire de façon très importante le nombre de tables nécessaires pour représenter une ontologie PLIB, est destinée à être implantée dans une version future de l'architecture *OntoDB*.

3 Conclusion

Nous avons dans ce chapitre fait une évaluation de performance de notre architecture de base de données à base ontologique. Notre évaluation s'est principalement portée sur les choix fait pour la représentation des données à base ontologique dans la partie *données* de notre architecture. En effet, il existe dans ce domaine d'autres propositions avec lesquelles il convenait de se comparer. Peu de comparaisons ont par contre été effectuées sur les ontologies car les schémas d'implantation des ontologies dépendent étroitement du modèle d'ontologie ciblé, et, de ce point de vue là, les différents systèmes disponibles ciblent des modèles différents.

Pour notre évaluation de la partie *données*, nous avons comparé les deux approches préexis-

tantes connues pour être les plus efficaces (*table par propriété* et *triplets*) à l'approche *table par classe* proposée dans le cadre de notre thèse. Des études antérieures ont montré que l'approche *table universelle* n'était en effet pas compétitive.

Nos tests ont porté sur trois familles de requêtes : (1) requêtes typées dans lesquelles l'utilisateur connaît la ou les classes dans lesquelles il effectue sa recherche, (2) requêtes non typées dans lesquelles l'utilisateur recherche sur l'ensemble de la population à la base de données, et (3) les requêtes de modification. Ces requêtes ont été réalisées sur différentes bases de données que nous avons générées à partir d'exemples réels appartenant à notre domaine d'application cible. Pour cela nous avons développé un générateur qui à partir de n'importe quelle ontologie fournie en paramètre, génère des populations d'instances pour les différentes classes de l'ontologie.

Les tests des requêtes typées (sélection, projection, et jointure sur des classes feuilles ou non) ont montré que l'approche *table par classe* était toujours plus performante que l'approche *table par propriété* elle-même plus performante que l'approche *triplets*. Le résultat le plus marquant est que les temps d'exécution des requêtes portant sur l'approche *TC* sont relativement stables aussi bien lorsqu'on fait varier le nombre de propriétés impliquées dans les requêtes (propriétés projetées ou dans le prédicat de sélection) que lorsqu'on faisait varié le nombre d'instances des classes, alors que dans l'approche *table par propriété*, ces temps croissent très fortement, de sorte que l'approche *TC* supporte beaucoup mieux le passage à l'échelle que l'approche *TP*.

Les tests sur les *requêtes non typées* ont montré un résultat plus nuancé. L'approche *table par propriété* est plus performante tant que très peu de propriétés sont concernées par les requêtes (1 à 3). Dès que le nombre de propriétés utilisées soit dans la sélection soit dans la jointure atteint un nombre de l'ordre de 4, les deux approches font jeu égal, puis l'approche *TC* devient de plus en plus préférable.

Enfin les tests des requêtes de modification ont montré que l'approche *table par classe* était toujours plus performante que l'approche *table par propriété*. Ils ont également montré que l'approche *triplet* était pratiquement inutilisable dès lors que les mises à jours intervenaient. En effet, la table unique demandait alors à être trié (clusterisée) par valeur de prédicat afin de garder des performances acceptables en recherche. Et un tel tri exige toujours des temps rédhibitoires (plus de 3 minutes pour nos plus petites bases de données).

De l'ensemble de ces tests, il résulte clairement que l'approche *table par classe* que nous avons proposée est pratiquement toujours plus efficace, voire beaucoup plus efficace pour les bases de données de taille significative, que les approches pré-existantes. Sa réelle limite réside dans les hypothèses que nous avons définies pour la mettre en œuvre, à savoir le *typage fort* des propriétés et la mono-instanciation des classes. Le typage fort des propriétés est une hypothèse également faite par beaucoup d'autres chercheurs. Elle ne nous parait guère limitative. La *mono-instanciation* par contre peut faire débat, car elle impose un style de modélisation où la *multi-instanciation* est représentée soit par l'héritage multiple, soit par la technique de l'agrégat d'instances. Quoi qu'il en soit, dans beaucoup de domaines, et en particulier dans notre domaine

cible, la *multi-instanciation* ne fait pas partie des techniques de modélisation utilisées.

Pour notre évaluation de la partie *ontologie*, nous avons, au moyen d'ontologies de tailles différentes, mesuré le temps de navigation à travers l'ontologie dans l'outil PLIBEditor. Les tests nous ont révélé des temps de réponse acceptables pour des ontologies de taille usuelle. Pour les grandes ontologies, les temps de réponse étaient assez considérables. Cette médiocre performance pour les grandes ontologies est due à l'approche de traduction systématique du modèle objet en modèle logique que nous avons adoptée. En effet celle-ci engendre de nombreuses tables, causant ainsi de nombreuses jointures pour la récupération des ontologies. Pour pallier à ce défaut, une nouvelle approche basée sur la transformation de modèle a été développée dans le laboratoire. Elle devrait être implantée dans une prochaine version d'*OntoDB*.

Conclusion et perspectives

Conclusion

Cette thèse avait un double objectif :

- du point de vue des bases de données, il s'agissait d'étudier la possibilité d'associer aux données contenues dans une base de données le schéma qui en définit la structure et l'ontologie qui en définit la signification,
- du point de vue de la représentation de connaissance, il s'agissait de définir une architecture efficace permettant (1) la gestion simultanée des ontologies et des données à base ontologique, (2) le passage à l'échelle et (3) la gestion des données de grande taille.

La principale difficulté qui empêchait la réalisation des bases de données à base ontologique susceptibles de supporter le passage à l'échelle était le caractère non typé des données à base ontologique. Dans toutes les approches existantes, souvent basées sur des données exprimées dans des formalismes de type XML Schema, chaque instance est considérée comme porteuse de sa propre structure. Elle appartient à un nombre quelconque de classes déconnectées et elle est décrite par des ensembles quelconques de propriétés. Pour des données ayant une telle structure, seuls des modèles de type décomposé peuvent être utilisés. Et ces modèles supportent mal le passage à l'échelle dès lors que chaque instance est décrite par un nombre significatif de valeurs de propriétés.

En fait, même si, théoriquement, dans des langages tels que OWL, les propriétés peuvent n'avoir ni domaine et/ou co-domaine bien défini, il semble de plus en plus rare de rencontrer des ontologies où toutes les propriétés ne sont pas fortement typées. Ceci est même souvent considéré comme un élément essentiel de leur définition. Concernant la multi-instanciation, même si celle-ci est effectivement utilisée dans certain cas, d'une part, il existe des domaines importants où cette multi-instanciation est interdite. C'est le cas par exemple dans le domaine technique lorsque le modèle d'ontologie PLIB est utilisé. C'est également le cas lorsqu'on utilise l'éditeur d'ontologie Protégé, qui est probablement l'éditeur le plus utilisé au niveau mondial et qui ne supporte pas non plus la multi-instanciation. D'autre part, la multi-instanciation peut être très souvent remplacée par la technique de l'agrégat d'instances où un objet du monde réel est représenté par un ensemble d'instances, chacune correspondant à un point de vue différent.

Ces considérations nous ont amené à restreindre le champ des données à base ontologique auquel nous nous intéressions en leur imposant deux conditions :

- chaque instance doit avoir une classe de base qui est le minorant unique des classes auxquelles elle appartient pour la relation de subsomption,
- chaque propriété doit être fortement typée : tant la classe qui définit le domaine sur lequel elle est définie, que son co-domaine de valeurs doivent être uniques et bien définis.

A partir de ces hypothèses, il devient possible d'associer à chaque classe un "schéma enveloppe" susceptible de contenir toutes ses instances. Ce schéma contient, au maximum, l'ensemble des propriétés applicables à cette classe.

Le modèle d'architecture que nous avons alors proposé pour les bases de données à base ontologique, appelé*OntoDB*, est constitué de quatre parties.

- La partie *données* correspond à la partie classique des bases de données et permet de représenter les données à base ontologique. Les données de chaque classe définie au niveau de l'ontologie, y sont représentées dans une relation. Cette relation peut être être une simple table, ou une vue constituée de plusieurs tables, mais chaque attribut de cette relation correspond à une propriété définie, au niveau ontologique, comme applicable à la classe correspondant à la relation. En fait, les attributs de cette relation sont constitués d'un sous-ensemble des propriétés applicables de la classe, à savoir l'union des propriétés initialisées pour au moins une instance de la classe. Certes, cette relation peut comporter un certain nombre de valeurs nulles, mais l'expérience montre que très fréquemment, tout particulièrement dans le domaine technique, mais aussi dans celui du Web sémantique, la description des différentes instances d'une classe est en général assez homogène. Notons que la partie *données* de notre architecture correspond au niveau M_0 de l'architecture du MOF.

- La partie *méta-base* est également une partie classique des bases de données. Elle permet de représenter la structure des tables et des attributs de la base de données.

- La partie *ontologie* constitue la première partie spécifique des BDBOs. Elle permet de représenter les ontologies sous forme de données d'un méta-modèle. En effet, toutes les descriptions usuelles des ontologies sont réalisées par l'intermédiaire de méta-modèles. Les ontologies sont représentées dans cette partie à l'aide d'une structure de tables image relationnelle du méta-modèle d'ontologie utilisé. Cette partie peut également contenir les relations existant entre l'ontologie locale et une éventuelle ontologie partagée préexistante. Notons que la partie *ontologie* de notre architecture correspond au niveau M_1 du MOF.

- La partie *méta-schéma* est, quant à elle, spécifique de notre architecture. Elle permet de représenter le méta-modèle d'ontologie utilisé dans la base de données sous forme d'instances d'un méta-méta-modèle. Elle vise à permettre que notre architecture supporte les changements ou évolutions du méta-modèle d'ontologie. Notre *méta-schéma* a la particularité d'être réflexif, i.e., il s'auto-représente dans la base de données. Un autre intérêt majeur de cette partie de notre architecture est qu'elle permet d'offrir une interface générique i.e., qui est indépendante de tout modèle d'ontologie particulier. Notons que la partie *méta-schéma* correspond au niveau M_2 et M_3 du MOF.

Cette architecture répond aux deux objectifs définis pour notre travail :

- du point de vue des bases de données, elle permet d'expliciter la sémantique d'une base de données existante en lui adjoignant les deux parties *ontologie* et *méta-schéma*, puis en ajoutant, si besoin est dans la partie *données* des vues correspondant à la population de chacune des classes ontologiques ;
- du point de vue de la représentation de connaissances, elle permet de représenter des données à base ontologique sous une forme usuelle des bases de données, ce qui la rend susceptibles de bénéficier de toutes les techniques d'indexation et d'optimisation propres aux bases de données.

Notons que cette architecture permet de laisser une grande indépendance entre les parties *ontologie* et *données*. En effet, l'ontologie ne définit ni les classes et les propriétés qui sont effectivement valuées dans la partie *données*, ni la structure de tables représentant les relations associées aux classes effectivement valuées. Par exemple, lorsqu'une BDBO de type OntoDB est créée, après la définition ou l'importation de l'ontologie, le choix des classes et propriétés à initialiser est laissé à la charge du concepteur de la base de données. Celui-ci sélectionne un sous-ensemble de classes et de propriétés pour composer le modèle conceptuel adapté à son application. Notons qu'il a le choix également de sélectionner les propriétés qui formeront la clé primaire des instances des classes. Le modèle conceptuel ainsi construit est lui-même représenté dans la partie *données* et peut être exploité par des programmes.

L'architecture *OntoDB* a été validée opérationnellement par un prototype du système de gestion de BDBO sur le SGBD relationnel objet PostgreSQL. Nous avons décrit dans cette thèse comment, en utilisant les techniques d'ingénierie dirigée les modèles, nous avons généré automatiquement tous les composants nécessaires pour le déploiement de notre prototype de système de gestion de BDBO.

Une particularité supplémentaire de notre prototype est qu'il est le premier, à notre connaissance, à offrir la possibilité de versionner les ontologies et de représenter le cycle de vie des données à base ontologique.

Nous avons également évalué les performances de notre approche de représentation des données à base ontologique à partir de jeux d'essai conçus sur la base d'une ontologie réelle et normalisée relevant de notre domaine d'application cible. Notre approche s'est avérée plus performante et supportant beaucoup mieux le passage à l'échelle que les deux autres approches existantes de représentation à savoir les approches *triples* et *table par propriété* dans les catégories de requêtes pour lesquelles on connaît les classes à interroger. Ce sont les requêtes les plus fréquentes. Pour les requêtes dites *non typées* i.e., les requêtes dans lesquelles les classes à interroger ne sont pas connues, notre approche est seulement moins performante pour les requêtes impliquant un très petit nombre de propriétés.

L'architecture de BDBO que nous avons décrite dans cette thèse présente les avantages suivants :

- Les performances de stockage, sécurité, d'intégrité et de requêtes des SGBDRs et SGBDROs sont conservées. Cela est simplement dû au fait que notre modèle d'architecture de BDBO est greffé sur ces systèmes et que les données sont gérées selon une algèbre relationnelle.

- L'accès aux données est offert au niveau sémantique, et ce sans aucune programmation spécifique. La présence de l'ontologie dans la base de données permet en effet de réaliser des requêtes sur les données en termes d'ontologies, indépendamment du modèle logique. Un langage de requêtes spécifique est d'ailleurs en fin de développement au laboratoire. Un éditeur graphique permet également d'explorer tout le contenu de la base de données au niveau ontologique, sans connaitre a priori le schéma effectif des données.

- L'ontologie est elle-même interrogeable simultanément avec les données. C'est une des grandes originalités des BDBOs. En effet, on peut effectuer des requêtes du type :"Je veux toutes les instances de la classe (générique) '*vis*' dont la propriété *longueur* est inférieure à *12 millimètres*, dont la propriété *diamètre* est égale à *3 millimètres*, ainsi que, pour chaque instance, le *nom* en français de sa classe d'appartenance".

- La représentation explicite dans la base non seulement de la sémantique des données à travers l'ontologie locale mais également de l'articulation de celle-ci, par subsomption, avec une éventuelle ontologie partagée prépare chaque source de données pour une future intégration avec d'autres sources de données référençant la même ontologie partagée. Ceci débouche sur une nouvelle approche, qualifiée de "*a priori*", permettant l'intégration automatique de sources de données hétérogènes, ayant chacune leur propre ontologie articulée sur une (ou plusieurs) ontologies globales [18].

- Dans une BDBO, le modèle logique ne pouvant être créé qu'à partir de l'ontologie, ces deux éléments restent nécessairement cohérents tout au long de l'évolution du modèle logique des données, ceci n'est pas souvent le cas dans les approches classiques, entre le modèle conceptuel et le schéma logique des données.

- Enfin, lorsque des ontologies PLIB sont utilisées, la structure proposée apparaît particulièrement bien adaptée à la fois pour des serveurs de commerce électroniques professionnels et pour les bases de données de composants industriels.

Pour finir, notons que notre travail, modèle et prototype, est au cœur de nombreux travaux en cours au sein de l'équipe d'ingénierie de données du LISI notamment pour la proposition d'un langage de requêtes pour les BDBOs, la proposition d'une nouvelle approche d'intégration automatique de sources de données hétérogènes basées sur les BDBOs et la proposition d'un modèle d'architecture pour les applications de commerce électronique.

Perspectives

Nos travaux offrent de nombreuses perspectives. Ces perspectives peuvent être classées en deux catégories : optimisations et évolutions.

Optimisations

Les tests de performance de la partie *ontologie* de notre prototype, ont mis en évidence des temps de réponse élevés pour les ontologies contenant beaucoup de classes et propriétés. Ces temps de réponse s'expliquent par la complexité du schéma logique de la partie *ontologie*. Celui-ci est composé de nombreuses tables, en majorité des tables d'aiguillages, impliquant, pour certain accès, de nombreuses jointures. Pour améliorer la performance de notre prototype *OntoDB*, une nouvelle approche de représentation des ontologies est actuellement explorée. Elle consiste à utiliser les techniques de transformation de modèles afin de simplifier le modèle d'ontologie PLIB avant de le traduire en relationnel. Cette simplification permet d'aplatir les hiérarchies d'entités et/ou de fusionner des entités liées par des relations de composition ou des agrégations en exploitant certaines contraintes d'intégrités du modèle. Ces transformations devraient améliorer considérablement les performances de la partie *ontologie* de notre architecture.

Évolutions

Les types de connaissances représentables dans une BDBO dépendent des caractéristiques du modèle d'ontologie utilisé et de la diversité des liens pouvant être établis, d'une part, entre le niveau conceptuel et le niveau logique et, d'autre part, entre ontologie locale et ontologie partagée.

Concernant le modèle d'ontologie utilisé, nous envisageons d'augmenter son pouvoir d'expression en y représentant :

- **les dépendances fonctionnelles entre les propriétés caractéristiques des classes.** Cela permettra par la suite de normaliser les schémas logiques des différentes classes.

- **les dépendances algébriques entre valeurs de propriétés**. Il s'agira ici, d'exprimer les relations mathématiques existantes entre les propriétés des classes et qui nous permettrons de calculer les valeurs des unes par rapport aux autres dynamiquement.
  Exemple : On pourra ainsi donc exprimer le *rayon* et le *diamètre* de la façon suivante :
  - diamètre := rayon x 2 (diamètre $\longmapsto$ rayon) ;
  - rayon := diamètre / 2 (rayon $\longmapsto$ diamètre) ;

- **Certains opérateurs d'équivalence entre classes**. Il s'agira ici d'introduire certains des constructeurs des logiques de description s'appliquant sur les classes. Ce sont entre autres les constructeurs d'union, d'intersection, de complémentarité et de restriction de classes. Ces constructeurs de classes impliqueront que nous définissions des mécanismes pour le calcul des instances des classes construites par ces constructions. Ceci sera réalisé par l'intermédiaire de vues SQL.

Soulignons que ces deux dernières extensions consistent à passer d'ontologies canoniques à des ontologies non canoniques. Une partie du traitement des requêtes sera alors effectuée de façon intentionnelle, au niveau ontologique, avant d'être traitées de façon extensionnelle au niveau de la partie *données*.

Concernant les rapports entre niveau conceptuel et logique au sein de la base de données, un grand nombre de contraintes d'intégrités, dont les contraintes d'unicité et de cardinalité, sont souvent de nature ontologique. Nous envisageons d'exploiter automatiquement toutes les contraintes définies au niveau ontologique pour générer les contraintes d'intégrités au niveau de la base de données.

Bibliographie

[1] Dictionnaire en ligne collaboratif dico du net. Technical report, Available at `http://www.dicodunet.com/`.

[2] I. 10303.14. Product data representation and exchange - part 14 :express-x language reference manual. *ISO*, (088), Oct 1999.

[3] Agile. Mapping objects to relational databases : O/R mapping in detail. *Available at* `http://www.agiledata.org/essays/mappingObjects.html`.

[4] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *SIGMOD Inter. Conf. On Manag. Of Data*, pages 253–262, 1989.

[5] R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *VLDB*, pages 149–158, 2001.

[6] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite : Managing voluminous RDF description bases. In *SemWeb*, 2001.

[7] J. Almeida, R. Dijkman, M. Sinderen, and L. Pires. On the notion of abstract platform in mda development. In *EDOC '04 : Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04)*, pages 253–263. IEEE Computer Society, 2004.

[8] ANSI/SPARC. Study group on data management systems : Interim report. *ACM*, 7(2), 1975.

[9] G. Antoniou and F. van Harmelen. Web ontology language : Owl, 2003.

[10] B. Appukuttan, T. Clark, S. Reddy, L. Tratt, and V. R. A model driven approach to model transformations. In *Model Driven Architecture :Foundations and Applications*, pages 1–12. University of Twente, June 2003.

[11] K. S. Arora, S. Dumpala, and K. Smith. Wcrc : An ansi sparc machine architecture for data base management. In *ISCA '81 : Proceedings of the 8th annual symposium on Computer Architecture*, pages 373–387. IEEE Computer Society Press, 1981.

[12] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. *In Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 223–240, December 1989.

[13] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook : Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[14] B. Bebel, J. Eder, C. Koncilia, T. Morzy, and R. Wrembel. Creation and management of versions in multiversion data warehouse. In *SAC '04 : Proceedings of the 2004 ACM symposium on Applied computing*, pages 717–723, New York, NY, USA, 2004. ACM Press.

[15] L. Bellatreche, D. Nguyen Xuan, G. Pierra, and H. Dehainsala. Contribution of ontology-based data modeling to automatic integration of electronic catalogues within engineering databases. *Computers in Industry Journal*, 57 :711–724, 2006.

[16] L. Bellatreche, G. Pierra, X. Nguyen, and H. Dehainsala. Intégration de sources de données autonomes par articulation a priori d'ontologies. In *Proc. du 23ème congrès Inforsid*, pages 283–298, may 2004.

[17] L. Bellatreche, G. Pierra, D. Nguyen Xuan, H. Dehainsala, and Y. Ait Ameur. An a priori approach for automatic integration of heterogeneous and autonomous databases. *International Conference on Database and Expert Systems Applications (DEXA'04)*, pages 475–485, September 2004.

[18] L. Bellatreche, G. Pierra, D. Xuan, and H. Dehainsala. An automated information integration technique using an ontology-based

database approach. In *Proc. of Concurrent Engineering(CE'2003)*, pages 217–224, 2003.

[19] D. Beneventano and S. Bergamaschi. The momis methodology for integrating heterogeneous data sources. In *IFIP Congress Topical Sessions*, pages 19–24, 2004.

[20] P. Bernstein. Applying model management to classical meta data problems. In *Proceedings of the Conf. on Innovative Database Research (CIDR'03)*, 2003.

[21] P. Bernstein, A. Halevy, and R. Pottinger. A vision for management of complex models. *SIGMOD Rec.*, 29(4) :55–63, 2000.

[22] D. Bitton, D. DeWitt, and D. Turbyfill. Benchmarking database systems a systematic approach. In *VLDB*, pages 8–19, 1983.

[23] B.McBride. Jena : Implementing the rdf model and syntax specification. *Proceedings of the 2nd International Workshop on the Semantic Web.*, 2001.

[24] P. A. Boncz and M. L. Kersten. Mil primitives for querying a fragmented world. *VLDB Journal*, 8(2) :101–119, 1999.

[25] 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 *EC-WEB '02 : Proceedings of the Third International Conference on E-Commerce and Web Technologies*, pages 304–313, London, UK, 2002. Springer-Verlag.

[26] T. Bray, J. Paoli, S.-M. C.M., and M. E. Extensible markup language (xml) 1.0 (second edition). *W3C Recommendation*, oct 2000.

[27] S. Bressan, C. Goh, N. Levina, S. Madnick, A. Shah, and M. Siegel. Context knowledge representation and reasoning in the context interchange system. *Applied Intelligence*, 13(2) :165–180, September 2000.

[28] D. Brickley and R. Guha. Rdf vocabulary description language 1.0 : Rdf schema. *W3C Recommendation (2004)*, February 2004.

[29] J. Broekstra and A. Kampman. Query Language Definition. On-To-Knowledge (IST-1999-10132) Deliverable 9, Aidministrator Nederland b.v., Apr. 2001. See `http://www.ontoknowledge.org/`.

[30] 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 First Internation Semantic Web Conference*, number 2342 in Lecture Notes in Computer Science, pages 54–68. Springer Verlag, July 2002.

[31] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena : implementing the semantic web recommendations. In *WWW Alt. '04 : Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83, New York, NY, USA, 2004. ACM Press.

[32] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *OOPSLA '93 : Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 271–287, New York, NY, USA, 1993. ACM Press.

[33] C. D. Castro, F. Grandi, and M. R. Scalas. Schema versioning for multitemporal relational databases. *Inf. Syst.*, 22(5) :249–290, 1997.

[34] R. G. G. Cattell. *The Object Database Standard – ODMG-93*. Morgan-Kaufmann Publishers, 1996.

[35] P. Chen. The entity-relationship model toward a unified view of data. *ACM Trans. Database Syst.*, 1(1) :9–36, 1976.

[36] V. Christophides, D. Plexousakis, M. Scholl, and S. Tourtounis. On labeling schemes for the semantic web. In *WWW*, pages 544–555, 2003.

[37] J.-Y. Chung, Y.-J. Lin, and D. T. Chang. Object and relational databases. In *OOPSLA '95 : Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum)*, pages 164–169, New York, NY, USA, 1995. ACM Press.

[38] I. Ciorascu, C. Ciorascu, and K. Stoffel. Scalable ontology implementation based on knowler. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003), Workshop on Practical and Scalable Semantic Systems*, Florida, USA, october 2003.

[39] J. Clifford and D. S. Warren. Formal semantics for time in databases. *ACM Trans. Database Syst.*, 8(2) :214–254, 1983.

[40] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6) :377–387, 1970.

[41] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic xml trees. In *PODS '02 : Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 271–281, New York, NY, USA, 2002. ACM Press.

[42] G. Copeland and S. Khoshafian. A decomposition storage model. In *ACM SIGMOD International Conference on Management of Data*, pages 268–279, 1985.

[43] D. Core. The dublin core metadata initiative. *Available at* `http://dublincore.org/`.

[44] M. Dean and Schreiber. Web Ontology Language Reference. *W3C Recommendation (2004)*, February 2004.

[45] H. Dehainsala. Base de données à base ontologique. In *Proc. du 23ème congrès Inforsid*, pages 539–540, may 2004.

[46] H. Dehainsala, S. Jean, X. Nguyen, and G. Pierra. Ingénierie dirigée par les modèles en express : un exemple d'application. In *1ére journée d'Ingénierie dirigée par les modèles*, pages 155–167, 2005.

[47] H. Dehainsala, G. Pierra, and L. Bellatreche. Ontodb : An ontology-based database for data intensive applications. In *to appear in the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07), edited by Springer's Lecture Notes in Computer Science*, pages 497–508, Bangkok - Thailand, April 2007.

[48] P. F. Dietz. Maintaining order in a linked list. *Fourteenth Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 122–127, 1982.

[49] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. *Sixteen Annual ACM Symposium on Theory of Computing (STOC'87)*, pages 365–372, 1987.

[50] K. Douglas and S. Douglas. *PostgreSQL*. New Riders Publishing, 2003.

[51] EHCACHE. ehcache project. *available at* `http://ehcache.sourceforge.net/`, 2003.

[52] A. Eisenberg and J. Melton. Sql : 1999, formerly known as sql3. *SIGMOD Rec.*, 28(1) :131–138, 1999.

[53] R. Elmasri and S. Navathe. *Fundamentals of Database Systems (3rd ed. ed.)*. Addison Wesley, 2000.

[54] P. Eric, J. 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*, pages 539–543. IEEE Computer Society, 2004.

[55] C. Fankam, Y. Ait-Ameur, and G. Pierra. Exploitation of ontology languages for both persistence and reasoning purposes : Mapping plib, owl and flight ontology models. In *To appear in Third International Conference on Web Information Systems and Technologies(WEBIST), Edited by : Joaquim Filipe, José Cordeiro, Bruno Encarnação and Vitor Pedrosa. INSTICC Press*, 2007.

[56] D. Fensel. *Ontologies : Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag, 2003.

[57] D. Fensel, F. van Harmelen, M. Klein, H. Akkermans, J. Broekstra, C. Fluit, J. van der Meer, H. Schnurr, R. Studer, J. Hughes, U. Krohn, J. Davies, R. Engels, B. Bremdal, F. Ygge, T. Lau, B. Novotny, U. Reimer, and I. Horrocks. to-knowledge : Ontology based tools for knowledge management.

[58] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical report, 1999.

[59] G. Frank. A general interface for interaction of special- purpose reasoners within a modular reasoning system. *Question Answering Systems. Papers from the AAAI Fall Symposium*, pages 57–62, 1999.

[60] L. Gallagher. Object sql : Language extensions for object data management. *Proc. Intl. Conf. on Knowledge and Information Mgmt*, November 1992.

[61] J. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, and S. W. Tu. The evolution of protégé : an environment for knowledge-based systems development. *Int. J. Hum.-Comput. Stud.*, 58(1) :89–123, 2003.

[62] G.Pierra, E. Sardet, and R. Withier. Modélisation des données : le langage express. Technical report, LISI-ENSMA, Futuroscope, 1995.

[63] O. M. Group. Meta object facility specification version 1.4. *OMG document formal*, 2003.

[64] T. R. Gruber. A translation approach to por-

table ontology specifications. *Knowledge Acquisition*, 5(2) :199–220, 1993.

[65] N. Guarino. Formal ontology and information systems. *National Research Council*, 98.

[66] N. Guarino and C. Welty. Evaluating ontological decision with ontoclean. *Comm. ACM*, 45(2) :61–65, 2002.

[67] Y. Guo, Z. Pan, and J. Heflin. Lubm : A benchmark for owl knowledge base systems. *Journal of Web Semantics*, 3(2) :158–182, 2005.

[68] S. Harris and N. Gibbins. 3store : Efficient bulk rdf storage, 2003.

[69] J. Heflin and Z. Pan. Univ-bench ontology. *Available at* `http://www.lehigh.edu/~zhp2/univ-bench.daml`.

[70] N. Heudecker. Introduction to hibernate. *Available at* `http://www.systemmobile.com/articles/IntroductionToHibernate.html`, Dec 2003.

[71] 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.

[72] I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From shiq and rdf to owl : the making of a web ontology language. *J. Web Sem.*, 1(1) :7–26, 2003.

[73] IBM. Informix product family. *Available at* `http://www-306.ibm.com/software/data/informix/`.

[74] ICS-FORTH. The ics-forth rdfsuite. *http ://139.91.183.30 :9090/RDF*, page web site, 2001.

[75] IEC. Iec 61360 - component data dictionary. *International Electrotechnical Commission. Available at* `http://dom2.iec.ch/iec61360?OpenFrameset`, 2001.

[76] I.Horrocks and U.Sattler. Ontology reasoning in the shoq(d) description logic. *In Proc. of the 17th Int. Joint Conf. on Artificial Intelligence*, pages 199–204, 2001.

[77] I.Horrocks and U.Sattler. Decidability of shiq with complex role inclusion axioms. *Artificial Intelligence*, 160 :79–104, 2004.

[78] I.Horrocks, U.Sattler, and S.Tobies. Practical reasoning for expressive description logics. *In Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, 1705 :161–180, 1999.

[79] IS010303.02. Product data representation and exchange - part 2 : Express reference manual. *ISO*, (055), 1994.

[80] ISO13584-25. Industrial automation systems and integration - parts library - part 25 : Logical resource : Logical model of supplier library with aggregate values and explicit content. Technical report, International Standards Organization, Genève, 2004.

[81] ISO13584-42. Industrial automation systems and integration parts library part 42 : Description methodology : Methodology for structuring parts families. Technical report, International Standards Organization, Genève, 1998.

[82] M. Jackson. thirty year (and more) of databases. *Information and Software Technology*, (44) :969–978, 1999.

[83] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley Eds, 1999.

[84] S. Jean, Y. Aït-Ameur, and G. Pierra. Querying ontology based database using ontoql (an ontology query language). In *Proceedings of On the Move to Meaningful Internet Systems 2006 : CoopIS, DOA, GADA, and ODBASE, OTM Confederated International Conferences (ODBASE'06)*, pages 704–721, 2006.

[85] S. Jean, H. Dehainsala, D. Nguyen Xuan, G. Pierra, L. Bellatreche, and Y. Aït-Ameur. Ontodb : It is time to embed your domain ontology in your database. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07) (Demo Paper), , edited by Springer's Lecture Notes in Computer Science*, pages 1119–1120, April 2007.

[86] S. Jean, G. Pierra, and Y. Ait-Ameur. Ontoql : an exploitation language for obdbs. In *VLDB PhD Workshop*, pages 41–45, 2005.

[87] S. Jean, G. Pierra, and Y. Aït-Ameur. Domain ontologies : a database-oriented analysis. In *Web Information Systems and Technologies (WEBIST'2006)*, pages 341–351, 2006.

[88] A. Kalyanpur, B. Parsia, and J. A. Hendler. A tool for working with web ontologies. *Int. J. Semantic Web Inf. Syst.*, 1(1) :36–49, 2005.

[89] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *SODA '02 : Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete*

algorithms, pages 954–963, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[90] G. Karvounarakis, V. Christophides, and D. Plexousakis. Querying semistructured (meta) data and schemas on the web : The case of RDF & RDFS. Technical Report 269, 2000.

[91] W. Kim. Research directions in object-oriented database systems. In *PODS '90 : Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1990. ACM Press.

[92] W. Kim, I. Choi, S. Gala, and M. Scheevel. On resolving schematic heterogeneity in multidatabase systems. *Distrib. Parallel Databases*, 1(3) :251–279, 1993.

[93] W. Kim and J. Seo. Classifying schematic and data heterogeneity in multidatabase systems. *Computer*, 24(12) :12–18, 1991.

[94] M. Klaene. Using ibatis sql maps for java data access. *Resources for Java server-side developers*, 1, 2004.

[95] M. C. A. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. Ontology versioning and change detection on the web. In *EKAW '02 : Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 197–212, London, UK, 2002. Springer-Verlag.

[96] E. Krasner, P. Glenn, and T. Stephen. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 3(1) :26–49, 1988.

[97] D. Kroenke. *Object Oriented Modelling and Design*. Prentice-Hall, 1991.

[98] R. Lassila, O. andSwick. Resource description framework (rdf) model and syntax specification. *World Wide Web Consortium. Available at* `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222`, February 1999.

[99] 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.

[100] LISI. Plibeditor. *Available at* `http://www.plib.ensma.fr/plib/upload/PLIB-Editor-Manual.pdf`, 2001.

[101] C. Liu and J. Layland. Third generation database system manifesto. *In Computer Standards and Interfaces*, pages 41–54, 1991.

[102] L.Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. Rstar : an rdf storage and query system for enterprise resource management. *thirteenth ACM international conference on Information and knowledge management*, pages 484 – 491, 2004.

[103] G. L. M. Kifer and J. Wu. Logical foundations of object-oriented and frame-based languages. *JACM*, 42(4) :741–843, 1995.

[104] A. Magkanaraki, S. Alexaki, V. Christophides, and D. Plexousakis. Benchmarking rdf schemas for the semantic web. In *ISWC'02 : Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 132–146, London, UK, 2002. Springer-Verlag.

[105] M. Magnan and C. Oussalah. *Ingenierie Objets : Chapitre II : Objets et composition*. InterEditions, mai 1997.

[106] D. Maier. Comments on the "third generation database system manifesto". *OGI Technical Report CS/E 91-012*, 1991.

[107] F. Marguerie. Choisir un outil de mapping objet-relationnel. *Available at* `http://dotnetguru.org/articles/articlets/choixmapping/mapping.htm`, Décembre 2004.

[108] V. M. Markowitz and A. Shoshani. On the correctness of representing extended entity-relationship structures in the relational model. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 430–439. ACM Press, 1989.

[109] N. Michael, N. Graham, H. Yousef, and A. Khalidi. A framework for caching in an object-oriented system, 1993.

[110] H. S. Michael R. Blaha, William J. Premerlani. Converting oo models into rdbms schema. *IEEE*, 11(3) :28–39, May 1994.

[111] A. Michel. Encyclopaedia universalis. (2000). dictionnaire de la philosophie. Technical report, 2000.

[112] M. Mimoune, G. Pierra, and Y. Aït-Ameur. An ontology based approach for exchanging data between heterogeneous database systems. *ICEIS'03*, 4 :512–524, 2003.

[113] M. Minsky. Matter, mind and models. In *Proc. of the International Federation of Information Processing Congress*, pages 45–49, 1965.

[114] M. Minsky. *The Society of Mind*. 1986.

[115] musicbrainz. mmsicbrainz metadatabase. *Available at* `http://musicbrainz.org/`.

[116] A. Napoli. Une brève introduction aux logiques de descriptions. Technical report, LORIA UMR 7503.

[117] Nescape. Open directory project (odp). *Available zt* `http://www.aef-dmoz.org/help/geninfo.html`.

[118] N. Noy and M. Musen. Ontology versioning in an ontology management framework. *Intelligent Systems, IEEE [see also IEEE Intelligent Systems and Their Applications]*, 19(4) :6–13, 2004.

[119] N. F. Noy and M. Klein. Ontology evolution : Not the same as schema evolution. *Knowl. Inf. Syst.*, 6(4) :428–440, 2004.

[120] OMG. Uml resource page. *Available at* `http://www.uml.org/`.

[121] Z. Pan and J. Heflin. Dldb : Extending relational databases to support semantic web queries. In *Workshop on Practical and Scaleable Semantic Web Systms, ISWC 2003*, pages 109–113, 2003.

[122] G. Pierra. Context-explication in conceptual ontologies : Plib ontologies and their use for industrial data. *Journal of Advanced Manufacturing Systems, World Scientific Publishing Company - à paraitre*.

[123] G. Pierra. New advances in computer aided design and computer graphics. In *International Academic Publishers*, pages 368–373, 1993.

[124] G. Pierra. Developing a new paradigm : Ontology-based information modelling. *ISO TC184/SC4/WG2 meeting*, Juin 2002.

[125] G. Pierra. Un modèle formel d'ontologie pour l'ingénierie, le commerce électronique et le web sémantique : Le modèle de dictionnaire sémantique plib. *Journées Scientifiques WEB SEMANTIQUE*, Octobre 2002.

[126] G. Pierra. Context-explication in conceptual ontologies : The PLIB approach. In *Proc. of Concurrent Engineering (CE'2003)*, pages 243–254, July 2003.

[127] G. Pierra, H. Dehainsala, Y. A. Ameur, L.Bellatreche, J. Chochon, and M. Mimoune. Base de Données à Base Ontologique : le modèle OntoDB. In *Proceeding of Base de Données Avancées 20èmes Journées (*BDA*'04)*, pages 263–286, Oct 2004.

[128] 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.

[129] G. Pierra, J. C. Potier, G. Battier, E. Sardet, J. C. Derouet, N. Willmann, and A. Mahir. Exchange of component data : The plib (iso 13584) model, standard and tools. pages 160–176, September 98.

[130] A. Plantec. *Utilisation de la norme STEP pour la spécification et la mise en œuvre de générateurs de code*. PhD thesis, Université de Bretagne Occidentale, fevrier 1999.

[131] D. Price. Standard Data Access Interface. *ISO-CD 10303-22*, 1995.

[132] J. C. Process. Java data objects (jdo). *Available at* `http://java.sun.com/products/jdo/index.jsp`.

[133] E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf. *Available at* `http://www.w3.org/TR/rdf-sparql-query/`.

[134] J. Raharu, E. Chang, T. Dillon, and D. Taniar. A methology for transforming inheritance relationships in an object-oriented conceptual model tables. *Information and Software Technology*, 42 :571–592, 2000.

[135] J. Rahayu, E. Chang, T. Dillon, and D. Taniar. Performance evaluation of the object-relational transformation methodology. *Data Knowl. Eng.*, 38(3) :265–300, 2001.

[136] W. Rahayu, E. Chang, T. Chang, and D. Taniar. Aggregation versus association in object modelling and databases. In *Australian Conf. on Information Systems*, 1996.

[137] W. Rahayu, E. Chang, and T. Dillon. A methodology for the design of relational databases from object-oriented conceptual models incorporating collection types. In *18th Internat. Conf. on Technology of Object-oriented Languages and Systems*. Prentice-Hall, 1995.

[138] W. Rahayu, E. Chang, and T. Dillon. Implementation of object-oriented association relationships in relational databases. In *IDEAS*, pages 254–263, 1998.

[139] W. Rahayu, E. Chang, and T. Dillon. Representation of multi-level composite objects in relational databases. In *Internat. Conf. on Object-oriented Information Systems OOIS'98*. Springer, 1998.

[140] W. Rahayu, E. Chang, and T. Dillon. Composite indices as a mechanism for transforming multi-level composite objects into relational databases (special issue on best of oois'98). *The OBJECT Journal*, 5(1), 1999.

[141] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Database Processing Fundamentals, Design, and Implementation*. Prentice-Hall International Editions, 1995.

[142] E. Sardet. *Intégration ds approches modélisation conceptuelle et structuration documentaire pour la saisie, la représentation, l'échange et l'exploitation d'informations. Application aux catalogues de composants industriels.* PhD thesis, Université de Poitiers, 1999.

[143] D. Schenk and P. Wilson. *Information Modelling The EXPRESS Way*. Oxford University Press, 1994.

[144] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language- Chapitre 10*. Kluwer, 1995.

[145] C. Soutou. Modeling relationships in object-relational databases. *Data and Knowledge Engineering*, 36(1) :79–107, 2001.

[146] P. Spyns, R. Meersman, and M. Jarrar. Data modelling versus ontology engineering. *SIGMOD Rec.*, 31(4) :12–17, 2002.

[147] 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.

[148] STEP. Sdai schemas. *available at* `http://www.steptools.com/support/stdev_docs/sdailib/sdailib-7.html#35537`, 2006.

[149] K. Stoffel, M. Taylor, and J. Hendler. Efficient management of very large ontologies. In *AAAI/IAAI*, pages 442–447, 1997.

[150] K. Stoffel, M. G. Taylor, and J. A. Hendler. Efficient management of very large ontologies. In *AAAI/IAAI*, pages 442–447, 1997.

[151] L. Stoimenov. Bridging objects and relations : a mediator for oo front-end to rdbms. *Information and Software Technology*, 41 :57–66, Oct. 1999.

[152] M. Stonebraker and D. Moore. *Object-relational DBMSs the Next Great Wave*. Morgan Kaufmann, 1996.

[153] J. Sutherland, M. Pope, and K. Rugg. The hybrid object-relational architecture (hora) : an integration of object-oriented and relational technology. In *SAC '93 : Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*, pages 326–333, New York, NY, USA, 1993. ACM Press.

[154] H. Tardieu, A. Rochfeld, and R. Coletti. *La methode MERISE : principles et outils*. Edition Organisation, 1983.

[155] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. 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 Press.

[156] C. Tempich and R. Volz. Towards a benchmark for semantic web reasoners - an analysis of the daml ontology library.

[157] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of rdf/s stores. In *International Semantic Web Conference*, pages 685–701, 2005.

[158] S. Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, RWTH Aachen, 2001.

[159] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.

[160] R. M. V. Haarslev. Description of the racer system and its applications. *Intl Workshop on Description Logics (DL-2001)*, August 2001.

[161] J. B. V.Psyché, O.Mendes. Apport de l'ingénierie ontologique aux environnements de formation à distance. *Revue STE*, pages 89–126, 2003.

[162] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-based integration of information - a survey of existing approaches. *Proceedings of the International Workshop on Ontologies and Information Sharing*, pages 108–117, August 2001.

[163] H.-C. Wei and R. Elmasri. Study and comparison of schema versioning and database

conversion techniques for bi-temporal databases. In *TIME*, pages 88–98, 1999.

[164] H. Wiedmer and G. Pierra. *Methodology For Structuring Part Families*. ISO-IS 13584-42. ISO Genève, 1998.

[165] 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.

[166] D. N. Xuan, L. Bellatreche, and G. Pierra. Ontology evolution and source autonomy in ontology-based data warehouses. *to appear to : Entrepôts de Données et l'Analyse en ligne (EDA'06)*, pages 55–76, June 2006.

[167] D. N. Xuan, L. Bellatreche, and G. Pierra. Un modele a base ontologique pour la gestion de l'evolution asynchrone des entrepots de donnees. *MOSIM'06 :Modélisation, Optimisation et Simulation des Systèmes : Défis et Opportunités*, pages 1682–1691, April 2006.

Annexe A

Le modèle PLIB

1 Introduction

Cette annexe est un complément de la description du modèle d'ontologie PLIB fait dans la section 2.2.3 du chapitre 2. Comme déjà introduit, le modèle d'ontologie PLIB a été initialement défini pour permettre la représentation, l'échange des bibliothèques ou catalogues de composants industriels. Le modèle d'ontologie PLIB a été défini dans le langage de modélisation EXPRESS [79] (cf. section 2 du chapitre 1)qui est un langage formel (i.e. compilable) et permet de définir des modèles fiables, précis et non ambigus.

La description faite de cet annexe sera portée sur les entités EXPRESS qui compose le modèle d'ontologie PLIB, ce qui permettra (1) de montrer comment une ontologie PLIB est structurée, et (2) de montrer comment toutes les caractéristiques des ontologies PLIB (listées dans la section 2.2.3 du chapitre 2) sont effectivement mises en œuvre.

Le modèle PLIB peut être divisé en trois parties :

1. Identification des concepts qui se compose des entités EXPRESS qui permettent d'associer un identifiant Globalement unique (GUI) aux concepts (classes, propriétés, types, etc.) des ontologies et qui permettront leur référencement.
2. Définition des concepts qui se compose des entités EXPRESS qui permettent de décrire les concepts au moyen de propriétés caractéristiques, ce qui permet de les rendre compréhensible pour des opérateurs humains. La définition d'un concept peut être définie en plusieurs langues.
3. Extension des concepts qui se compose des entités EXPRESS qui permettent de décrire les instances des concepts de l'ontologie.

Dans les sections qui suivent, nous présentons brièvement les entités qui composent chacune de ces parties.

2 Identification des concepts (BSU)

L'identifiant GUI des concepts permet (1) d'éviter tous problèmes liés à la confusion de concepts lors d'une intégration et échange, (2) d'accéder à la description et l'extension des concepts, (3) d'échanger (si on le souhaite) séparément la description des concepts et leurs extensions, et (4) permet de garder un historique des différents versions d'un composant.

Dans le jargon PLIB, un identifiant GUI est appelé *BSU* (Basic Semantic Unit). Le terme BSU provient de la norme ISO/IEC11179-3 où il est défini comme "*the smallest unit of information that may be agreed up on at the conceptual level*". L'identifiant d'un fournisseur est appelé *Supplier_BSU*. Celui d'une classe est *class_BSU*, celui d'une propriété est *property_BSU*, celui des types de données *datatype_BSU*, et enfin celui des documents *document_BSU*. La figure A.1 montre un schéma EXPRESS-G des principaux concepts de PLIB avec les relations entre eux.

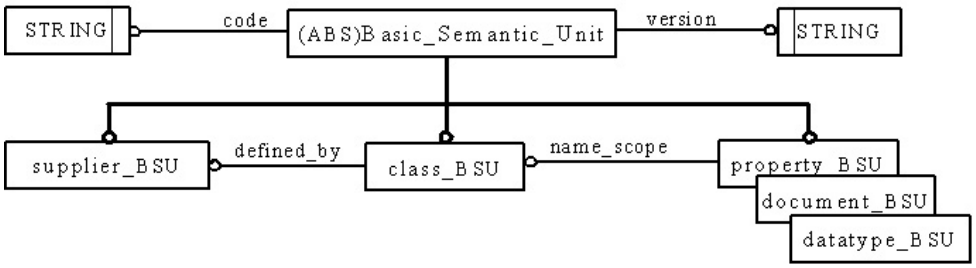


Fig. A.1 – Schéma EXPRESS-G d'identifications des concepts en PLIB

Un identifiant (BSU) PLIB, est constitué d'un code en chaîne de caractères dont la syntaxe est définie par [164], d'un numéro de version qui indique le numéro de version du concept. PLIB autorise qu'un concept soit identifié plusieurs fois en faisant évoluer le numéro de version du concept. Ce versionnement est contrôlé par des règles et principes définis informellement dans la partie 26 de la norme PLIB. Nous discutons de façon plus détaillée du versionnement des concepts dans la section 5.

Dans la figure A.1, on peut constater qu'une classe (class_BSU) (respectivement une propriété, un type de données et un document) référence le BSU de son fournisseur (supplier_BSU) (respectivement sa classe) qui la définie. Ce mécanisme permet de rendre unique universellement un concept. En effet, le code et la version du BSU d'une classe (respectivement d'une propriété, d'un type, ou d'un document) permet de la rendre unique seulement dans le contexte du fournisseur (respectivement de la classe). Mais la concaténation du code et de la version du fournisseur (respectivement classe) avec celle d'une classe (respectivement une propriété, un type ou un document) permet de la rendre unique universellement. Cette fusion est appelée *absolute_id* dans le jargon PLIB. L'exemple suivant, présente comment est défini l'*absolute_id* des différents concepts d'une ontologie PLIB.

Exemple :

	Concepts	code	version	absolute_id
Fournisseur	LISI	EA1232	-	EA1232
Classe	personne	PERSONNE	001	EA1232//PERSONNE-001
Propriété	nom	NOM	001	EA1232//ETUDIANT-001//NOM-001
Classe	étudiant	ETUDIANT	002	EA1232//ETUDIANT-002
Propriété	niveau	NIVEAU	003	EA1232//ETUDIANT-002//NIVEAU-003

3 Définition des concepts

Un concept, une fois identifié, est ensuite décrit formellement. La définition consiste à décrire précisément un concept en vue de le rendre compréhensible, opérationnelle et non ambiguë. Les entités du modèle PLIB pour la définition des concepts se trouve dans la partie 42 de la norme PLIB [164]. La définition d'une classe consiste, entre autres, à définir les propriétés qui la caractérisent et la mettre en relation avec d'autres classes (subsomption, association, etc.). La définition d'une propriété consistera entre autres à définir son co-domaine, son unité de mesure et de spécifier sa *typologie* [129] (propriété caractéristique, de contexte ou dépendante de contexte). Chaque définition de concept référence le BSU de son concept. Le modèle PLIB impose une seule définition par concept d'une ontologie.

Nous présentons dans les sections qui suivent, avec plus de détails la définition des concepts en PLIB. Cette définition se décomposera suivant les aspects structurel et descriptif de la modélisation de la connaissance.

3.1 Aspect structurel

La figure A.2 montre la structure simplifiée de la définition d'une classe en EXPRESS-G.

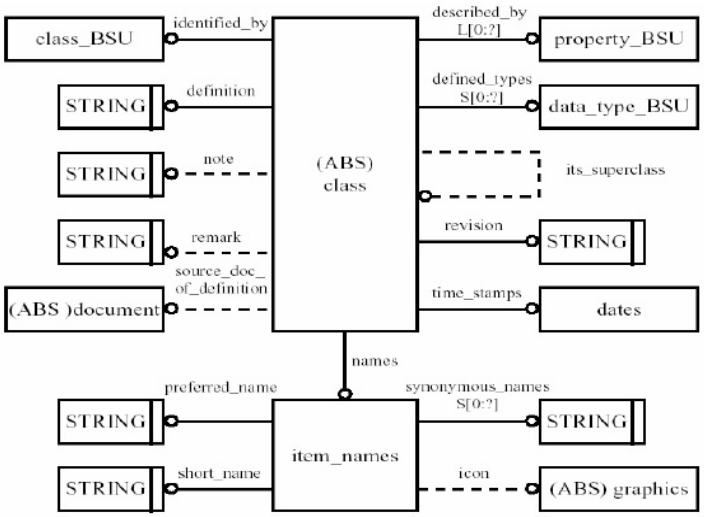


Fig. A.2 – Définition d'une classe en PLIB

Une classe est identifiée par un *class_BSU* donné par l'attribut *identified_by*. L'attribut *described_by* indique les propriétés qui caractérisent la classe. Les propriétés de cette liste sont

dites les propriétés *applicables* de la classe. Ces propriétés applicables sont à distinguer des propriétés dites *visibles* qui sont l'ensemble des propriétés définies dans le contexte d'une classe. Concrètement, ces propriétés sont celles qui référencent directement la classe à laquelle elles appartiennent (cf. figure A.1). A ces deux types de propriétés (applicables et visibles), il faut ajouter les propriétés *fournies* ou *utilisées*, qui sont les propriétés effectivement initialisées par les instances des classes. Nous verrons plus loin que le modèle PLIB impose que les propriétés fournies dans l'extension d'une classe doivent être un sous-ensemble de l'ensemble des propriétés applicables de la classe. Remarquons qu'une propriété visible ne peut être rendue applicable dans une sous-classe de la classe où elle est visible. Cette particularité de PLIB permet à juste titre de factoriser [142] au niveau des hiérarchies des propriétés communes à un sous-ensemble de classes repartie de façon disparate dans la hiérarchie.

Une classe PLIB peut avoir au plus et une seule super-classe renseignée par l'attribut *its_superclass*. Les attributs (*definition*, *note*, *remark*, *source_doc_of_definition*) permettent d'associer à la classe des descriptions textuelles et/ou graphiques et/ou dessins techniques qui indiquent entre autres des informations sur les conditions d'utilisation de la classe. L'attribut *names* de type *item_names* sert à nommer la classe en lui attribuant un nom préféré (*preferred_name*), un nom court (*short_name*) et un ensemble de synonymes. Enfin, il faut noter dans la définition d'un concept PLIB, toute information textuelle peut être fournie en plusieurs langues différentes.

Le modèle PLIB définit quatre types de relations sémantiques entre classes :

1. Les relations sémantique de subsomption *is-a* et *is-case-of* : Ce sont les relations de *généralisation/spécialisation* de l'approche objet. Le relation est *is-a* est la subsomption avec héritage tandis que la relation *is-case-of* vise à importer implicitement un sous ensemble des propriétés des classes subsumées.

2. La relation sémantique d'agrégation *is-part-of* : C'est l'habituelle relation d'agrégation de l'approche objets qui associe un tout et ses constituants, ou encore un groupe et ses membres. Selon PLIB, chaque objet, par exemple, peut être soit un objet atomique (une vis par exemple), soit un assemblage d'objets (un assemblage vis-écrou-rondelle).

3. La relation sémantique *is-view-of* : C'est la relation entre une classe particulière (une classe vis) et une classe de modèles (techniques) particuliers (par exemple, les modèles de géométrie paramétrique aptes à représenter les géométries particulières de toutes les instances de sa classe vis).

Concrètement en PLIB, une classe en relation de subsomption *is-a* (respectivement *is-case-of*) avec une deuxième classe sera construite avec l'entité *item_class* (respectivement *item_class_case_of*). Les classes composites d'une classe agrégat sont définies sous forme de *feature_class*, la relation d'agrégation *is-part-of* est définie comme pour association ou une collection (suivant la cardinalité) définies dans PLIB (cf. section sur les types de valeurs). Enfin, les classes des vues fonctionnelles sont définies avec les entités *functional_view_class*. Notons que toutes ces entités EXPRESS du modèle PLIB sont des sous-types de l'entité *class* présentés dans la figure A.2.

3.2 Aspect descriptif

Comme une classe, les propriétés identifiées (par des *property_ BSU*), sont également décrites. PLIB définit trois types de propriétés pour caractériser les instances des classes :

- les propriétés caractéristiques qui représentent les propriétés intrinsèques des instances des classes,
- les propriétés paramètres de contexte qui permettent de caractériser le contexte où les instances (ou objets) vont être insérées, ou dans lequel une propriété dépendante du contexte est évaluée, et
- les propriétés dépendantes du contexte qui représentent les propriétés ayant une dépendance fonctionnelle avec au moins un paramètre de contexte et caractérisant le comportement du composant en fonction de son contexte d'utilisation.

Une propriété caractéristique est définie dans le modèle PLIB comme une instance de l'entité *non_ dependent_ P_ DET*. Une propriété paramètre de contexte est définie au moyen de l'entité *condition_ DET*. Une propriété dépendante de contexte est définie au moyen de l'entité *dependent_ P_ DET*.

La figure A.3 présente un schéma EXPRESS-G des entités PLIB intervenantes dans la description d'une propriété.

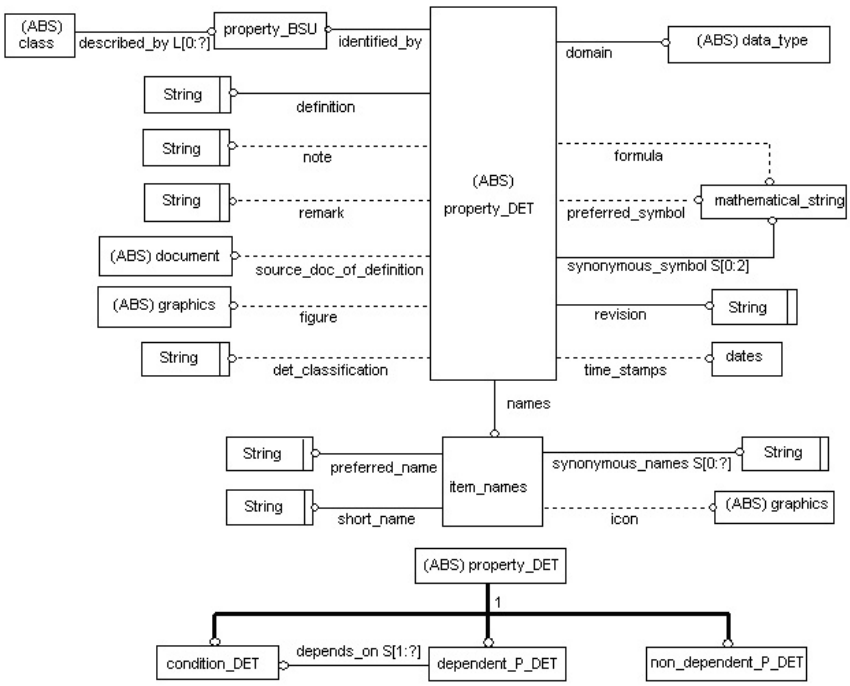


Fig. A.3 – Schéma EXPRESS-G de la description d'une propriété en PLIB

On peut remarquer sur la figure que tous les types de propriétés héritent de l'entité *property_ det* dans laquelle est définie l'attribut *identified_ by* qui est une référence vers l'identifiant de la propriété. Les attributs *preferred_ symbol*, *synonymous_ symbol*, *formula*, *figure* permettent

d'associer à la propriété des informations graphiques et textuelles qui étayent la description de la propriété pour son utilisation. L'attribut *domain* permet de spécifier le domaine de valeur de la propriété. Enfin, une propriété peut être associée à des descriptions textuelles et éventuellement graphiques (les attributs *figure*, *définition*, *note*, *remark*, etc.) qui permettent de clarifier son sens pour qu'elle soit opérationnelle.

Dans le paragraphe suivant, nous présentons tous les types valeurs décrits dans PLIB.

3.2.1 Domaine des propriétés

Le modèle PLIB propose plusieurs types de données pour les valeurs des propriétés [164]. Ces types peuvent être regroupés en quatre catégories (cf. figure A.4) :

- les types simples,
- les types complexes,
- les types nommés,
- les types agrégats.

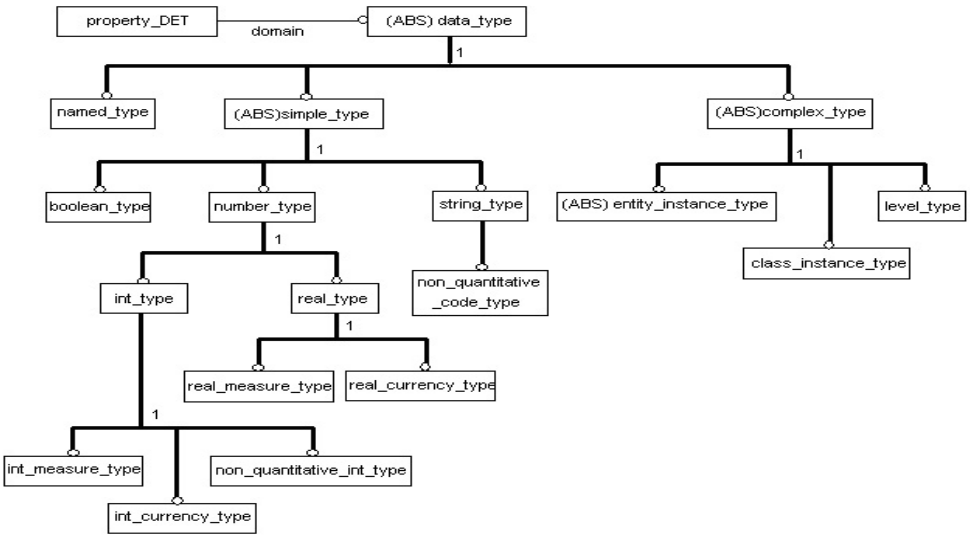


Fig. A.4 – Les types de PLIB

Les types simples

Les types simples sont constitués des types de bases classiques utilisés dans langages de programmations (Entier, Réel, String, Booléen). Les types *[int,real]_ measure_ type* (cf. figure A.4) sont des types entiers (respectivement réels) associés à une unité de mesure. Par exemple, la longueur d'une vis peut être exprimée en mètre (m) ou centimètre (cm). Les types *[int, real]_ currency_ type* sont des types associés à une unité monétaire (Euro, CFA, Dollar, Yen, etc.). Le modèle PLIB permet de définir les valeurs des propriétés dans presque toutess les unités de mesure : température (Celsus $C°$, Faraday $F°$, etc.), distance (KiloMètre km, Mètre M, etc.), masse (Tonne T, KiloGramme KG, Gramme G, etc.), poids (Newton N, etc.), etc.). Les types *non_ quantitative_ [int, code]_ type* sont des types énumérés (ou de hachages) avec respectivement

un entier (respectivement une chaîne de caractères) pour la clé.

Les types complexes
Les types complexes sont au nombre de trois :

1. le type quantitatif : *level_ type*,
2. le type pour définir une association : *class_ instance_ type*,
3. le type *entity_ instance_ type*.

Le type *level_ type* fournit un indicateur pour qualifier les valeurs d'un type quantitatif. Son attribut *levels* est une liste d'éléments de type *level* (qui est un type énuméré).

TYPE level = **ENUMERATION OF** (min, nom, typ, max) ;
END_TYPE ;

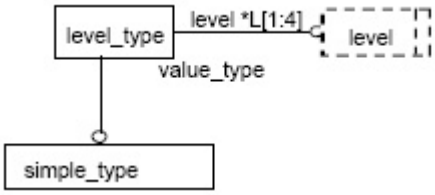


Fig. A.5 – Le type *level_ type* en PLIB

La valeur de l'élément *min* (minimal) qui correspond à la valeur minimal d'une quantité physique ; *nom* (nominal) qui correspond à la valeur nominale d'une quantité physique ; *typ* (typical) qui correspond à la valeur caractéristique d'une quantité physique ; enfin *max* (maximal) qui correspond à la valeur maximale d'une quantité physique. Cet attribut (*levels*) spécifie quels types de valeurs qualificatives devraient être indiquées pour une propriété de type *level_ type*. L'attribut *value_ type* définit le type de valeur des différents éléments de type *level_ type*.

Le type *class_ instance_ type* de PLIB permet de définir une association entre deux classes. Il est utilisé particulièrement pour des agrégations et des compositions entre des classes. L'attribut *domain* indique l'identifiant de la classe (*class_ bsu)* mis en relation (cf. figure A.6).

Le type *entity_ instance_ type* est un type *générique* qui permet d'étendre le système de type de PLIB. Ce type est identique au type *class_ instance_ type* à la différence qu'il permet non

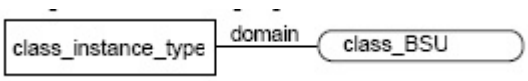


Fig. A.6 – Structure d'une association en PLIB

pas de référencer des instances de classes PLIB mais plutôt des instances d'entités EXPRESS. L'attribut *type_name* permet de préciser le nom des entités auxquelles les instances devraient appartenir. Ce type a été justement étendu pour la définition de type de ressources externes (*external type*) qui permet à une propriété de référencer des ressources externes. Ces ressources sont entre autres des documents multimédia textuelles, images, sons, etc. (cf. section 3.2.2 pour plus de détails sur la description des ressources externes en PLIB).

Les types nommés
Les types nommés sont des types définis à partir de types existants et ont la particularité d'être associés à un identifiant universel (*data_type_bsu*). Ces types sont décrits dans le but d'être éventuellement réutilisés et/ou échangés (comme les classes et propriétés) dans d'autres ontologies.

Les types agrégats
Les types agrégats permettent de définir des propriétés dont les valeurs peuvent être une collection de données. Les agrégats en PLIB peuvent être soit des ensembles (*SET_TYPE*), des listes (*LIST_TYPE*), des tableaux (*ARRAY_TYPE*), ou des sacs (BAG_TYPE). Le type de base des agrégats peut être de n'importe quels autres types que propose PLIB.

3.2.2 Description des ressources externes de la norme PLIB : les fichiers externes

Le modèle PLIB offre la possibilité de référencer des ressources qui peuvent être des documents multimédias (sons, vidéos, images, etc.). Ces ressources peuvent être utilisées à deux niveaux dans le modèle d'ontologie PLIB (niveau définition des concepts et niveau extension des concepts) :

1. *Définition des concepts de l'ontologie.*
   La norme PLIB permet d'associer à la description des propriétés et des classes, des documents (techniques) qui peuvent être graphiques (images, figures, schémas) et/ou textuels. Ces documents permettent entre autres d'étayer la description d'un concept.
   Par exemple, sur la figure A.3, les attributs : *figure* et *icon*, permettent justement de référencer des ressources externes.

2. *Extension des concepts.*
   La norme PLIB définit un type de données particulier nommé *external type*. Une valeur de ce type permet de référencer des documents (fichiers) ou des ressources web. Par exemple, pour une classe *personne*, on pourrait envisager de définir une propriété *photo* dont le *co-domaine* serait de type *External type* et qui permettra d'associer à chaque instance de *personne*, sa photographie numérisée dans un fichier sur disque.

Les ressources externes peuvent être soit des URLs (Uniforme Resource Location) accessibles à travers le Net ou un chemin de fichiers stockés localement dans une machine. Dans ce deuxième cas, lors de l'échange des ontologies ou des instances entre différents systèmes, l'ensemble des documents (fichiers) associés aux classes ou aux propriétés ou valeurs de propriétés des instances doivent l'être aussi. Le modèle PLIB définit un protocole pour permettre cet échange.

4 Extension des concepts

La troisième partie du modèle PLIB, dite extension, permet de définir les données à base ontologique (i.e. les instances des concepts). Le modèle de représentation explicite est basé sur la description explicite de chaque composant, en énumérant toutes les valeurs de ses propriétés. Cette méthode présente l'avantage de simplifier le modèle de données PLIB et de faciliter l'intégration de données de composants décrites dans le modèle PLIB. De plus, cela facilite la gestion de données de catalogues de composants par des bases de données relationnelles objets [112]. Les avantages du modèle d'extension explicite sont qu'il permet de décrire simplement les composants et offre ainsi une implémentation facile. La partie 24 de la norme modélise ces instances sous forme de méta-modèle. Elle suit donc dans la même optique que le méta-modèle de la partie 42 de la norme.

La figure A.7 donne une vue réduite en EXPRESS-G du modèle d'extension de PLIB. L'entité *class_ extension* permet de définir l'ensemble des instances d'une classe. Son attribut *population*, qui est une liste, permet d'énumérer toutes les instances de la classe. L'entité *dic_ class_ instance* définit une instance. Elle est constituée d'une référence vers sa classe d'origine (*its_ class*) et d'une liste de propriété-valeur (*property_ value*) qui associe une valeur (*valeur*) à chaque propriété de l'instance (*its_ property*).

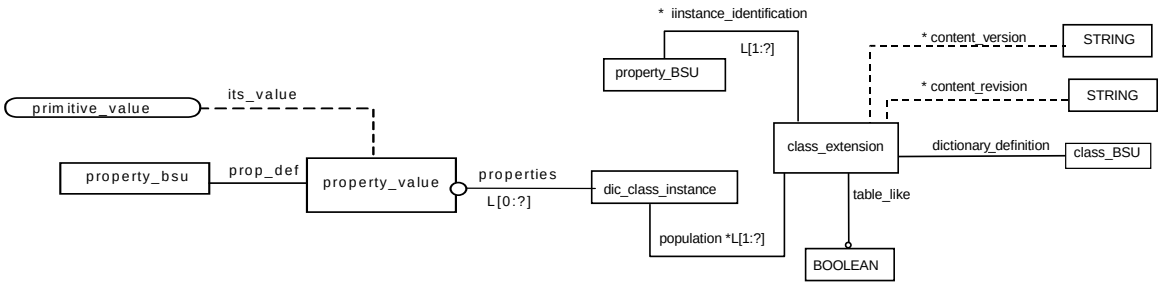


Fig. A.7 – Schéma simplifié de l'extension de classes et propriétés en PLIB

5 Versionnement en PLIB

Une ontologie, comme tous les modèles et les systèmes, est susceptible d'évoluer. Ces évolutions peuvent porter soit sur les concepts (classes, propriétés, etc.) ou les populations des instances. Ces évolutions peuvent consister à l'ajout des nouveaux concepts ou instances de classes ou soit à la modification ou suppression de concepts ou instances des ontologies.

Face à ces évolutions et pour permettre une bonne suivie des différentes versions des ontologies i.e., leur cycle de vie, il est indispensable de mettre en œuvre des mécanismes qui permettront d'assurer une bonne gestion des différentes versions des ontologies et de permettre à leur réintégration (mise à jour) dans les systèmes qui utilisent ces ontologies.

Le modèle d'ontologie PLIB définit un mécanisme simple et un protocole (informel) qui permet une évolution des différents concepts et populations des ontologies. Dans les sections qui suivent, nous décrivons ces mécanismes et protocoles définis dans PLIB.

5.1 Versionnement des concepts

Chaque concept de PLIB est associé à deux attributs *version* et *révision* qui permettent la suivie d'évolution des concepts. L'évolution du numéro de *version* et de *révision* d'un concept n'obéit à aucune règle formelle. Mais la norme PLIB [81], définit un protocole *informel* pour l'évolution du numéro de *version* et *révision* des concepts. Ce protocole se base sur les modifications des attributs caractéristiques des concepts (cf. figures A.2 et A.3). Les tables A.1 et A.2 donnent la façon à laquelle l'initialisation, la modification ou la suppression d'un attribut d'une classe affectent la version et/ou révision des concepts en PLIB. Sur ces tables :

- **V** signifie que l'opération sur l'attribut affecte la *Version* du concept,
- **R** signifie que l'opération affecte *Révision* du concept,
- **X** signifie que l'opération n'est pas autorisé sur l'attribut du concept, et
- **-** signifie que l'opération sur cet attribut n'affecte ni la version ni la révision.

Attribut	Ajout	Modification	Suppression
Code	-	X	X
Definition Class	X	X	X
Data type	-	V	X
Preferred Name	-	R	X
Short Name	-	R	X
Synonumous Name	R	R	R
Definition	-	R/V	X
Note	R	R	R
Remark	R	R	R
Unit	-	X	X
Condition	R	X/R	X
Formula	-	R	X
Format	-	R	R

Tab. A.1 – Versioning propriétés en PLIB

5.2 Versionnement des extensions concepts

Les populations des classes peuvent également évoluer. D'une version à une autre, les opérations susceptibles d'être appliquées sur les populations d'instances sont entre autres :

- ajout d'une nouvelle instance d'une classe,
- suppression d'une instance,
- modification de la valeur d'une propriété,
- suppression d'une propriété d'une instance,
- etc.

Attribut	Ajout	Modification	Suppression
Code	-	X	X
Superclass	V	V	V
Preferred Name	-	R	X
Short Name	-	R	X
Synonumous Name	R	R	R
Definition	-	R	X
Note	R	R	R
Remark	R	R	R
Simplified Drawing	-	R	R
applicable properties	V	V	X
visible properties	V	V	X

Tab. A.2 – Versioning des classes en PLIB

Pour assurer une bonne suivie des populations d'instances d'une version à une autre, PLIB définit au niveau de l'extension des classes deux propriétés *content_version* et *content_revision* qui sont respectivement le numéro de version et de révision associés à une population d'instances (cf. section 4 et la figure A.7).

PLIB définit le protocole suivant à respecter pour l'incrémentation de la version et révision d'une instance :

- on incrémente la version d'une extension s'il y a de nouvelles instances ou des instances supprimées dans l'extension,
- on incrémente la version si la version de sa classe (définition) correspondante a évolué (cf. section 5.1),
- on incrémente la révision d'une extension si la description de la classe d'extension (cf. figure A.7) a subit une quelconque modification,
- lorsque la version d'une extension est incrémentée, la révision est réinitialisée à "0".

6 Conclusion

Nous venons de montrer dans cette annexe, les principales caractéristiques du modèle d'ontologie PLIB modéliser dans la langage EXPRESS. Le modèle EXPRESS de PLIB est très complexe et est constitué au total de 218 entités et de 80 types définis. Celui-ci a servi de base pour le modèle architecture de base de données à base ontologique *OntoDB* et le prototype du modèle d'architecture que nous avons implémenté sur le SGBDRO PostgreSQL.

Annexe B

Le SGBD relationnel objet : PostgreSQL

1 Introduction

PostgreSQL, développé à l'origine par le Computer Science Department de l'University of California Berkeley. Le projet fut dirigé par le professeur Michael Stonebraker, et a été sponsorisé par la DARPA-*Defense Research Projects Agency*, la ARO-*Army Research Office*, la NSF-*National Science Foundation* et ESL, Inc. Aujourd'hui PostgreSQL est un logiciel libre diffusé sous une licence réalisée par l'université de Berkeley. Le développement de PostgreSQL est maintenant réalisé par une équipe de développeurs[24].

PostgreSQL est aujourd'hui largement utilisé[25] comme serveur de données non seulement pour sa gratuité mais aussi pour ses performances. Certains SGBDs commercialisés ont d'ailleurs récemment incorporé des fonctionnalités développées pour PostgreSQL (le serveur Informix[73] Universal).

La version courante, au moment de la rédaction de cette thèse, est la *8.1*. Cette version vient d'être justement portée sur le système d'exploitation Windows. Les versions précédentes ne fonctionnaient que sur Linux. Comme la plupart des SGBDRO, PostgreSQL implémente en totalité la norme SQL92 [40] et très partiellement la norme SQL99 [52]. Dans la section suivante, nous présentons brièvement les principaux éléments de la norme SQL99 implémentés sous PostgreSQL, à savoir (1) classes, (2) héritage, (3) types complexes, (4) fonctions et procédures, (5) transactions, (6) déclencheurs.

2 Classes

Les tables des bases de données sont gérées comme étant des classes. Les tuples des tables sont vus comme des objets. Un identifiant unique OID (Object Identifier) identifie chaque objet

[24] http ://archives.postgresql.org/pgsql-hackers/
[25] http ://www.postgresql.org/about/casestudies/

(ou tuple) dans la base de données. Dans PostgreSQL, cet OID n'est pas référençable par d'autres objets (tuples) dans une association. La référence doit se faire par le mécanisme de clé étrangère.

3 Héritage

PostgreSQL supporte l'héritage (simple ou multiple) entre tables. L'opérateur "*" est défini pour permettre de faire de requêtes hiérarchique.

4 Types complexes

Outre les types de bases (int, float, varchar,boolean,...), PostgreSQL propose

- l'utilisation des types collections (Tableau) pour gérer les agrégats de données. Les tableaux peuvent être en multi-dimensionnels.
- l'utilisation des types complexes prédéfinis. Exemple : types géométriques (point, circles,...), types de données réseaux (inet, macaddr, cidr, ...), ...
- la définition de nouveau type utilisateur complexe correspondant aux RECORD dans les langages de programmation.

  Exemple :

```
CREATE TYPE adresse(
codepostal INTEGER,
pays VARCHAR,
boitepostale INTEGER,
...
);
```

5 Fonctions et procédures

PostgreSQL permet la création de fonctions et procédures qui seront stockées dans la base de données. Ces fonctions peuvent être écrites en plusieurs langages (plpgsql, perl, tcl, python). Ces fonctions et procédures permettent d'écrire des programmes complexes (i.e. au delà des requêtes que peut faire le langage SQL99). Ces programmes pourront être appelés par les clients de la base de données.

6 Transactions

PostgreSQL implémente la gestion des transactions. Les transactions permettent d'annuler ou de valider un ensemble d'opérations s'effectuant sur une base de données. PostgreSQL définit un degré de granularité supplémentaire au moyen de la commande SAVEPOINTS pour une gestion encore plus fine de sous-ensembles d'opérations d'une transaction. Une transaction peut donc être annulée ou validée à partir d'un certain "Point" à l'intérieur de la transaction.

7 Déclencheurs

Les déclencheurs sont représentés dans PostgreSQL au moyen des triggers. Les triggers sont associés aux tables et sont exécutés avant ou après des évènements qui peuvent être soit une insertion (INSERT), une mise à jour (UPDATE) ou une suppression(DELETE).

Annexe C

Règles de correspondances entre les mécanismes du langage EXPRESS et le relationnel

Dans cette annexe, nous présentons les règles de correspondances entre les mécanismes du langage EXPRESS et ceux du SGBDRO PostgreSQL que nous avons définies pour la représentation des instances des entités EXPRESS dans une base de données PostgreSQL.

L'annexe s'organise en deux sections. Dans la première section, nous discutons de la problématique de la définition des règles des correspondances des concepts du langage EXPRESS en relationnel ou relationnel objet. Dans la deuxième section, nous décrivons la mise en correspondance de chacun des concepts du langage EXPRESS dans le relationnel objet du SGBD PostgreSQL.

1 Problématique

Le langage EXPRESS comme nous l'avons présenté dans la section 2 du chapitre 1, est un langage mettant en œuvre tous les mécanismes objets. Sa représentation dans un univers relationnel objet présente un certain nombre de difficultés. Nous abordons ces problèmes séparément suivant les trois points de vues (structurel, descriptif, procédural) liés à la modélisation de la connaissance (cf. section 1 du chapitre 1).

1.1 Connaissance structurelle

La connaissance structurelle définit la façon dont les concepts de l'univers du discours (le domaine auquel on s'intéresse) sont structurés et des relations entre ces concepts.

1.1.1 Structure des classes

Le langage EXPRESS, étant un formalisme orienté objet, permet de modéliser la connaissance structurelle sous forme de hiérarchie de classes (entités) associée à un mécanisme de factorisa-

tion/héritage. Dans PostgreSQL, cette connaissance est représentée par des tables entre lesquelles l'héritage est admis.

1.1.2 Association et polymorphisme

Les relations entre les objets, ou associations, sont exprimées en EXPRESS au moyen d'attribut dont le domaine est une autre entité. Cette association est polymorphe, c'est à dire, la valeur de l'attribut peut, dans un contexte donné être du type d'un des descendants de l'entité associée. Bien que l'on puisse exprimer une relation entre les tables entre PostgreSQL au moyen de clé étrangère (FOREIGN KEY), malheureusement cette solution a deux insuffisances :

(1) la contrainte FOREIGN KEY n'est pas héritée dans les sous-tables des entités en d'autres termes la contrainte ne s'appliquera pas sur les tables de ses sous classes,

(2) ensuite elle n'est pas polymorphe, c'est à dire que seulement les instances de la table référencée interviendront dans le FOREIGN KEY. Les instances des sous-tables ne pourront pas être référencées.

L'exemple de la figure C.1 illustre cette problématique. Les flèches en pointillées désignent les relations impossibles en utilisant une contrainte FOREIGN KEY classique entre les tables *Table 1* et *Table 2*.

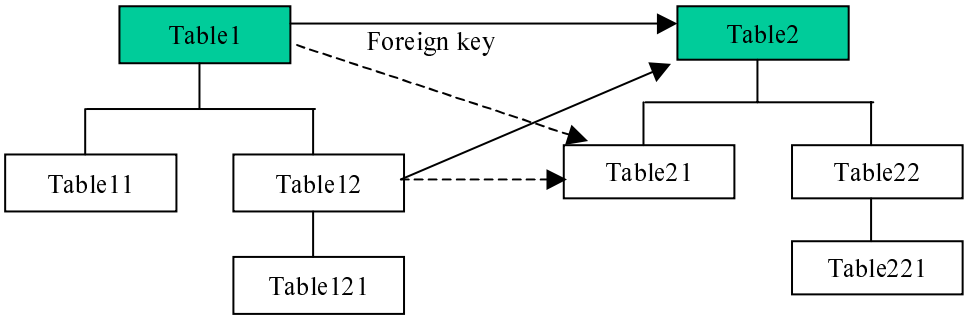


Fig. C.1 – Limites de la contrainte FOREIGN KEY lors du polymorphisme

1.1.3 Identifiant d'objet et référence

Dans le langage EXPRESS, toute instance d'une entité est associée un OID (Object IDentifier) unique pour toutes les instances du système. Cet identifiant semblable existe également en PostgreSQL. En effet, celui-ci définit une colonne *OID* automatiquement à toute nouvelle table dans une base de données et à chaque insertion d'un tuple dans une des tables, le système initialise cette colonne OID a une valeur unique dans toute la base de donnée. L'inconvénient de cette colonne OID de PostgreSQL, est qu'elle est à un usage interne pour le système. Elle ne peut être par exemple référencée par une autre colonne clé étrangère.

Pour cette connaissance structurelle, nous sommes contraints de trouver des mécanismes élaborés afin de représenter l'association, le polymorphisme qui sont deux mécanismes fortement utilisés dans de le langage EXPRESS et de trouver également une solution au problème de l'iden-

tifiant des instances. Nous abordons dans la section suivante, les problèmes liés à la représentation de la connaissance descriptive.

1.2 Connaissance descriptive

La connaissance descriptive associe aux concepts de l'univers du discours les valeurs de propriétés qui permettent de les caractériser. En EXPRESS, elle est représentée au moyen des attributs. Il existe trois types d'attributs dans PostgreSQL et les valeurs de ces attributs peuvent être de types variés.

1.2.1 Valeurs des attributs

Les valeurs des attributs sont soit des types simples (entiers, réels, booléens, chaînes de caractères, logiques, binaires, énumérations, union de types), soit des types structurés (listes, ensembles, paquets, tableaux) soit encore des entités. Le langage EXPRESS admet qu'un agrégat puisse contenir des valeurs nulles.

Au niveau de PostgreSQL, la connaissance descriptive est représentée par les colonnes des tables qui peuvent être soit de types simples, soit des tableaux de types simples. Les autres formes d'agrégats (listes, ensembles, paquets) n'existent pas dans PostgreSQL. Un tableau PostgreSQL ne peut contenir une valeur nulle. Tous les éléments d'un tableau doivent être significatifs.

Les attributs des entités EXPRESS de type union de type : SELECT (cf. section 2.1.2 du chapitre 1) peuvent être instanciés par une valeur de l'un des types de l'union. Le problème qui se pose est que les colonnes en PostgreSQL doivent être déclarées d'un type bien précis (VARCHAR ou INT4 ou INT8 ou BOOL, etc.) et les valeurs doivent respecter ce type. Nous devons trouver une représentation du type SELECT et de ses valeurs.

Les attributs EXPRESS de type énumération prennent leurs valeurs dans une liste d'énumération. Cette notion de liste de valeur n'existe pas dans PostgreSQL. Elle nécessitera donc d'être traduite.

1.2.1.1 Les modes d'attributs

Le langage EXPRESS définit trois modes d'attributs (cf. section 2 du chapitre 1) :

1. les attributs explicites (ou libres),
2. les attributs dérivés (ou non libres),
3. les attributs inverses.

Le problème se pose au niveau de la représentation des attributs dérivés et des inverses qui, en EXPRESS, ont une valeur calculée non stockée.

Les attributs dérivés étant calculés, il est nécessaire de définir une correspondance sachant que PostgreSQL ne permet pas que la colonne d'une table dépende d'une autre en fonction d'un

quelconque algorithme.

Les attributs inverses sont des attributs qui servent à spécifier la cardinalité de la relation réciproque à une relation de référence donnée. Aucun mécanisme de PostgreSQL ne correspond à ce concept d'EXPRESS. Nous sommes conduits à élaborer des procédés pour mettre en œuvre ce concept.

1.3 Connaissance procédurale

Ce type de connaissance correspond aux règles de raisonnement qui peuvent être appliquées aux différents concepts de l'univers du discours. En EXPRESS, cette connaissance est représentée par des fonctions de dérivations (les attributs dérivés) et des prédicats (contraintes d'intégrités). Nous avons discuté brièvement des attributs dérivés dans le paragraphe précédent.

Concernant les contraintes, le langage EXPRESS est un langage de description très élaboré. On classe les contraintes en deux catégories. Une première catégorie s'appliquant individuellement sur chaque instance d'une entité et la deuxième s'appliquant globalement sur toutes les instances. PostgreSQL comme la plupart des SGBDs aujourd'hui permet de stocker des procédures et fonctions, il offre également la possibilité de définir des contraintes. Les contraintes locales peuvent être définies au moyen de triggers : procédures stockées pour garantir l'intégrité des données. Ils sont appelés automatiquement par le système à la suite d'un certain évènement (suppression d'un tuple, ajout d'un tuple, mise à jour d'un tuple). Les contraintes globales sur une table peuvent être réalisées avec de RULEs. Le seul problème qu'on a est que les RULEs de PostgreSQL ne peuvent s'appliquer que sur une seule table alors que celles d'EXPRESS peuvent s'appliquer sur plusieurs entités.

Après cette présentation des différences entre le langage EXPRESS et SQL/DDL de SGBD PostgreSQL, nous présentons dans la section suivante, les mécanismes que nous avons élaborés en vue de concilier ces deux univers.

2 Notre proposition de règles de correspondances

Nous abordons dans cette section la présentation des solutions élaborées pour permettre la représentation de modèles EXPRESS dans l'univers PostgreSQL. Nous reprenons l'ordre de présentation de la section précédente : connaissance structurelle, connaissance descriptive et connaissance procédurale.

2.1 Connaissance structurelle

2.1.1 Représentation des entités

Pour représenter une entité, nous proposons d'utiliser une table dont les colonnes seront les attributs explicites de l'entité.

2.1.2 Hiérarchisation

Chaque entité étant représentée par une table, ainsi la table de toute entité descendant d'une (ou de plusieurs) entité, héritera de la table (ou des tables) de son ancêtre (ou ses ancêtres). Pour une entité racine, sa table héritera de la table *ROOT_TABLE_ENTITY*.

EXPRESS	POSTGRESQL
ENTITY personne son_nom : **STRING**; son_prenom : **STRING**; **END_ENTITY;** **ENTITY** étudiant **SUPERTYPE OF** (etudiant_salarie) **SUBTYPE OF** (personne); sa_classe : **STRING**; **END_ENTITY;** **ENTITY** salarie **SUPERTYPE OF** (etudiant_salarie) **SUBTYPE OF** (personne); son_salaire : **REAL**; **END_ENTITY;** **ENTITY** etudiant_salarie **SUBTYPE OF** (etudiant, salarie); **END_ENTITY;**	**CREATE TABLE** personne (son_nom **VARCHAR**, son_prenom **VARCHAR**,) **INHERITS** (ROOT_TABLE_ENTITY); **CREATE TABLE** étudiant (sa_classe : **VARCHAR**) **INHERITS** (personne); **CREATE TABLE** salarie (son_salaire **FLOAT**) **INHERITS** (personne); **CREATE TABLE** etudiant_salarie ( ) **INHERITS** (étudiant, salarie); **CREATE TABLE** ROOT_TABLE_ENTITY (RID : **SERIAL8**);

FIG. C.2 – Mapping de la connaissance structurelle du langage EXPRESS en relationnel.

2.1.3 Représentation de l'identité d'une instance

En EXPRESS, chaque instance est identifiée par un identificateur unique représenté par un numéro au sein du fichier d'instances. Compte tenu du fait qu'il est impossible d'utiliser de l'OID (Object IDentifier) que offre PostgreSQL, nous proposons de définir notre propre identifiant jouera la fonction d'OID (i.e. unique dans toutes la base de données, etc.). Pour cela, nous utilisons les séquences qu'offre PostgreSQL. Une séquence est un nombre qui est automatiquement incrémenté par le système. Associée à une colonne (de type ENTIER), celle-ci peut être librement utilisée comme le permette les SGBDs, par exemple, elle peut être référencée comme clé étrangère ou être utilisée dans un tableau.

Nous avons donc décidé de créer une colonne de nom *RID* (pour **R**ow **ID**entifier) liée à une séquence dans toutes les tables associées aux entités du modèle EXPRESS. Pour s'assurer de son *unicité* dans toute la base de données comme pour les OID, cette colonne RID est crée dans la table racine *ROOT_TABLE_ENTITY* héritée par toutes les tables des entités du modèle EXPRESS.
Notons que PostgreSQL offre un type prédéfini nommé*SERIAL8* de type de base ENTIER positif compris entre 0 et $2^{63} - 1$. Ce type permet de créer automatiquement à la fois une SEQUENCE, une contrainte PRIMARY KEY et un INDEX sur la colonne de table ayant ce type (cf. figure C.2). C'est ce type que nous préconisons d'utiliser.

2.1.4 Représentation des entités abstraites dans PostgreSQL

Dans le langage EXPRESS, une entité déclarée abstraite ne peut être instanciée qu'à travers ses sous-types. Nous avons donc représenté cette abstraction par une règle de PostgreSQL appliquée sur la table représentant une entité abstraite. Cette règle permet en effet d'empêcher l'insertion d'instance pour la table déclarée abstraite (cf. figure C.3).

EXPRESS	POSTGRESQL
ENTITY personne **ABSTRACT SUPERTYPE OF** (**ONEOF** (etudiant, enseignant))**;** son_nom : **STRING**; son_prenom : **STRING**; **END_ENTITY**	**CREATE RULE** abstract_personne **AS** **ON** INSERT **TO** personne **DO** **INSTEAD** nothing**;**

FIG. C.3 – Mapping des entité abstrait EXPRESS en relationnel.

2.1.5 Représentation des associations entre classes

Pour la représentation des associations, la solution que nous retiendrons, devrait permettre de supporter le polymorphisme et le calcul des inverses d'une instance. Ces deux contraintes font que l'utilisation d'une clé étrangère entre les deux tables des classes en relation n'est pas possible puisque la norme SQL3 ne supporte pas le polymorphisme. Même si en plus de la clé étrangère, on choisissait de créer une deuxième colonne qui contiendra le nom de la table d'où provient l'instance, le calcul des inverses restent difficile à mettre en œuvre.
La solution, qui répond au mieux aux deux exigences, consiste à définir une table intermédiaire entre les deux tables des entités en relation comme sur la figure C.4. La table intermédiaire ou *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
2. **RID_S** : (**RID S**ource) de type ENTIER est le RID de l'entité d'origine (dont l'attribut fait référence)
3. **TableName_S** : (**TableName S**ource) de type CHAINE est le nom de la table associé à l'entité dont l'attribut fait référence.
4. **RID_D** : (**RID D**estination) de type ENTIER est RID de l'entité de référence.
5. **TableName_D** : (**TableName D**estination) de type CHAINE est le nom de la table associée à l'entité cible.

Dans l'exemple de la figure C.4, l'attribut de type association *sa_personne* défini entre l'entité *personne* et *adresse* (cf. figure C.4a) est traduit par une table d'aiguillage *personne_2_son_adresse* entre les tables de la hiérarchie de *personne* et celle de la table *adresse*.

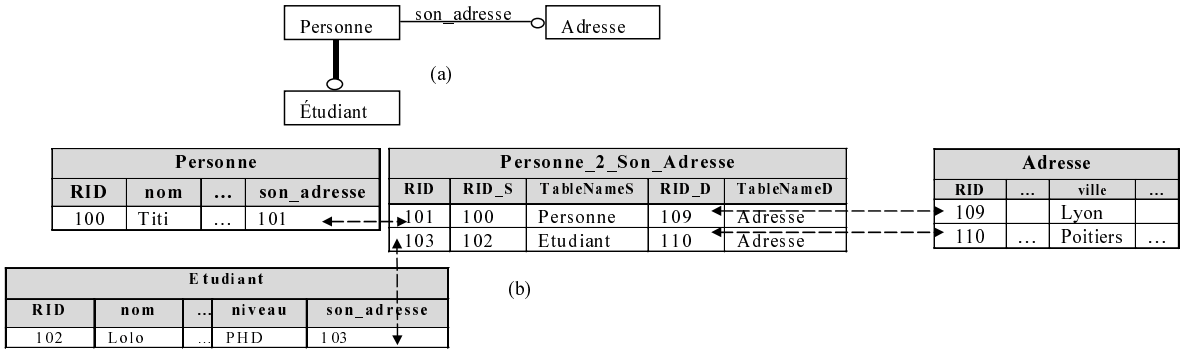


Fig. C.4 – Mapping de la relation d'association EXPRESS en relationnel

Remarque :
Un des principaux inconvénients de cette approche est la gestion de l'intégrité référentielle et l'intégrité des données stockées dans ces tables de données. Aucun mécanisme par défaut dans les SGBDs ne permet de gérer cela. Ceux-ci doivent être définis. Dans la section 2.3.1, nous montrons les mécanismes mises en œuvre pour s'assurer de l'intégrité référentielle des instances.

2.2 Connaissance descriptive

Nous donnons dans cette section, la représentation des attributs dérivés, des attributs inverses et la représentation des types.

2.2.1 Représentation des attributs dérivés

Un attribut dérivé est calculé au moyen d'une fonction. Deux solutions sont possibles :

1. Les attributs dérivés sont aussi associés à des colonnes comme pour les attributs explicites. Mais les valeurs de ces colonnes sont calculées par un trigger qui sera exécuté à chaque mise à jour de la table. Les attributs dérivés pourront être référencés dans d'autres tables. Le problème de cette solution est que la fonction de calcul de l'attribut dérivé peut faire référence à des attributs d'autres entités. Et ainsi si une modification est faite dans l'une des tables de ses entités, la valeur de la colonne ne sera mise à jour. On pourrait résoudre ce problème en définissant des triggers sur toutes les tables impliquées dans le calcul de la fonction. Alors, lorsqu'une de ses tables est mise à jour, la valeur serait calculée. Cette solution pose un autre problème : Comment identifier les tables impliquées ? Seule l'analyse du corps de la fonction peut nous donner la réponse. Malheureusement cette tache est difficile puisqu'elle nécessite de réaliser une analyse complète du code EXPRESS. Une solution extrême serait de définir les triggers sur toutes les tables des entités de la base de données : la conséquence est que toutes les colonnes des attributs dérivés de toutes les tables seront recalculées dès qu'une table est modifiée.

2. La deuxième solution est d'utiliser une vue (**VIEW**) dans laquelle on calcule les attributs

dérivés par une fonction *plpgsql* à chaque accès à la vue. Le problème qui se pose dans cette approche est l'implémentation du corps des fonctions *plpgsql*. La génération du corps des fonctions des attributs dérivés nécessiterait de programmer un compilateur EXPRESS. Une solution pour contourner cet obstacle est de générer automatiquement la signature de la fonction (en renvoyant la valeur NULL). Ensuite de laisser la charge ultérieurement au programmeur d'implémenter manuellement le corps de la fonction *plpgsql* par un **CREATE OR REPLACE**.

Ces deux solutions sont très différentes. Dans tous les cas la génération peut être automatique à travers, l'écriture d'un compilateur sinon le programmeur doit intervenir dans l'écriture du corps de la fonction *plpgsql* ou du trigger. N'ayant pas encore de compilateur EXPRESS totalement opérationnel pour générer le corps de fonctions *plpgsql*, notre choix s'est porté sur la deuxième solution. Par contre, la première solution n'est envisageable que dans le cas où il n'y a jamais ou très peu de mise à jour.

Remarque :
Notons toutefois que cette solution manuelle n'est que provisoire. Nous envisageons d'effectuer la transformation des modèles EXPRESS en PostgreSQL entièrement automatique. Nous venons de réaliser un compilateur EXPRESS que nous avons exploité pour la génération de classes java associé à chaque entité EXPRESS dans lesquelles, nous générons des fonctions pour chacun des attributs dérivés. La génération des fonctions des attributs dérivés constitue une perspective de notre travail.

Dans l'exemple de la figure C.5, l'attribut dérivé *son_initial* de l'entité *personne*, qui permet de calculer l'initial d'un individu, est traduit une fonction *plpgsql* de nom *plpgsql_personne_son_initial* de corps vide et qui est ensuite remplacé par le programmeur après implémenter le corps de la fonction. La fonction de l'attribut dérivé est ensuite renvoyée dans la vue associée à l'entité de l'attribut dérivé.

2.2.2 Représentation des attributs inverses

Rappelons qu'attribut INVERSE permet de recenser toutes les instances des entités référençant une instance donnée à partir d'un attribut précis. Un attribut inverse est en d'autre terme la réciproque de l'association. Les problèmes à résoudre pour la représentation des attributs inverses sont :

1. la représentation de la relation directe et inverse,
2. la représentation des valeurs des attributs inverses (qui est une collection d'instances).

Concernant le premier problème. Il suffit d'ajouter à chaque lien figurant dans la table d'association le RID Source (RID_S) et le nom de la table Source (TableName_S) pour représenter par une unique ligne à la fois le lien direct et son inverse. Cette solution est donc préférable à l'utilisation de deux tables (lien direct et lien inverse) qui seraient redondantes. C'est bien ce choix que nous avons fait dans la section 2.1.5 pour la représentation des associations (cf. figure

EXPRESS	POSTGRESQL
ENTITY personne son_nom : **STRING**; son_prenom : **STRING**; **DERIVE** son_initial : **STRING** := son_nom[1]+son_prenom[1]; **END_ENTITY;**	**CREATE TABLE** personne (son_nom **VARCHAR,** son_prenom **VARCHAR**)**INHERITS**(ROOT_TABLE_ENTITY); **CREATE VIEW** personne_v **AS** **SELECT** son_nom, son_prenom, plpgsql_personne_son_initial(RID) **AS** son_initial **FROM** personne ;

	--Générer automatiquement.
	CREATE OR REPLACE FUNCTION Plpgsql_personne_son_id(**INT8**) **RETURNS VARCHAR AS** '
	RETURN NULL ;
	END;
	'**LANGUAGE** 'plpgsql';
	
	--Ecrit par le programmeur
	CREATE OR REPLACE FUNCTION plpgsql_personne_son_ initial (**INT8**)
	RETURNS VARCHAR AS '
	DECLARE
	RID **ALIAS FOR** $1 ;
	Tuple **RECORD** ;
	BEGIN
	SELECT INTO tuple.nom,prenom **FROM** personne **WHERE** RID =RID ;
	RETURN SUBSTR(tuple.nom,1,1) || **SUBSTR**(tuple.prenom,1,1);
	END;
	'**LANGUAGE** 'plpgsql';

FIG. C.5 – Mapping des attributs dérivés EXPRESS en relationnel.

C.4).

Concernant le deuxième problème, seuls certains attributs inverses sont déclarés explicitement en EXPRESS. Les autres sont néanmoins calculables en EXPRESS par une fonction prédéfinie "USED_IN". La différence entre attributs inverses déclarés et non déclarés est donc seulement syntaxique. La solution retenue ci-dessus traite également de façon identique les associations inverses qu'elles correspondent à des attributs inverses explicites ou implicites. Dans ce cas, un "SELECT" dans la *table d'aiguillage* permettra de les calculer. Concernant le stockage éventuel du résultat de cette fonction on peut au choix ne pas le représenter mais de les retourner dans une vue SQL, ou bien de le représenter uniquement les attributs inverses explicites et les mettre à jour chaque fois qu'une entité référençante est mise à jour. Il n'y a pas de critère fort pour choisir la deuxième solution, nous avons donc décidé en application du principe général de la non-redondance, de ne pas les représenter mais de les calculer au moyen d'une vue (cf. figure C.6). Pour cela, nous avons implémenté l'équivalent de la fonction *USED_IN* en EXPRESS dans le langage procédural du SGBD PostgreSQL (plpgsql).

Dans l'exemple de la figure C.6, nous avons un attribut inverse *locataire* dans l'entité *adresse* qui permet de donner les personnes résidantes à une adresse donnée. Celui-ci est retourné dans la vue SQL de l'entité *adresse* calculer grâce à la fonction *plpgsql USED_IN*.

EXPRESS	POSTGRESQL

ENTITY Adresse
appt : **STRING**;
rue : **STRING**;
ville : **STRING**;
codepostal : **STRING**;
INVERSE
locataire : **SET**[1:?] **OF** personne **FOR** son_adresse
END_ENTITY;
	CREATE VIEW adresse_v **AS**
	SELECT
	appt,rue,ville,codepostal,
	USEDIN(RID,'personne','son_adresse') **AS** locataire
	FROM adresse ;
	
	CREATE FUNCTION USEDIN(**INT8,VARCHAR, VARCHAR) RETURNS INT8[] AS'**
	DECLARE
	RID **ALIAS FOR** $1 ;-- le RID de l'entité
	Entité **ALIAS FOR** $2; -- le nom de l'entité source
	Attribut **ALIAS FOR** $3; --le nom de l'attribut
	BEGIN
	SELECT RID_S, TableName_S **FROM** Entité_2_Attribut **WHERE** RID_D=RID
	...
	END;
	' **LANGUAGE** 'plpgsql';

FIG. C.6 – Mapping de l'attribut inverse EXPRESS en relationnel.

2.2.3 Représentation des types

Dans cette partie nous établissons la correspondance entre les types de base d'EXPRESS dans PostgreSQL puis de la représentation des autres catégories de types du langage(le type SELECT, le type énuméré, les agrégats).

2.2.3.1 Correspondance des types de bases

La correspondance des types de bases du langage EXPRESS en PostgreSQL est présentée dans la figure C.7.

Types EXPRESS	Types PostgreSQL
BINARY	**BIT**
BOOLEAN	**BOOLEAN** (avec TRUE ou 't', 'true', 'y' 'yes' '1' et FALSE ou 'f' 'false' 'n' 'no' '0')
LOGICAL	**BOOLEAN** (null = unknow)
NUMBER	**DECIMAL**
REAL	**FLOAT8**
INTEGER	**INT8**
STRING	**VARCHAR** (Chaîne de caractères de longueur variable)
STRING [N]	**VARCHAR(N)** (Chaîne de caractères de longueur N variable)
STRING [N] FIXED	**CHAR(N)** (Chaîne de caractères de longueur N fixe)

FIG. C.7 – Mapping des types de base EXPRESS en relationnel.

2.2.3.2 Type énuméré

Le type énuméré est un type dont les valeurs sont définies dans une liste. En EXPRESS, les éléments de cette liste sont ordonnées et des opérations comparaisons ($\leq, \geq, ...$) peuvent être appliquées sur ceux-ci.

Deux solutions sont possibles pour la représentation du type énuméré :

- la première solution consiste à définir le type énuméré comme une chaîne (**VARCHAR**) en PostgreSQL et d'associer à toute colonne d'attribut ayant pour domaine un type énuméré une contrainte **CHECK** qui vérifiera la conformité des valeurs représentées dans la colonne.
- la deuxième solution consiste à créer une table pour chaque type énuméré dans laquelle sont insérés les éléments de l'énumération. Afin de conserver l'ordre entre les éléments de la liste, une colonne supplémentaire ("rang") de type ENTIER peut être définie et qui contiendra l'indice de chaque élément. La colonne d'un attribut de type énuméré sera une clé étrangère sur la table de l'énumération.

Dans notre prototype nous avons opté pour la deuxième approche à cause de sa simplicité d'implémentation et de mise à jour.

Dans l'exemple de la figure C.8, le type énuméré *enum_ niveau* est traduit en une table (*enum_ niveau*) puis initialisée avec les différents éléments du type énuméré. Les attributs des entités de ce type énuméré sont traduits en clé étrangère sur la table du type énuméré. C'est justement le cas de l'attribut *niveau* de l'entité *étudiant*. Remarquez que pour définir un ordre entre les éléments du tableau, nous avons introduit la colonne "rang" dans la table des types énumérés.

EXPRESS	POSTGRESQL
TYPE enum_niveau :=**ENUMARATION OF** (Licence, Maîtrise, Doctorat); **END_TYPE** ; **ENTITY** étudiant **SUBTYPES OF** (personne) niveau: enum_niveau; **END_ENTITY;**	**CREATE TABLE** enum_niveau (RID **INT8** élément: **VARCHAR**, rang **INTEGER**)**INHERITS**(ROOT_TABLE_DEFINED_TYPE); **INSERT INTO** enum_niveau **VALUES**('Licence,0) **INSERT INTO** enum_niveau **VALUES**('Maîtrise,1) **INSERT INTO** enum_niveau **VALUES**('Doctorat,2) **CREATE TABLE** étudiant (niveau **INT8** **FOREIGN KEY** niveau **REFERENCES** (enum_niveau , RID))**INHERITS**(personne);

Fig. C.8 – Mapping du type énuméré EXPRESS en relationnel.

2.2.3.3 Type SELECT

Le type SELECT ou union de types est un type particulier du langage EXPRESS qui autorise un attribut donné à prendre ses valeurs dans l'un des types du type SELECT. Les types listés dans le type SELECT peuvent être de n'importe quel type définissable en EXPRESS (type simple, entité, type énuméré, type agrégat, ...). Deux implémentations sont possibles pour la représentation du type SELECT :

1. Une première approche consiste à créer autant de colonnes pour un attribut de type SELECT qu'il y a de types dans le SELECT, i.e. une colonne pour chaque type du SELECT. Pour les types entités du SELECT on crée exceptionnellement deux colonnes (RID, TableName) pour pouvoir supporter le polymorphisme. Les inconvénients se situent (1) au niveau du calcul des inverses des relations, et (2) au niveau du calcul de la valeur de la colonne du type effectivement renseignée car elle nécessite de parcourir toutes les colonnes

créées et de le tester un à un pour déterminer celle qui n'est pas nulle.

2. Dans la deuxième approche, (1) on crée une table spécifique pour chaque type du SELECT. Pour les types simples, la table correspondante est constituée de deux colonnes (RID, Valeur), et (2) ensuite, comme pour la représentation des associations 2.1.5, on crée une table intermédiaire (ou *table d'aiguillage*) entre la table de l'entité qui définit l'attribut de type SELECT et les tables des types du SELECT. La structure de la table d'aiguillage est constituée de cinq colonnes comme sur la figure C.9. Dans la table de l'entité ayant l'attribut de type SELECT, on crée une clé étrangère vers la table d'aiguillage. Les colonnes *RID_D* et *TableName_D* permettent respectivement de stocker l'identifiant (RID) et le nom de la table où est stockée la valeur de l'attribut. Les colonnes *RID_S* et *TableName_S* permettent respectivement de stocker l'identifiant des instances de la table de l'entité ayant l'attribut de type SELECT et le nom de la table.

Notre choix s'est porté sur la deuxième solution puisqu'elle répond a toutes les exigences que nous nous sommes fixés (calcul des inverses, etc.).

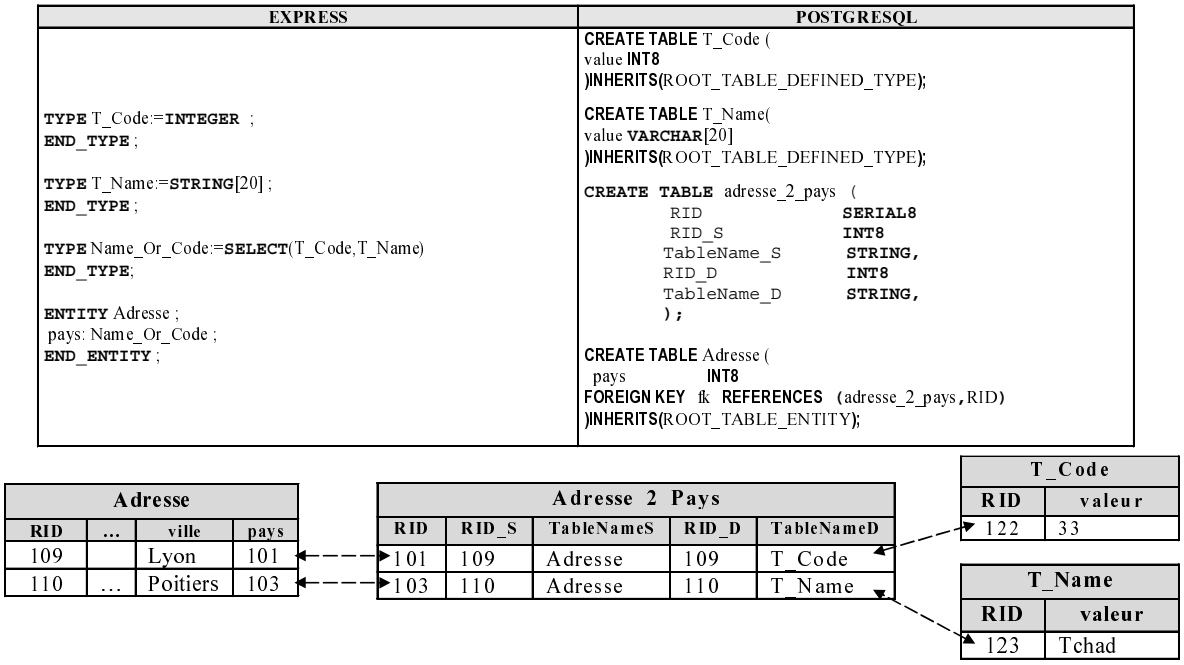


FIG. C.9 – Mapping du type SELECT EXPRESS en relationnel.

Dans l'exemple de la figure C.9, l'attribut *pays* de type SELECT *Name_Or_Code* est traduit en une table d'aiguillage *adresse_2_pays*. Tous les types du SELECT (*T_Code*, *T_Name*) sont aussi traduits en des tables.

2.2.3.4 Les agrégats

PostgreSQL offre la possibilité de définir des tableaux de type simple (INT8, VARCHAR, etc.). Mais toutefois PostgreSQL exige qu'aucun élément du tableau ne soit nul, ce qui est autorisé en EXPRESS. L'approche de représentation que nous allons choisir doit prendre en considération cette caractéristique des agrégats EXPRESS. Dans la suite, nous présenterons séparément la représentation des agrégats de type simple et des agrégats de type complexe (énumération, union de types, entité, ...).

2.2.3.4.1 agrégats de type complexe

Étant donné que les entités, les types énumérations, les types SELECT sont représentés par des tables, une solution possible de la représentation des ces types est (1) de créer une table pour chaque agrégat à l'identique des tables des associations et du type SELECT, i.e. une table bilatère entre les tables des entités en relation, et (2) de créer une colonne de type tableau qui contiendra l'identifiant (RID) des instances de ces types complexes. L'avantage de cette représentation est qu'elle nous permet également de calculer les relations inverses des instances des agrégats.

Pour résoudre le problème des valeurs nulles, on peut envisager de prendre zéro comme valeur nulle et de s'assurer que dans tout le système il n'ait aucune instance ayant pour identifiant la valeur zéro. Ceci est réalisable facilement en initialisant les valeurs de départ des SEQUENCEs PostgreSQL à une valeur supérieure à zéro (1 par exemple).

2.2.3.4.2 agrégat de type simple

Pour la représentation des agrégats de type simple, deux solutions sont possibles :

(1) Utiliser directement les tableaux de PostgreSQL et, selon le type simple, de fixer une valeur qui servira de valeur nulle (ENTIER -999999999, etc.). Les inconvénients de cette solution sont d'une part que les éléments des tableaux doivent être analysés pour repérer les valeurs nulles avant de faire des opérations sur le tableau (l'affichage par exemple, ...), et d'autre part qu'on a des difficultés à attribuer une valeur nulle pour le type booléen.

(2) Créer pour chaque agrégat, une table (RID, VALEUR), et une colonne de type tableau d'entier qui contiendra l'identifiant des éléments de l'agrégat correspondant à une instance. Comme pour les agrégats de types complexes, les valeurs nulles seront désignées par une identifiant de valeur 0.

Dans l'exemple de la figure C.10, les attributs *ses_amis* et *ses_notes* de types agrégats sont traduits respectivement en des tables d'aiguillages *personne_2_ses_amis* et *Étudiant_2_ses_notes*. Notons que sur le zéro dans la liste de la colonne *ses_amis* de la table *personne* désigne une valeur nulle. Remarquez aussi qu'il n'existe pas dans la table *personne_2_ses_amis* une entrée de valeur zéro.

EXPRESS	POSTGRESQL
ENTITY Personne ses_amis : **SET OF** Personne ; **END_ENTITY** ; **ENTITY** Etudiant ses_notes: **LIST OF INTEGER**; **END_ENTITY** ;	**CREATE TABLE** personne_2_ses_amies(RID **SERIAL8** RID_S **INT8** TableName_S **STRING,** RID_D **INT8** TableName_D **STRING,**) **CREATE TABLE** Personne(RID **SERIAL8**, ses_amis **INT[8]**) **CREATE TABLE** Etudiant_2_ses_notes(RID **SERIAL8**, value **VARCHAR**(20)) **CREATE TABLE** Etudiant ses_notes **INT8**[],)

Personne

RID	nom	...	son_amis
100	Titi	...	{1,0,2}

personne_2_ses_amis

RID	RID_S	TableNameS	RID_D	TableNameD
1	100	Personne	99	Etudiant
2	100	Personne	102	Etudiant

etudiant_2_ses_notes

RID	Valeur
1	10
2	09
3	15

Etudiant

RID	nom	...	niveau	son_notes
99	Remi		PHD	{1,2,0}
102	Lolo	...	PHD	{0,3,0}

FIG. C.10 – Mapping des types agrégats EXPRESS en relationnel.

2.3 Connaissance procédurale

Cette connaissance est constituée en EXPRESS des fonctions de calcul des attributs dérivés et des contraintes d'intégrité. Vu que nous avons déjà discuté de la représentation des attributs dérivés dans la section 2.2.1, nous nous intéresserons dans cette section que de la représentation des contraintes.

Les contraintes peuvent être classées en deux catégories :

1. les contraintes référentielles ou d'intégrités référentielles : ce sont les règles qui doivent être mises en oeuvre pour assurer l'intégrité des données de la base de données, et
2. les contraintes portant sur des valeurs d'attributs.

2.3.1 Intégrité référentielle

Les différentes solutions retenues jusqu'à présent nous amenaient soient à la création de tables de valeurs ou de tables d'aiguillages, à l'utilisation de tableaux de RIDs, de clés étrangères etc. Cette section indique les contraintes référentielles qu'il faut mettre en place en vue d'assurer l'intégrité des données de la base de données. La définition de ces contraintes est très importante car si elle n'existe pas très vite notre système atteindra un niveau d'incohérence total. Les contraintes référentielles devant être assurées portent sur les trois catégories de colonnes :

1. les colonnes des attributs dont les types sont représentées par des RIDs (type énuméré, type select, type entité),
2. les colonnes des attributs de types agrégats,
3. les colonnes des tables d'aiguillages.

2.3.1.1 Colonne d'attributs à valeur RID (type entité, type select, type énuméré)

Pour assurer l'intégrité référentielle des colonnes des attributs dont les types sont une entité ou un type SELECT ou un type énuméré, nous les déclarons comme des clés étrangères (**CONSTRAINT ... FOREIGN KEY**) sur les tables auxquelles elles font référence.

2.3.1.2 Colonne des attributs de type agrégat

PostgreSQL n'admet pas la déclaration des éléments d'un tableau comme clé étrangère d'une table. Nous créons une fonction plpgsql qui vérifie l'intégrité référentielle des éléments du tableau par une contrainte check (**CONSTRAINT ... CHECK**). Cette fonction a quatre paramètres : le nom de la table, le nom de l'attribut, le nom de la table d'association et la valeur de l'attribut.

La figure C.11 présente un exemple de définition de contraintes sur une table d'aiguillage pour assurer l'intégrité des données.

2.3.2 Les contraintes explicites du langage EXPRESS

Ces contraintes sont les suivantes :
- la contrainte locale (WHERE RULE),
- la contrainte d'unicité (UNIQUE),

EXPRESS	POSTGRESQL
ENTITY Personne ; ses_amis : **LIST OF** Personne, **END_ENTITY** ;	**CREATE TABLE** personne_2_ses_amies(RID SERIAL8 RID_S INT8 TableName_S STRING, RID_D INT8 TableName_D STRING,) **CREATE TABLE** Personne(ses_amis **INT8[]** ,)**INHERITS**(ROOT_TABLE_ENTITY); **ALTER TABLE** Personne **ADD CONSTRAINT** CK_ses_amis **CHECK** (CHECK_ARRAY('personne','ses_amis','personne_2_ses_amies',ses_amis)
	CREATE FUNCTION CHECK_ARRAY **(VARCHAR,VARCHAR,VARCHAR,INT8||) RETURNS BOOLEAN AS'** **DECLARE** la_table **ALIAS FOR** $1; --le nom de la table le_champ **ALIAS FOR** $2; --le nom du champ table_reference **ALIAS FOR** $3 ; -- la table de référence la_valeur **ALIAS FOR** $4; --la valeur du champ **BEGIN** .... ' **LANGUAGE** 'plpgsql';

Fig. C.11 – Contraintes d'intégrités sur une table d'aiguillage

– la contrainte globale (RULE).

2.3.2.1 Contrainte locale (WHERE RULE)

En EXPRESS, cette règle s'applique sur chaque instance en vérifiant la cohérence de certains attributs. En PostgreSQL, elle peut être traduite par une contrainte **CHECK**, qui a la même fonctionnalité. Ainsi tous les WHERE RULE d'une entité et des types utilisateurs seront traduites par des **CONSTRAINT CHECK**, dont les conditions seront des prédicats (fonctions) implémentés en plpgsql. Lorsqu'un attribut a pour type un type utilisateur, la contrainte du type utilisateur sera sur la colonne de l'attribut. Quand une table lui est associée (cas où il est dans un type SELECT) la contrainte est sur la colonne "*value*".

Dans l'exemple de la figure C.12 la contrainte locale *wr* sur l'attribut *salaire* est traduite en une contrainte *CHECK* en relationnel. La contrainte *CHECK* fait appel à la fonction *Check_ Salaire* implémentée manuellement par le programmeur.

2.3.2.2 Contrainte d'unicité (UNIQUE)

En EXPRESS, la contrainte d'unicité s'applique sur l'ensemble de la population des instances. Elle assure l'unicité de la valeur d'un attribut sur la population de l'entité où elle est déclarée mais aussi dans toutes les sous entités. En PostgreSQL, la contrainte d'unicité s'applique uniquement à la table où elle est déclarée. Les instances des sous-tables n'étant pas référencées dans la table mère. Cela signifie que la contrainte d'unicité validera uniquement les instances de la table mère ou uniquement les instances d'une des tables filles, mais elle ne vérifiera pas dans l'ensemble des instances de toutes les tables. Nous pouvons donc trouver une instance dans la table mère et

EXPRESS	POSTGRESQL
ENTITY salarie SUPERTYPE OF (etudiant_salarie) SUBTYPE OF (personne); salaire : FLOAT; WHERE wr : salaire >= 0.0 ; END_ENTITY;	CREATE TABLE salarie (salaire FLOAT, CONSTRAINT wr CHECK Check_Salaire(salaire))) INHERITS (personne);
	CREATE FUNCTION Check_Salaire (FLOAT) RETURNS BOOLEAN AS' DECLARE salaire ALIAS FOR $1;--le salaire BEGIN RETURN (salaire >= 0) END ' LANGUAGE 'plpgsql';

Fig. C.12 – Mapping des contraintes locales EXPRESS en relationnel.

une instance de la table fille avec la même valeur de la colonne. Cette remarque est aussi valable pour la clé primaire (**PRIMARY KEY**). L'unicité de la clé primaire est assurée seulement pour la table où elle est déclarée. Pour assurer cette unicité sur l'ensemble de la population, nous la déclarons comme une contrainte CHECK sur la colonne et une fonction *plpgsql* (*is_unique*) qui vérifiera s'il existe une instance dans l'une des tables de la hiérarchie qui a la même valeur. Les contraintes sur les attributs (**CONSTRAINT ... CHECK**) sont transmises par l'héritage entre tables et donc la contrainte sera propagée de la table vers ses sous-tables.

EXPRESS	POSTGRESQL
ENTITY personne SUPERTYPE OF son_num_SS : STRING; son_nom : STRING; son_prenom : STRING; UNIQUE ur1 : son_num_SS; END_ENTITY;	CREATE TABLE personne (son_num_SS : VARCHAR, son_nom : VARCHAR, son_prenom : VARCHAR, CONSTRAINT ur CHECK (is_unique('personne','son_num_SS',son_num_SS)));

	CREATE FUNCTION is_unique (VARCHAR,VARCHAR,INTEGER) RETURNS BOOLEAN AS'
	DECLARE
	la_table ALIAS FOR $1;
	le_champ ALIAS FOR $2;
	la_valeur ALIAS FOR $3;
	BEGIN
	LA REQUETE DANS LA BD
	IF TROUVE THEN
	RETURN true,
	ELSE
	RETURN false;
	END IF;
	END;
	' LANGUAGE 'plpgsql';

Fig. C.13 – Mapping de la contrainte UNIQUE EXPRESS en relationnel.

Un exemple illustrant le mapping de la contrainte UNIQUE est présenté dans la figure C.13. La contrainte UNIQUE *url* est traduite en une contrainte *CHECK*.

2.3.2.3 Contrainte globale (RULE)

Cette contrainte en EXPRESS s'applique à l'ensemble d'une population d'entités d'un schéma donné et permet de valider les instances des entités. En PostgreSQL, elle peut être exprimée par un CREATE RULE. La contrainte globale RULE de PostgreSQL ne peut s'appliquer que sur une seule table. Ce qui n'est pas le cas avec EXPRESS où elle peut s'appliquer sur plusieurs entités à la fois. La vérification de RULE globale multi-entité reste donc un problème ouvert (cf. figure C.14). Précisons toutefois qu'il n'en existe pas dans le modèle d'ontologie PLIB auquel nous appliquerons toutes ces règles de correspondances.

EXPRESS	POSTGRESQL
RULE nombre_max_etudiant FOR (etudiant_salarie); WHERE max_de_10 SIZEOF(etudiant_salarie)<=10; END_RULE;	CREATE RULE nombre_max_etudiant AS ON INSERT TO etudiant_salarie WHERE Nbre_Etudiant_Salarie()>10 DO NOTHING

Fig. C.14 – Mapping contrainte globale EXPRESS en relationnel

2.4 Convention des noms

Dans cette partie nous présentons les conventions sur le nom des tables, le nom des vues, le nom des procédures stockées, etc. Ceci afin de permettre aux programmeurs de connaître les noms PostgreSQL apparaissant dans la base de données générée à partir des noms figurant dans les schémas conceptuels EXPRESS.

2.4.1 Structure

Tables et vues

Compte tenu du fait que les entités EXPRESS sont associées à des tables et des vues et que PostgreSQL n'admettant pas qu'il y ait dans la base de données deux objets portant le même nom, nous avons décidé d'attribuer des noms différents aux tables et aux vues des entités. Le nom d'une table est la concaténation du nom de l'entité et de "$\_E$" et celui d'une vue est le nom de l'entité suivie de "$\_V$".

Table d'aiguillage

Une table d'association portera le nom de l'entité concaténé à une "$\_2\_$" et suivie du nom de l'attribut dont le type est une autre entité ou un SELECT.

Table des types utilisateurs

Le nom des tables des types utilisateurs est la concaténation des noms du types et suivie de "$\_T$".

Table des valeurs des agrégats

Le nom des tables de valeurs des agrégats est formé par la concaténation de "*array_value*" au nom de l'entité et du nom de l'attribut.

2.4.2 Contraintes

Contrainte FOREIGN KEY
Toutes les FOREIGN KEY sont nommées de la manière suivante : Concaténation de "*fk_*" plus du nom de l'attribut.

Contrainte CHECK
Le nom d'une contrainte *CHECK* correspondante à une clause unique aura la forme "*ur_*" plus de nom de l'attribut. Pour une contrainte *CHECK* correspondante à la vérification des agrégats de référence ou de SELECT ou d'énumération sera "ck_" plus le nom de l'attribut.

Fonction plpgsql
Les fonctions plpgsql correspondantes au WHERE RULE auront pour nom "*check_*" suivi du nom du type utilisateur ou du nom de l'attribut et du nom de la WHERE RULE. Les fonctions des attributs dérivés sont formées de la façon suivante : "*plpgsql_NomEntite_NomAttribut*".

3 Conclusion

Nous avons présenté dans cet annexe les règles de correspondances EXPRESS en relationnel en vue de générer la structure des modèles logiques de bases de données à partir de modèle EXPRESS. Ces règles de correspondances ont été justement utilisées pour la génération automatiquement de la structure des tables de la partie *ontologie* et *méta-schéma* de notre prototype de SGBDBO (cf. chapitre 4 section 1).

Glossaire

PLIB	Parts Library (PLIB) - Norme ISO 13584
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	RDF Schema
BSU	Basic Sémantique Unit
URI	Uniform Resource Identifier
BDBO	Base de Données à Base Ontologique
DBO	Données à Base Ontologique
OL	Ontologie Linguistique
OC	Ontologie Conceptuelle
IEC	International Electrotechnical Commission
SGBD	Système de Gestion de Bases de Données
BD	Base de Données
MC	Modèle Conceptuel
IDM	Ingénierie Dirigée par les Modèles
MDA	Model Driven Architecture
API	Application Programming Interface
SDAI	Standard Data Access Interface - Part 22 (ISO 10303-22 :1998)
MOF	Meta-Object Facility est un standard de l'OMG
XML	Extensible Markup Language
STEP	Standard for the Exchange of Product Model Data Norme ISO 10303
E/A	Entité / Association
UML	Unified Modeling Language

Explicitation de la sémantique dans les bases de données :

Base de données à base ontologique et le modèle OntoDB

Présenté par

Hondjack DEHAINSALA

Directeurs de thèse

Guy PIERRA, Ladjel BELLATRECHE

Résumé. Une ontologie de domaine est une représentation de la sémantique des concepts d'un domaine en termes de classes et de propriétés, ainsi que des relations qui les lient. Avec le développement de modèles d'ontologies stables dans différents domaines, OWL dans le domaine du Web sémantique, PLIB dans le domaine technique, de plus en plus de données (ou de métadonnées) sont décrites par référence à ces ontologies. La taille croissante de telles données rend nécessaire de les gérer au sein de bases de données originales, que nous appelons *bases de données à base ontologique* (BDBO), et qui possèdent la particularité de représenter, outre les données, les ontologies qui en définissent le sens. Plusieurs architectures de BDBO ont ainsi été proposées au cours des dernières années. Les schémas qu'elles utilisent pour la représentation des données sont soit constitués d'une unique table de triplets de type (sujet, prédicat, objet), soit éclatés en des tables unaires et binaires respectivement pour chaque classe et pour chaque propriété. Si de telles représentations permettent une grande flexibilité dans la structure des données représentées, elles ne sont ni susceptibles de passer à grande échelle lorsque chaque instance est décrite par un nombre significatif de propriétés, ni adaptée à la structure des bases de données usuelles, fondée sur les relations n-aires. C'est ce double inconvénient que vise à résoudre le modèle OntoDB. En introduisant des hypothèses de typages qui semblent acceptables dans beaucoup de domaine d'application, nous proposons une architecture de BDBO constituée de quatre parties : les deux premières parties correspondent à la structure usuelle des bases de données : *données* reposant sur un schéma logique de données, et *méta-base* décrivant l'ensemble de la structure de tables. Les deux autres parties, originales, représentent respectivement les ontologies, et le méta-modèle d'ontologie au sein d'un méta-schéma réflexif. Des mécanismes d'abstraction et de nomination permettent respectivement d'associer à chaque donnée le concept ontologique qui en définit le sens, et d'accéder aux données à partir des concepts, sans se préoccuper de la représentation des données. Cette architecture permet à la fois de gérer de façon efficace des données de grande taille définies par référence à des ontologies (données à base ontologique), mais aussi d'indexer des bases de données usuelles au niveau connaissance en leur adjoignant les deux parties : *ontologie* et *méta-schéma*. Le modèle d'architecture que nous proposons a été validé par le développement d'un prototype opérationnel implanté sur le système PostgreSQL avec le modèle d'ontologie PLIB. Nous présentons également une évaluation comparative de nos propositions avec les modèles présentés antérieurement.

Mots-clés : Base de données, ontologie, données à base ontologique, base de données à base ontologique, PLIB, RDF Schéma, OWL
